P2R: Implementation of Processing in Racket

Hugo Filipe Fonseca Correia

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson:	Prof. Dr. João António Madeiras Pereira
Supervisor:	Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member of the Committee:	Prof. Dr. João Coelho Garcia

June 2015

”As palavras não fazem o homem compreender,
é preciso fazer-se homem para entender as palavras”
— Herberto Helder

Acknowledgments

There are many people to thank for the development of this work, many of them without whom it would not have been possible to complete. 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.

Firstly, I would like to thank professor António Leitão. His knowledge, vision, and assistance were a fundamental factor for the development of this work, and its quality is due to him. I would also like to thank Fundação para a Ciência e a Tecnologia and INESC-ID for the opportunity to take part and contribute to project Rosetta.

I would like to thank my colleagues from the Architecture and Computing group, that gave me critical feedback and help along the way.

A special thanks to my parents for their unconditional love and support, and to my friends, which have joined me on this journey and made it possible.

Resumo

O Processing é uma linguagem de programação criada para o ensino de programação nas áreas de multimédia, design e arquitetura. Ao longo do tempo, a linguagem tornou-se muito popular nestas comunidades, devido a uma forte componente pedagógica e a um vasto conjunto de primitivas de modelação geométrica. Contudo, apesar de ter alcançado grande visibilidade e massa crítica, a linguagem Processing continua a ser uma linguagem com pouca aplicabilidade na área da arquitetura, visto que nenhum software de CAD (Computer-Aided Design) a suporta.

De modo a preencher esta lacuna, propomos uma implementação de Processing que consiga gerar modelos nas ferramentas de CAD mais utilizadas. Deste modo, arquitetos e designers que tenham aprendido Processing podem tirar partido das capacidades gráficas e pedagógicas da linguagem para desenvolver modelos arquitetónicos numa ferramenta CAD.

Para tal, iremos tirar partido da biblioteca Rosetta, criada para desenho generativo e desenvolvida na plataforma Racket. A biblioteca Rosetta suporta múltiplas linguagens de programação, nomeadamente AutoLISP, Racket, Python, etc. Simultaneamente, o Rosetta permite gerar modelos geométricos em várias ferramentas CADs, entre elas, o AutoCAD, o Rhinoceros 3D, e o SketchUp. Adicionalmente, o Rosetta implementa um leque de primitivas de modelação apropriadas para o trabalho arquitetónico.

Este trabalho apresenta uma implementação de Processing para Racket, que transforma Processing em código Racket equivalente, tirando proveito do Rosetta para aceder às diferentes ferramentas de CAD e a um conjunto alargado de primitivas de modelação geométrica. Desta forma, usando a nossa implementação arquitectos podem usar a linguagem Processing para trabalho arquitectonico na sua ferramenta de CAD favorita.

Palavras-chave: Processing, Racket, Compiladores, Desenho Generativo, Interoperabilidade

Abstract

The Processing language was created to teach programming to the design, architecture, and electronic arts communities. The language has become very popular among these communities, mainly due to its pedagogical and graphical features. However, despite its success, Processing remains a niche language with limited applicability in the architectural realm, as no CAD (Computer-Aided Design) application supports Processing. As a result, architects that have learnt Processing are unable to use the language in the context of modern, script-based, architectural work.

To address this issue, we propose an implementation of Processing that runs in the context of the most used CAD tools. This way, architects and designers that have learnt the Processing language can take advantage of its educational and graphical capabilities to develop architectural work in a CAD environment.

To connect Processing with a CAD application, we will take advantage of Rosetta, a Generative Design library implemented on top of Racket. Rosetta allows programs written in AutoLISP, Racket, Python, etc., to generate models in different CAD applications, such as AutoCAD, Rhinoceros 3D, and SketchUp. Moreover, Rosetta implements a wide range of modelling primitives specifically tailored for architectural work.

This work presents an implementation of Processing for the Racket platform, that transforms Processing into equivalent Racket source code, taking advantage of Rosetta to access several CAD backends and a wider set of modelling primitives. Therefore, using our implementation, architects can use Processing to generate models for architectural work in their favourite CAD application.

Keywords: Processing, Racket, Compilers, Generative Design, Interoperability

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xiii
List of Figures xvi
List of Acronyms xvii

1 Introduction **1**
1.1 Processing 1
1.2 Rosetta 4
1.3 Goals 5
1.4 Outline 6

2 Related Work **7**
2.1 Compiler Design 7
2.1.1 Recursive-Descent Parsing 8
2.1.2 Lex/Yacc 9
2.1.3 ANTLR 10
2.1.4 Additional Systems 11
2.2 Language Workbench 11
2.2.1 MPS 13
2.2.2 Spoofax 14
2.2.3 Racket 15
2.2.4 Alternative Language Workbenches 17
2.3 Source-to-Source Compilers 17
2.3.1 Processing.js 18
2.3.2 Processing.py & Ruby-Processing 18
2.3.3 ProfessorJ 18
2.4 Comparison 19

3 Implementing Processing in Racket **21**
3.1 Module Decomposition 21

3.1.1 Compiler Modules . . . 22
3.1.2 Language Module . . . 23
3.2 Compilation Process . . . 23
3.2.1 Parsing Phase . . . 24
3.2.2 Code Analysis Phase . . . 24
3.2.3 Code Generation Phase . . . 25
3.3 Runtime . . . 29
3.3.1 Primitive Operations & Built-in Classes . . . 29
3.3.2 Drawing Primitives . . . 29
3.3.3 Types in the runtime . . . 31
3.4 Interoperability . . . 33
3.5 Integration with DrRacket . . . 35
3.5.1 Comparing both IDEs . . . 35
3.5.2 Adapting Processing to DrRacket . . . 35

4 Evaluation **37**
4.1 Connecting Processing with CAD applications . . . 37
4.2 Augmenting Processing's 3D modelling primitives . . . 39
4.3 Integration with DrRacket . . . 42
4.4 Interoperability with Racket . . . 44
4.5 Performance . . . 46

5 Conclusions **49**
5.1 Future Work . . . 50

Bibliography **55**

List of Tables

2.1 Comparison of the different parsing alternatives 11
2.2 Comparison of different implementations of source-to-source compilers 19

3.1 Racket to Processing translation rules for identifiers 34
3.2 Comparison between the Processing Development Environment and DrRacket 35

4.1 Performance comparison for the different test scenarios 48

List of Figures

1.1 The Processing Development Environment 2
1.2 Rosetta: The Generative Design tool 4

2.1 DrRacket: The pedagogic IDE 16

3.1 Main module decomposition 22
3.2 Overview of the compilation pipeline 24
3.3 Macro to generate Processing's functions 26
3.4 Macro to generate Processing's `if` statements 27
3.5 Processing's `while` loop implemented in Racket using a named `let` 27
3.6 Processing's `for` loop implemented in Racket using a named `let` and `let/ec` 27
3.7 Processing example in *Static* mode 28
3.8 Processing example in *Active* mode 28
3.9 Macro to generate Processing's `setup` and `draw` mechanisms 28
3.10 Generating 2D shapes in AutoCAD using P2R 30
3.11 Generating circles in AutoCAD using P2R 30
3.12 Example of 3D modelling primitives in P2R 31
3.13 The `union` of a `sphere` with a `box` 31
3.14 The `intersection` of a `sphere` with a `box` 31
3.15 The `subtraction` of a `sphere` to a `box` 31
3.16 Generating a tours chain in `AutoCAD` using P2R 32
3.17 Implementing the `triangle` primitive in our runtime using `define-types` 32
3.18 The `foo-bar` module in Racket 34
3.19 The `checkFoo` function in Processing using `foo-bar` 34

4.1 Processing code to generate a 2D fractal tree 37
4.2 Fractal tree in different rendering environments 38
4.3 Processing code to generate a 3D tree 38
4.4 3D tree generated in AutoCAD and Rhinoceros 3D 39
4.5 Building a Pyramid Tower using P2R 39
4.6 Double helix generated from Processing code in AutoCAD 40
4.7 Generated Racket code for `helix` 41

4.8 Structure made from boolean operations applied to rotated cylinders 41
4.9 Perforated building generated in Firefox using Processing.js 42
4.10 Perforated building generated in AutoCAD using P2R 42
4.11 Similarities between DrRacket using Processing and the PDE 42
4.12 Error messages in DrRacket and the PDE 43
4.13 Definition tracking and REPL interactions with Processing code in DrRacket 43
4.14 Processing code to generate mosaics in P2R 44
4.15 Mosaic pattern generated in AutoCAD 45
4.16 Elliptic torus generated in AutoCAD and Rhinoceros 3D 45
4.17 Computing the value of `fib(40)` 47
4.18 Computing the value of `ackermann(3,7)` 47
4.19 Computing the value of `factorial(1000)` 48

List of Acronyms

API Application Program Interface.

AST Abstract Syntax Tree.

BIM Building Information Modelling.

CAD Computer-Aided Design.

DSL Domain Specific Language.

GD Generative Design.

GPL General-Purpose programming Language.

IDE Interactive Development Environment.

IR Intermediate Representation.

JVM Java Virtual Machine.

LOP Language Oriented Programming.

LW Language Workbench.

MPS Meta Programming System.

PDE Processing Development Environment.

REPL Read-Eval-Print-Loop.

VM Virtual Machine.

Chapter 1

Introduction

Technological evolution has given mankind the ability to create more robust and efficient solutions, changing the way problems are approached and solved. Many of these technological solutions have influenced areas, such as design and architecture, where they have explored different tools to aid their design creation process, including Computer-Aided Design (CAD) and Building Information Modelling (BIM) applications. With the help of powerful CAD environments, these communities have been able to develop increasingly complex and innovative designs in a more productive manner. However, in some cases, a manual design process in CAD is not sufficient. For instance, trivial geometry that is constantly repeated in a design could easily be generated using automated mechanisms. For this reason, CAD systems quickly found the need to provide Application Program Interfaces (APIs). For example, AutoCAD offers an API usable from C++, AutoLisp, and VBA; while Rhinoceros 3D supports RhinoScript, Python, and Grasshopper.

Programming offers many benefits for architects and designers, the most obvious is to generate repetitive geometry that otherwise would have to be done manually. This is an example explored by Generative Design (GD), an approach to design that is being used to help artists, designers, and architects, explore the benefits of programming. GD enables architects and designers to test different design approaches and combinations, providing them a low-cost solution to generate different prototypes. As a result, the benefits of programming in the day-to-day work of architects cannot be ignored, as it fosters their capacity for innovation and creativity. Due to this phenomenon, several programming languages and Interactive Development Environments (IDEs) have been adapted, or specifically developed, to fulfil the needs of these artistic communities. A well-known example is Processing, created to provide a pedagogical and highly visual approach to programming.

1.1 Processing

The Processing language was developed at MIT Media Labs and was heavily inspired by the *Design by Numbers* project [Maeda, 1999], 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

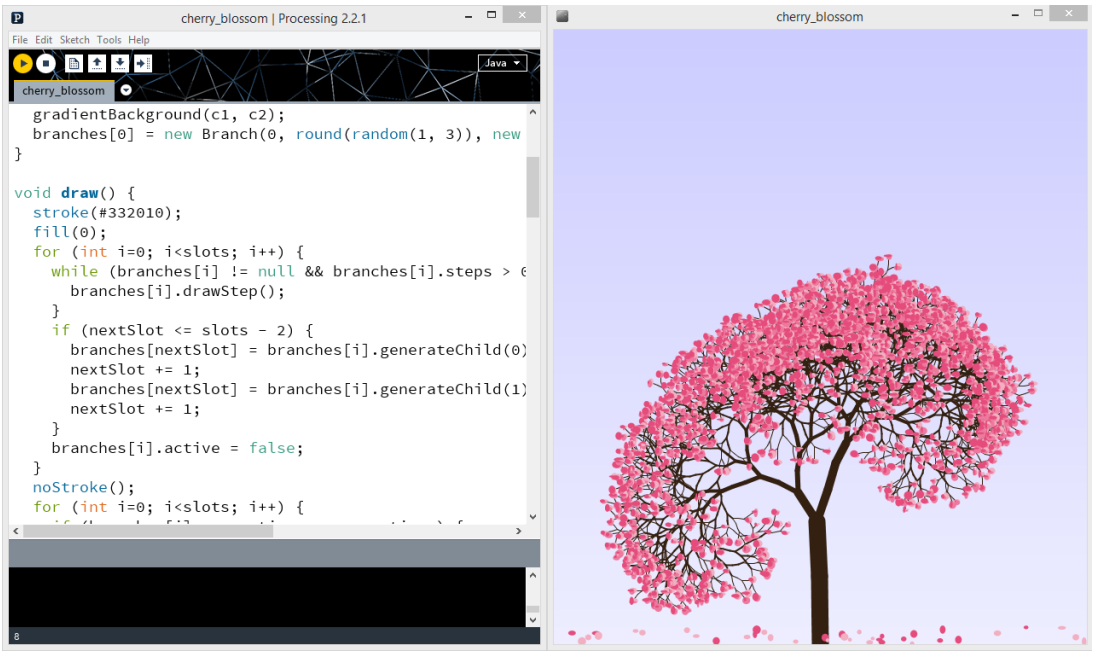


Figure 1.1: The Processing Development Environment

academic community, which has written several educational materials demonstrating how programming can be used in the visual arts. Processing is, arguably, the most successful effort to bring programming into the realm of design, art, and architecture. There are numerous examples of the use of Processing in digitally-generated paintings, tapestries, photographs, choreographies, visualizations, simulations, sculptures, music, games, etc.

Processing is based on the Java programming language, being statically typed and sharing Java's object-oriented capabilities. This design decision was due to Java being a mainstream programming language used by a large community of developers. Moreover, as Processing was developed to promote programming literacy in design and architecture, its syntax enables users to easily migrate to other languages that share Java's syntax, such as C, C++, C#, or JavaScript.

Notwithstanding, as Java is a fully featured multi-purpose language, it requires users to grasp a considerable amount of knowledge which can be irrelevant for users that want to develop simple scripts for visual purposes. As a result, several simplifying features were introduced in Processing that allow users to quickly test their design ideas without requiring extensive knowledge of the Java language. For instance, in order to execute code, Java requires users to define a `public class` and a `public static void main()` method. Processing simplifies this by removing these requirements, allowing users to write simple scripts (i.e. simple sequences of statements) that produce designs without the boilerplate code that is needed in Java.

The Processing language introduces the notion of a *sketch*, a common metaphor in the visual arts, acting as representation of a project that artists can use to organize their source code. Within a *sketch*, artists can develop their models in several Processing source files, yet they are viewed as a single compilation unit. Each *sketch* can operate in one of two distinct modes: *Static* or *Active*. *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 designs 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: `setup` and `draw`. 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 `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.

On top of being a programming language, Processing offers the Processing Development Environment (PDE). The PDE (illustrated in figure 1.1) is an IDE for the Processing language, where users can develop their scripts using a 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, colour pickers, sketch archivers, etc.

One of the main advantages of Processing is its ability to help designers quickly test and visualize their ideas, incrementally evolving them. This is possible due to Processing's rendering system, which is based on OpenGL, therefore allowing designers to rapidly render complex and computationally intensive designs that would take the typical CAD system much longer to produce. Furthermore, Processing offers users a set of modelling tools that are specially tailored for visual artists, namely 2D primitives, and a set of built-in classes specifically created for design. These primitives demonstrate the advantages of using Processing as a tool for testing and developing design prototypes.

Unfortunately, Processing fails to provide a connection with CAD or BIM applications. The Processing community is therefore limited to Processing's rendering system, that is based on OpenGL, lacking the CAD modelling features which are used by architects in their day-to-day work. This is an unfortunate situation, as the pedagogical capabilities of Processing become a trap for architects that, after learning Processing, face the unpleasant fact that they cannot use the language for architectural work, and, as a result, they are forced to learn a completely different programming language.

Our goal is to join both worlds, that is, to allow architects and designers to prototype new designs using Processing and generate the results in a CAD or BIM environment. Our solution, enables current Processing users to access a CAD application, allowing a wide community of artistic programmers to explore the modelling capabilities of a CAD environment. Moreover, architects that already use CAD in their day-to-day work, would have a new language programming available, exploring its wide range of examples and taking advantage of Processing's educational materials.

To achieve our goal, we will use Rosetta (discussed in section 1.2) to provide a connection with CAD tools. Using our work, architects will be able to program in Processing in the context of several CAD environments, and, at the same time, benefit from an additional set of modelling primitives designed for architectural work.

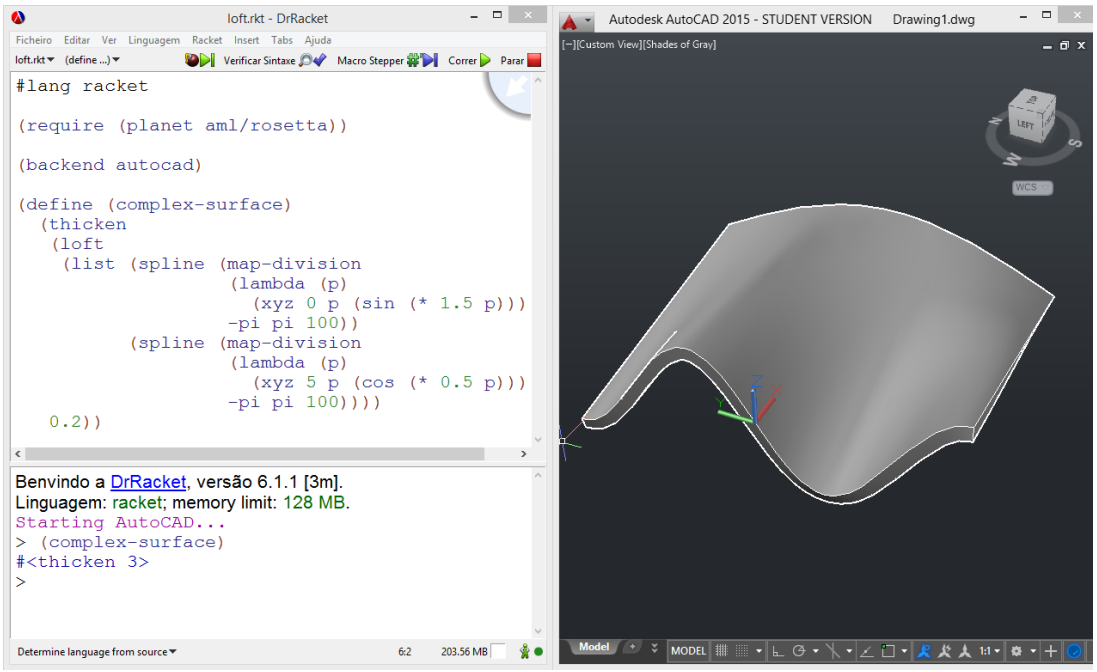


Figure 1.2: Rosetta using the Racket language as front-end and AutoCAD as back-end

1.2 Rosetta

Rosetta [Lopes and Leitão, 2011] presents a robust approach to GD, by providing an IDE and modelling primitives that are tailored for architectural work. Rosetta is implemented using the Racket language [Flatt and Findler, 2011b] and takes advantage of the pedagogic capabilities of DrRacket [Findler, 2013] — Racket's IDE.

Instead of limiting developers to use one programming language or CAD tool to create their designs, Rosetta (illustrated in figure 1.2) is composed of several programming language front-ends. The user can write scripts using one of the different supported languages, while generating identical models in any of its supported CAD back-ends. At the moment, Rosetta supports AutoLisp, JavaScript, Scheme, Racket, and Python as language front-ends, and AutoCAD, Rhinoceros 3D, SketchUp, and Revit, as CAD back-ends.

Each program written in one of the language front-ends is compiled to an intermediate Racket form, where a wide range of 3D modelling primitives are defined. Every primitive in Rosetta was adapted to work with each supported CAD tool. This way, new languages and CAD applications can be easily added to Rosetta, encouraging portability and interoperability. Moreover, as the modelling primitives have the same interface across all language front-ends, changing the used programming language is a trivial endeavour. Therefore, comparing to other environments used for GD, the main advantage of Rosetta is the emphasis on choice and portability.

Although Rosetta is already a powerful option for GD, it does not support Processing and, as a result, it is not usable to the large Processing community. Moreover, connecting Processing with a professional CAD or BIM environment would benefit architects that have learnt the language, enabling them to use Processing in the context of modern, script-based, architectural work. Therefore, the combination of

Processing with Rosetta offers a pedagogical programming language to Rosetta, and, at the same time, a full-featured CAD environment for Processing users to explore.

1.3 Goals

The fundamental goal is to make the Processing language available in a CAD environment, creating a solution that allows architects to prototype new designs using Processing and generating them in a CAD or BIM application. Therefore, to make Processing more useful for the designer and architect communities, we propose to extend Rosetta to support the Processing language. More specifically we have the following goals in mind:

- **Implement Processing in Racket** — the aim is to develop an implementation of Processing for the Racket language, respecting the language's semantics and implementing its type-checking mechanism, built-in operations, and standard library. As Processing is a complex and large language, our goal is to identify the most used aspects of the language that are explored by designers and architects, and then expand from this base line with the remaining features of the language.

- **Connect Processing with CAD applications** — the goal is to enable architects that have learnt Processing to prototype their designs in a CAD or BIM application, by exploring Rosetta's CAD abstractions and primitive operations.

- **Augment Processing's 3D modelling primitives** — as Processing's standard 3D modelling primitives are limited, our goal is to augment Processing with new 3D primitives that are heavily used in the context of CAD-based architectural work.

- **Integration with DrRacket** — Processing and Racket both offer an IDE to its users. On the one hand, Processing offers the PDE which provides features such as syntax highlighting, a tabbed editor, code formatting, etc. On the other, Racket has DrRacket, which also offers similar IDE features and complements them with additional ones, such as a Read-Eval-Print-Loop (REPL). As a result, our goal is to have IDE support for Processing in DrRacket, providing a similar development environment to Processing developers.

- **Interoperability with Racket** — as we are developing a new language on top of the Racket ecosystem, being able to take advantage of Racket libraries is an important feature to have. Therefore, our goal is to develop a interoperability mechanism that would allow us to access Racket libraries. This mechanism, for example, would enable us to use external modelling libraries built for Rosetta, and use them seamlessly with our Processing code. Also, using this mechanism, we can have access to other languages of the Racket ecosystem, such as Typed Racket [Tobin-Hochstadt et al., 2011].

To summarize, our goals are to: (1) implement Processing for Racket; (2) connect Processing with the most used CAD tools in architecture; (3) extend Processing with 3D modelling primitives adequate

for architectural work; (4) adapt DrRacket to work with Processing; and, finally, (5) have interoperability between Processing and the Racket language.

1.4 Outline

Chapter 2 explores different language implementations, focusing on compiler design, language workbench mechanisms, and on relevant implementations of the Processing language. Chapter 3 shows how our implementation was designed, describing the main design decisions and how the compilation process, runtime, and interoperability mechanism were developed and structured. Chapter 3 also describes how DrRacket was adapted to support the Processing language. Chapter 4 illustrates the main achievements and contributions of our implementation in relation to our proposed goals. Finally, Chapter 5 presents the main conclusions of our work, describing additional work to be explored in the future.

Chapter 2

Related Work

Creating a new language implementation is a complex endeavour that can be achieved using different approaches, namely compilation, interpretation, or language translation. Translating a high-level language such as Processing into another high-level language such as Racket is designated as a source-to-source compilation. As our goal is to implement Processing in Racket, different compilation approaches were studied, focusing on how source-to-source compilers are developed, analysing available language construction and translation methods, and exploring implementations related to our source-to-source compilation approach.

2.1 Compiler Design

A compiler's architecture is generally composed by a pipeline of phases which are usually divided into two distinct groups: the compilation front-end and back-end.

The front-end phases analyse the source code and produce an Intermediate Representation (IR) that will serve as an outline of the original code [Aho et al., 2006]. The front-end is implemented by a set of distinct steps, namely lexical, syntactic, and semantic analysis. Many techniques are employed to implement this phase of the compiler, either through custom tailored solutions built specifically for the language at hand, or through lexical analysers and parser generators that take as input a formal description and output programs that will be used to analyse the source code.

The back-end's responsibility is to use the IR to generate the target code. This phase is commonly designated as the synthesis phase of the compiler and is composed by an optimization and code generation steps. Generally, the main concern of these steps is to produce correct and efficient code, however some compilation efforts can have other requirements, as for example, generating human readable code.

The following sections present possible solutions to implement a compiler's front-end. To better understand the following section, some terminology is introduced:

- **Lexical Analyser** is a program that takes as input a sequence of characters and produces a list of tokens according to a set of predefined rules. A **token** is a string that has a meaning in the language. For instance, in a programming language a token can be an identifier, an operator, or a

keyword of the language. The process of dividing a stream of text into tokens is designated by **tokenization**. Generally, lexical analysers are implemented through the use of regular expressions.

- **Parser** is a program that consumes tokens and organizes them according to a formal specification (context-free grammar), producing a representation of the original source code. This grammar definition has several rules that define how tokens should be composed together, to produce a meaningful construct of the language. However, some elements of a language can lead to ambiguous semantics that the parser cannot handle, resulting in a poor grammar definition that generates parser conflicts. In order to solve this, **lookahead** tokens are used to help the parsing process. Depending on the specified number (**K**) of lookahead tokens, a parser can use the next occurring tokens to guide the construction of the parse tree. For instance, in the case of conflicts, when the parser has multiple rules with the same symbol to expand, lookahead tokens can be used to decide which rule to follow.

  A parser can follow one of these two types of parsing:

  Top-down parsing starts by expanding the root grammar symbol, and repeatedly expands each non-terminal rule until the full parse tree is produced. These parsers consume the input from left to right and follow a left-most derivation. Therefore, this method produces parsers that belong to the **LL(K)** family of parsers.

  Bottom-up parsing starts from the input tokens and tries to build increasingly complex structures until the root symbol is reached. These parsers are considered to be **shift-reduce** parsers. A **shift** operation occurs when the parser consumes another token from the input stream. A **reduce** operation occurs when a grammar rule is completed, aggregating all of the matched tokens into one non-terminal symbol that can then be composed, forming the parse tree. These parsers read the text left-to-right and follows a right-most derivation, thus creating the **LR(K)** family of parsers.

2.1.1 Recursive-Descent Parsing

Recursive-Descent parsing is a top-down method of syntax analysis in which a set of recursive procedures are used to process the source code [Aho et al., 2006]. It is a straightforward way of implementing hand-built parsers, where each non-terminal symbol is mapped to a recursive procedure.

Given an input stream, the parser checks whether it conforms to the syntax of the language. At each step, it maps the given token into a defined procedure, building a tree like structure of calls that directly resembles the syntax tree. Additionally, lookahead tokens can help the parser decide what grammar rule to expand, based on the next token in the input stream. These parsers can uses a custom number of lookahead tokens, producing LL(K) parsers.

A common problem when writing a parser is presenting good error messages to the developer when an error occurs; as these parsers are normally hand-built, the developer can easily create specific error messages. Also, due to their manual implementation, these parsers can easily be optimized in critical parts. As a result, these parsers are better tailored for the development of smaller languages due to their

simplicity. However, after some time, the implementation process becomes a tedious task, where the developer is constantly using boilerplate code that could easily be created automatically [Parr, 2010].

Another issue present in these parsers, is that they are unable to parse grammars that have left-recursion. A grammar is left-recursive if some non-terminal symbol will derive a form with itself as a left-symbol. Left-recursion can either be direct, when a rule immediately derives itself, or indirect, when a rule expands a non-terminal symbol that in turn will expand the same rule. To solve these problems, left-recursion can be removed from the grammar by re-writing it. However, as many programming languages employ left-recursive rules in their grammars, this is a costly task which has lead users to consider more robust alternatives, such as Lex/Yacc or ANTLR, that provide time-saving and more expressive way of analysing input and producing an IR.

2.1.2 Lex/Yacc

The combination of these historical tools is a famous partnership among compiler design, providing a two-phased approach to read the source program, discover its structure, and generate an IR.

Lex is a program generator designed for lexical processing of character input streams. Lex generates a deterministic finite automaton from user defined regular expressions and produces a lexer program, designated `yylex` [Lesk and Schmidt, 1975].

A Lex specification is composed by three distinct sections: definitions, rules and code section. In the first section, the user can define Lex macros that help in the definition of new rules. In the rules section, the user must define all of the forms/patterns that will be recognized by the `lexer` and map them into a specific action. The last section comprises C statements and functions that are copied verbatim to the generated source file. In this code section, the user can define routines that are used in the grammar definition to generate the IR.

Lex can be used as a standalone tool to perform lexical tasks, yet it is mostly used to split the input stream into lexical tokens. These tokens are then given as input to a parser (generally produced by Yacc), that will organize them according to its grammar, thus a combination of Lex and Yacc is generally used.

On the other hand, Yacc provides a general tool for imposing structure [Johnson, 1975]. Yacc takes as input a lexer routine (`yylex`) that produces tokens which are then organized according to grammar rules specified in the parser.

Resembling Lex, Yacc's structure is separated in three sections, where the definitions section includes the declaration of all tokens used in the grammar and some types of values that are used in the parser; the rules section is composed by a list of grammar rules with corresponding actions to be performed. Finally, in the same way as Lex, the code section provides a way of adding specific routines to the parser.

Internally, Yacc produces a LALR(1) (Lookahead LR parser) table driven parser, contrasting with the Recursive-descent parsers, given that they follow a bottom-up approach. The parser program generated by Yacc, is designated `yyparse`.

Despite of its expressiveness, Yacc may fail to produce a parser when specifications are ambiguous, thus representing errors in the grammar's design. It is difficult for Yacc to produce good error messages to the developer and, normally, debugging the grammar is a painful procedure.

A dreaded issue for Yacc users is the existence of shift/reduce and reduce/reduce conflicts. On the one hand, shift/reduce occurs when the parser cannot decide if another token should be consumed or if the rule should be immediately reduced. On the other, reduce/reduce conflicts occur when given two rules, both can be applied to the same sequence of tokens, generally indicating a serious error in the grammar's design.

Although some constructions are hard for Yacc to handle, the defence is that these constructions are frequently hard for humans to handle as well. Moreover, users have reported that the process of writing Yacc specifications has helped them detect errors at the beginning of the grammar's design [Johnson, 1975].

2.1.3 ANTLR

ANTLR is a powerful parser generator that can be used to read, process, execute, or translate structured text or binary files. It is widely used in academia and software industries to build all sorts of languages, tools, and frameworks [Parr, 2013].

ANTLR differs from the Lex/Yacc approach, in the sense that it integrates the specification of lexical and syntactical analysis, that is, both the grammar and the lexical definition are defined in a single file. It provides easy methods of creating Abstract Syntax Trees (ASTs) and generates a human-readable recursive-descent parser [Parr and Quong, 1995]. These top-down, recursive-descent parsers, belong to the LL(*) family. The key idea behind LL(*) parsers is to use regular expressions rather than a fixed constant or backtracking with a full parser to do lookahead.

As mentioned, the LL(K) family of grammars cannot handle left-recursive rules. The new version (ANTLR v4) automatically rewrites left-recursive rules into equivalent forms, yet the constraint is that the left-recursion must be direct, that is, when rules immediately reference themselves. In addition, ANTLR helps the language creation process by automatically creating representations of the parsed input, called parse trees, as well as parse-tree walkers that provide a quick way to traverse the parse tree.

ANTLR's specification was derived from Yacc's structure, in order to reduce the learning curve. It has a collection of rules that map into specific actions and a header zone where the developer can define required data types. However, the specification has evolved with the addition of new ANTLR features such as predicates, lexical analysis specification, and error reporting.

A point in favour of ANTLR is that it provides a graphical Interactive Development Environment (IDE) tool, called ANTLRworks. It helps the developer create new rules by informing of possible conflicts and how to resolve them. In addition, it can automatically refactor and resolve conflicts in the grammar such as left-recursion. After parsing a file, ANTLRworks builds parse trees for the developer, proving visual representations of the parse tree. This aids developers in debugging the grammar, thus saving many hours of work. Therefore, ANTLRworks is a good tool for developers that do not want to create their

parsers manually but want to quickly prototype their newly designed languages.

2.1.4 Additional Systems

Other relevant alternatives that are used to implement the compiler front-end are summarized and presented in this section. Systems such as Bison [Donnelly and Stallman, 1993], Coco/R [Moessenboeck, 1990], or JavaCC [Viswanadha et al., 2009] are possible alternatives to implement parser generators.

Bison is a popular choice mainly due to its reuse of Yacc syntax, thus supporting any Yacc compatible grammar. Bison generates a LALR(1), but also provides more powerful generalized LR parser (GLR). The main difference of these parsers is that in the event of a conflict (shift/reduce or reduce/reduce), the parser expands all possibilities therefore generating a set of different parsers for each case. Like Yacc, Bison uses an external lexical-analyser to fulfil its lexical analysis needs.

Coco/R produces LL(1) recursive-descent parsers, instead of producing a table parser. It integrates a scanner system that already provides support for some common lexical concerns such as nested comments and `pragmas` (instructions that occur multiple times in the source code), that normally require additional effort to implement.

JavaCC belongs to the LL(1) family of parsers but at certain points of the grammar can use additional look-ahead tokens to resolve some ambiguities. It integrates a lexical-analyser that is very similar to Lex. Additionally, it provides frameworks to produce documentation (JJDoc) and generate tree building processing (JJTree).

Table 2.1 provides a comparison of the analysed parser generators according to the parsing algorithm that they follow, the lexical-analyser that is used, and what grammar notation they use. In the case of recursive-descent, as they are hand-crafted, the lexer and the grammar representation are defined according to the developer's preferences.

	Parser Type	Lexer	Grammar Notation
Recursive-descent	LL(K)	N/A	N/A
Yacc	LALR(1)	External	Yacc
ANTLR	LL(*)	Internal	E-BNF[1]
Coco/R	LL(1)	Internal	E-BNF
Bison	LALR(1), GLR	External	Yacc
JavaCC	LL(K)	Internal	E-BNF

Table 2.1: Comparison of the different parsing alternatives

2.2 Language Workbench

Traditionally, programming languages are developed by creating a standalone compiler. Afterwards, IDEs are created to ease the development process, requiring additional compiler development.

[1] Extended Backus-Naur Form

Language Workbench (LW), a term introduced by Martin Fowler [Fowler, 2005], attempt to simplify this process by providing an integrated environment for building and composing new languages, and by creating editor services automatically.

LW enables the use of Language Oriented Programming (LOP) [Ward, 1994]. Opposed to general programming styles (imperative, functional, object-oriented, etc.), LOP focuses on the creation of multiple languages for each specific problem domain. An example of LOP is Lex and Yacc, where the grammar acts as a Domain Specific Language (DSL) to define the syntax/structure of the source code. Another good example is in Unix systems, where several specific purpose languages that where created to fulfil specific needs (make, awk, bash, etc).

The goal is to use LWs to easily implement General-Purpose programming Languages (GPLs) and DSLs that are equipped with all the features of a modern IDE. One of the issues with available programming languages is that they are not expressive enough, in the sense that, when reasoning about a problem, the developer creates a mental representation of the solution that can easily be explain to his peers, yet in order to implement it, certain language specific constructs are introduced cluttering the developer's mental model. Therefore, LWs provides a high-level of abstraction to developers that attempts to closely resemble their initial mental representation.

A LW must have some form of the following features [Erdweg et al., 2013]:

- **Syntax definition** defines how programs or models are presented to the users, either through text, graphical, or tabular representation.
- **Semantic definition** defines how programs or modules are analysed and transformed into other representations. These semantic definitions state how the language can be translated to another (translational semantics) or how to directly analyse and interpret the program or model (interpretation semantics). Translational semantics can be either from model-to-text or model-to-model. Additionally, some LW can have a concrete syntax to implement these language translations.
- **Editor services** can provide different editing modes (free-form or projectional [Fowler, 2008]). The free-form manipulates the actual source. On the other hand, the projectional mode creates representations of the source, that can modified and composed without changing the original code.

  Editor services can be based on two types: the syntactic and semantic editor services. For example, syntactic services are syntax highlighting, source outline, folding, and auto formatting. Semantic services are reference resolution, code assist, quick fixes, and error markers.
- **Language composability** addresses the evolution of the language through extension (languages are extended with new constructs for specific needs) or composition (languages of different domains can be joined or be embedded into each other).

Furthermore, some LWs may provide additional features for created languages, such as language specific validations (e.g. structure, names, and types), testing and debugging support.

LW are growing in number and diversity, due to academic research and the industry's rising needs. Existing workbenches have chosen separate approaches to the problem, implementing different features

and using distinct terminology, so it is hard for users and developers to understand the underlying principles and design alternatives. In essence, LW offer the developer flexibility in creating new languages (GPLs or DSLs), as well as an efficient way to create intelligent tools to manipulate them.

The following sections present relevant LW systems that can be used for LOP.

2.2.1 MPS

Meta Programming System (MPS) is a LW created by JetBrains, greatly inspired by Charles Simonyi's Intentional Software [Simonyi et al., 2006]. MPS enables users to design DSLs and start using them straight away. It offers a simple way of working with DSLs, where even domain experts that are not familiar with programming can easily use them to solve their specific domain problems.

In MPS, the developer works with two levels of programming: the meta level, that describes what are the properties of each node of the program and what type of relationships are available for their instances [Dmitriev, 2004], and the program level, that provides normal programming semantics.

MPS offers a `Base Language` and a set of `Language Definition` languages. The `Base Language` offers simple features that any programming language supports, such as arithmetic, conditional, functions, loops, etc. It is useful to developers because it provides them the core constructs of any GPL, to be extended to fit new DSLs specific needs. This is particularly relevant because DSLs usually require a small sub-set of constructs of a GPL. MPS also offers some built-in extension languages to the `Base Language`, namely the `Collection Language` or the `Regular Expression Language` that provides support of collections and regular expressions.

The `Structure`, `Editor`, `Type System` and `Transformation` are some examples of languages that belong to the set of `Language Definition` languages. For example, the `Structure Language` allows the developer to define the language's structure that is being created. This is particularly useful because MPS does not use a grammar. Another example is the `Editor` Language that lets the developer define the editor layout for the language, providing a modern IDE.

During the language creation process, the developer can define a set of rules and constraints to the language, allowing MPS to constantly verify produced code, making development with the language an easier and less error-prone experience. Moreover, it provides support for building debuggers and integration with version control systems for newly created languages.

The following paragraphs describe MPS according to the core concepts of LWs:

Syntax Definition MPS is a projectional editor, thus language definitions do not involve a grammar. Instead, these languages are implemented through concepts. Subsequently, production rules (editors) render these concepts through a specific notation. MPS can use a textual or tabular notation to define their DSLs. MPS also supports mathematical symbols, such as integrals or fractions.

Semantic Definition In MPS, the developer can define transformations between languages because they are based on the language's structure, thus this translation process is accomplished by mapping

structures from the source language to the target language. Therefore, translations can be from model-to-model or model-to-text, enabling the use of DSLs with other DSLs or GPLs.

Editor Services The use of a projectional editor, enables the generation of projections for each syntax node, allowing their composition. Contrary to parser-based environments, where developers write text into a file, projectional environment require a projection engine to edit their programs. Therefore, as projections of the source are being manipulated, IDE services must be provided to the developer in order to modify the source. Some relevant editor features that the language will benefit from include auto-complete, refactorings, syntax highlighting, and error highlighting.

Language Composition Languages in MPS are well-known concepts that can be interlinked. Using the object-oriented paradigm as a metaphor, concepts represent a class and models are like objects. So object-oriented programming practices can be applied in the language composition. Since this composition is performed in a meta-level, process of joining languages syntactically is a trivial one.

Languages can inherit from other languages, so the existence of the `Base` Language greatly helps the developer in creating languages by extending the *Base* Language with specific concepts, hence creating a DSL that has features that are present in most GPLs.

2.2.2 Spoofax

Spoofax [Kats and Visser, 2010] is a LW that focuses on the creation of textual DSLs with the full support of the Eclipse IDE. The Spoofax environment provides a set of tools that are required for language creation such as a parser generator, meta-programing tools and an IDE that aids the developer in the creation of new languages. Since Spoofax is deeply connected with Eclipse's environment, it can take advantage of its plug-in system, offering a large choice of plug-ins that can help the developer in many development tasks such as version control or issue tracking. Spoofax has been used to developed a number of experimental languages and has been taught in model-driven software engineering courses. Moreover, as Spoofax is an integrated environment for the specification of languages, it generates editors that can be dynamically loaded into Eclipse. This provides an agile way of incrementally developing new languages, by showing an editor of the language as the definition is being created. Therefore, applied modifications to the language can be quickly observed, tested, and changed according to desired language features.

Spoofax employs a pipeline of steps to perform analysis and transformations to the source code, where it is parsed, transformed, and generated as target language code. Naturally, the syntactic analysis is done by a parser that generates a normalized representation of the source code. Between these two steps, de-sugaring is performed to the syntax tree, as well as a step that decorates the tree with semantic information to be used by the editor services. Afterwards, a normalized representation is produced to generate the target code.

The following paragraphs describe Spoofax according to the core concepts of LWs:

Syntax Definition Spoofax only supports textual notations for its created languages. Syntactic definition is accomplished through the use of SDF (Syntax Definition Formalism). SDF employs highly modular, declarative grammars that combine lexical and context-free syntax into one formalism. Spoofax defines two types of syntax in its grammar, the concrete (keywords and so on) and the abstract syntax (used for analysis and transformation of programs).

The grammar is a core dependency for the implementation of all other services offered by the LW. Through the grammar, editor services can be derived, as well as parsers, that analyse the source code and produce abstract representations of it.

Semantic Definition Spoofax uses Stratego to specify and describe the semantic definitions of a language. Stratego is a DSL and a set of tools for developing standalone software transformation systems based on formal language descriptions. It is based on rewrite rules and strategies that control these rules. Rules may employ transformations to abstract syntax or may use concrete syntax to do so. Additionally, code generation rules can be uses to transform the syntax to a compilable form.

Editor Services As Spoofax is based on the Eclipse IDE, it supports features that Eclipse offers like outline view of the source, syntax highlighting, code folding, syntax completion, bracket management, error/warning marking, etc. These editor services are automatically derived from the syntax definition.

Language Composition Spoofax supports composition due to the modularity of SDF and Stratego. SDF grammars generate scannerless generalized LR (SGLR) parsers that can be composed to support language embedding and extension. The composition of semantics is assured by Stratego, using its rules and strategies. In addition, due to the use of de-sugaring, semantic decorating of the syntax tree, and the translation to a normalized representation, languages can be easily inter-winded.

2.2.3 Racket

Racket [Flatt and Findler, 2011b] is a descendant of the Scheme language, thus belonging to the LISP family of languages. Racket has been a very popular choice to teach programming to beginners due to its simple syntax and its multi-paradigm approach to programming. In spite of being a language that is used for beginners, it is also used as a research environment for construction and study of new languages and programming paradigms.

Racket provides DrRacket (previously DrScheme, presented in figure 2.1) a pedagogic IDE, that offers syntax highlighting, syntax checking, profiling and debugging. It is considered to be easy to use and an appealing development environment for novice programmers. DrRacket also supports a student-friendly interface for teaching purposes, providing a multiple language level system so that students can incrementally learn a language. In recent versions, a new packaging system was developed, allowing developers a better way to share their custom libraries and tools with the Racket community.

Racket encourages developers to extend the base language when needed. To this end, it provides mechanisms to create language extensions that can be packed into modules to reuse in different pro-

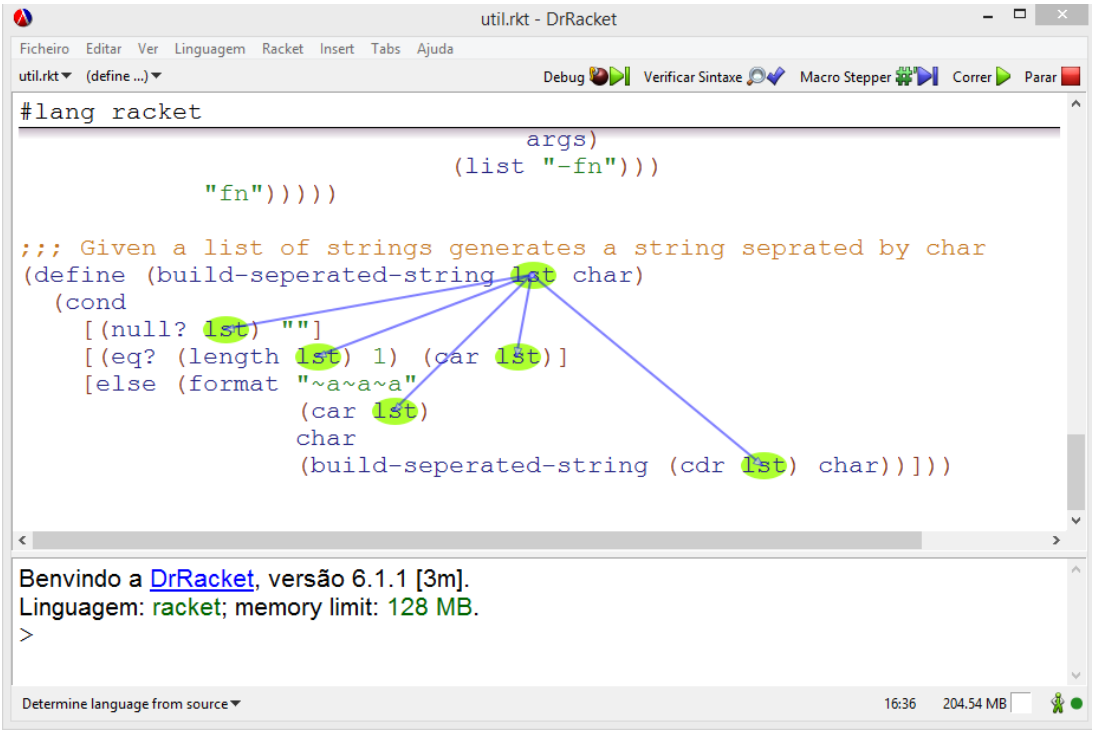


Figure 2.1: DrRacket: Racket's pedagogic IDE

grams. So in addition to being a programming language, we can view Racket as a framework for creating and extending new languages [Flatt, 2012].

Racket supports language extension through the use of macros, which are functions that transform syntax into different syntax at compile-type. These macros use Racket's Syntax-objects to represent the syntax that they are manipulating. Syntax-objects [Flatt and Findler, 2013] are ASTs that contain syntactic data as well as meta-data such as source location information.

The following paragraphs describe Racket according to the core concepts of LWs:

Syntax Definition As Racket is a textual GPL, it is natural that the supported language notation is a textual one. The Racket base syntax is parenthesized syntax, yet it supports the creation of new syntax for its DSLs by creating lexers and parsers.

Semantic Definition The translation of the language is accomplished through the use of Racket's macros. Macros will use the generate IR to generate semantically equivalent Racket code, that can then be composed.

Editor Services The use of syntax-objects and source preserving tokens, enables DrRacket's tools to take advantage of this information to easily create editor services such as syntax highlighter, syntax checker, a debugger and a profiler. However it is not guaranteed that these editor services will work out-of-the-box, thus requiring some manual intervention by the developer.

Language Composition Racket allows the combination of modules that are written in different languages. This module system enables developers to make a fine-grained selection of the tools that are required for their needs, providing a simple way of embedding and extending different language definitions, as ultimately the source code is translated to Racket. Therefore languages can be composed and embedded into each other by selecting and importing different modules.

2.2.4 Alternative Language Workbenches

In addition to the presented alternatives, other popular systems exist such as SugarJ [Erdweg et al., 2011], Rascal [van der Storm, 2011] and Xtext [Efftinge and Völter, 2006]. These LW are all textual-based and are integrated with Eclipse IDE. This enables them to take advantages of all the modern editor services that Eclipse provides, namely syntax highlighting, outlining, folding, reference resolution, semantic completion, etc.

SugarJ and Rascal, both employ SDF for syntactic definitions, thus they support generalized parsing that enables syntactic composition. Xtext uses ANTLR to implement its parsing needs, thus grammar composability is limited. SugarJ and Rascal take advantage of rewrite rules to provide semantic definition, allowing model-to-model and model-to-text translations. This is accomplished through the use of an intermediate form (ASTs) that can be used for the composition of multiple DSLs. Both systems provide a concrete syntax to support this translation. SugarJ attempts to provide language extension through the use of sugar libraries, that have syntactic sugar and de-sugaring rules which enable the creation of new extensions and that specify how the generation of the target code is made. Xtext separates the parsing from the linking of modules. It attempts to achieve composability by connecting the meta-model of the languages together and then generating the target code from this meta-model.

2.3 Source-to-Source Compilers

Source-to-Source compilers (or transcompilers) are a type of compiler that translates a high-level language into another high-level language. These allow developers to use and access frameworks and libraries from other languages, providing more interoperability between them. In addition, they can be used to port legacy code into a more recent language or perform source optimizations on the existing code. Another advantage to be considered, is that as high-level language code is being generated, the code can be understood and debugged by the developer.

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. We also analysed ProfessorJ due to the similarities that Java shares with Processing, and that Scheme shares with Racket.

2.3.1 Processing.js

Processing.js [Resig et al., 2008] 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 such as Firefox, Safari, or Chrome, using the `<canvas>` element. 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. Aside from the parser, it enables de embedding of other web technologies into Processing sketches.

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 [Bader-Natal, 2011] 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.

2.3.2 Processing.py & Ruby-Processing

Ruby-Processing [Ashkenas, 2015] and Processing.py [Feinberg et al., 2014] produce Processing/Java as target code. Both Ruby and Python have language implementations for the Java Virtual Machine (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.

2.3.3 ProfessorJ

ProfessorJ [Gray and Flatt, 2003] was developed to be a language extension for Scheme's pedagogical development environment, DrScheme [Findler et al., 1997]. The aim was to provide a smoother learning curve for students that needed to learn Java. ProfessorJ includes three language levels: the beginner, the intermediate and the advanced level, that progressively cover more complex structures of the Java language. The beginner level, provides an introduction to the Java syntax. The intermediate level introduces new concepts such as object-oriented programming and polymorphism. Finally, the advanced level, enables the use of loops, arrays and other more advanced language concepts.

ProfessorJ was developed as a source-to-source compiler that translates Java to Scheme. First, it starts by parsing the Java code using a Lex/Yacc approach available in Scheme. This produces

an Scheme IR where source tokens are converted into location-preserving identifiers. Afterwards, the Scheme code is processed by the Scheme compiler [Gray and Flatt, 2004]. Therefore the approach was to implement the parts of Java that easily mapped into Scheme through the use of macros. Tasks that could not be solved by the use of macros, were implemented directly in the compiler. The use of source preserving tokens throughout the compilation process and the mapping of Java constructs directly into Scheme's constructs, provided a nearly out-of-the-box usage of DrScheme's tools.

Another main concern that ProfessorJ tried to address is the ability to use Java frameworks and libraries in Scheme. An example of this is ANTLR, that is not available in Scheme and could bring new parsing solutions to the Scheme world.

However some issues needed to be solved. For instance, due to Java having many namespaces and Scheme having a single namespace, required the use of name-mangling techniques to avoid name collision of methods. Another issue to take into account, is that, in Scheme, numbers do not have limited range and will automatically become bignums, thus there could semantic mismatch of Scheme's primitive types with Java's types resulting in semantically incorrect code.

On the other hand, statement expressions such as return, break and continue, are implemented by the use of continuations. This is an expensive operation in Scheme, thus the usage of continuations to implement these statements is a performance bottleneck.

Performance wise, ProfessorJ performs poorly in comparison with Java and equivalent programs written directly in Scheme. Source for this degradation of performance is manly due for some non-optimal solutions that where considered in the development of the compiler, namely continuations, the implementation of primitive types, and to the fact that the JVM performs smart optimizations to Java code.

2.4 Comparison

The previously presented systems were analysed to understand what are the main design decisions, concerns, and similarities that they share among them. Table 2.2 provides an overview of their main features.

	Target Environment	IDE	Runtime
Processing.js	JavaScript	PDE / SketchPad	JavaScript
Ruby-Processing	JRuby	—	Java
Processing.py	Jython	PDE	Java
ProfessorJ	Scheme	DrScheme	Scheme

Table 2.2: Comparison of different implementations of source-to-source compilers

Observing all previous implementation, we arrive to the conclusion that an IDE is an important feature to have in a new implementation of the Processing language (as only Ruby-Processing is lacking one). Furthermore, Processing.js and ProfessorJ implement the runtime in the target language to achieve

greater interoperability. On the other hand, Ruby-Processing and Processing.py take advantage of JVM language implementations to provide Processing's runtime.

Analysing the previously presented systems, we observe that none offers a solution that allow us to explore Processing in the context of a Computer-Aided Design (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 the CADs applications. For instance, OBJExport [Louis-Rosenberg, 2013] 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.

Notwithstanding, after a careful analysis of different ways of implementing languages and Processing implementations, we conclude that the Racket LW has sufficient tools to implement the Processing language. Firstly, because it allows for 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 LW capabilities enable us to easily adapt our Processing implementation to work with DrRacket (Racket's educational IDE), providing an IDE to users, a fundamental feature to have in any Processing implementation. Moreover, after analysing ProfessorJ (presented in section 2.3.3), we concluded that due to the similarities between Java's and Processing's language definitions, parts of the lexical and syntactical definitions, and type-checking procedures could be adapted and reused in our implementation. The next chapter explains how these mechanisms were achieve, detailing the main design decisions of our implementation.

Chapter 3

Implementing Processing in Racket

Our goal was to develop a solution that would enable Processing to generate models in a Computer-Aided Design (CAD) application. Therefore, we analysed different solutions that would allows us to achieve our goals. The most prominent one was Rosetta, as it allows us to access several different CAD applications, and, at the same time, it provides several modelling primitives, specifically tailored for architectural work, that we could use to augment Processing's modelling capabilities. Moreover, the set of different Processing implementations reveal that Processing has a strong potential to be combined with other languages, a feature that Rosetta (due to Racket's language combination mechanisms) also addresses.

Therefore, we concluded that it would be better to build our solution on top of the Racket platform. Furthermore, as Racket is an ecosystem for the creation of new languages, our chosen solution was to develop a source-to-source compiler that translates Processing to Racket. This way we are able to use Rosetta to connect Processing to the several CAD back-ends that Rosetta supports, and augment Processing with a wide range of 3D modelling primitives. Our solution also consists of a runtime for the Processing language written in Racket, and of an interoperability mechanism to access Racket libraries from Processing code. Finally, to provide a similar development environment, we adapted the DrRacket Interactive Development Environment (IDE) to support the Processing language.

The following sections explain in detail how our implementation was structured and developed. Section 3.1 details the main modules of our compiler and how they relate to each other. Section 3.2 describes the whole compilation process and how each phase is implemented. Section 3.3 shows how the runtime was developed and how Rosetta's modelling primitives were used. Section 3.4 shows how the interoperability mechanism with Racket was implemented. Lastly, section 3.5 describes how DrRacket was adapted to support the Processing language.

3.1 Module Decomposition

A Racket source file usually starts with the line `#lang <language>` to specify which language of the Racket ecosystem is used. Naturally, a file using the Processing language is specified using the

`#lang processing` declaration. To implement Processing in Racket, two modules are required: a *Reader* module, defining how Processing's code is translated to Racket code, and a *Language* module, which provides the runtime environment for our implementation. Figure 3.1 illustrates how the main modules were implemented, as well the dependencies between them. Moreover, as Racket creates a distinction between the compiler and the runtime (i.e. the *reader* and *language* modules), two major modules were created: the *Compiler* and *Language* modules. The following paragraphs provide a detailed description of the most important modules in our compiler.

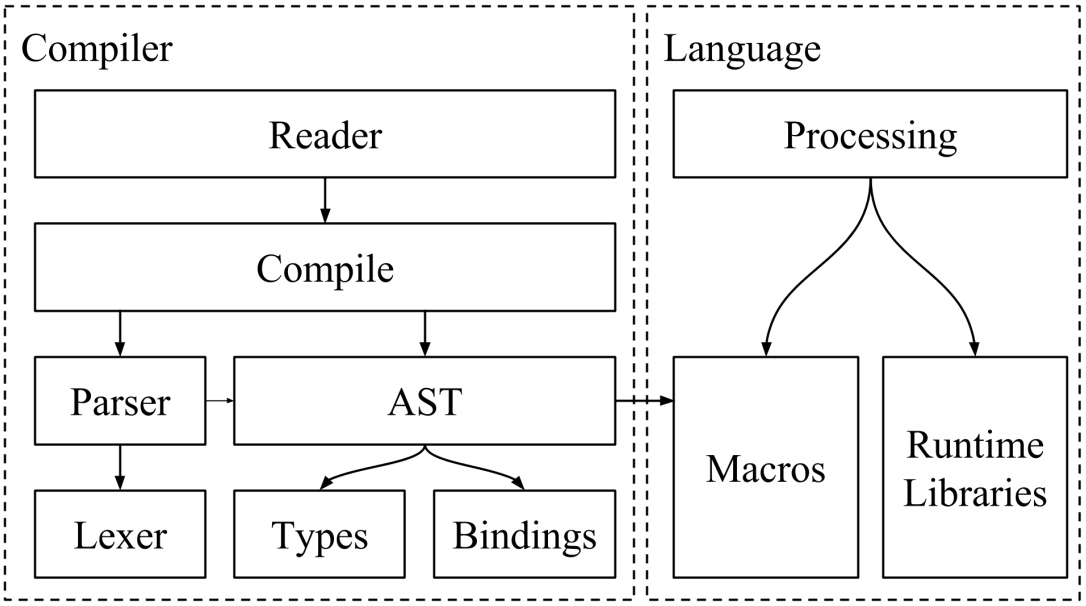


Figure 3.1: Main module decomposition and dependencies: The arrows indicate a uses relationship between modules - module A uses ($\longrightarrow$) module B

3.1.1 Compiler Modules

Reader Module To add Processing as a new language module [Flatt and Findler, 2011c], a new specifically tailored *Reader* is needed for Processing. This enables Racket to parse Processing source code and transform it to semantically equivalent Racket code. The *Reader* must provide two important functions: `read` and `read-syntax`, receiving an `input-port` as input. The former produces a list of s-expressions, while the latter generates `syntax-objects` [Flatt and Findler, 2013]. i.e. s-expressions with lexical-context and source-location information. The *Reader* uses functions provided from the *Compile* module to create and analyse the Intermediate Representation (IR) 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 Abstract Syntax Tree (AST), check types, generate Racket code, and load the runtime bindings.

Parser & Lexer Modules The *Parse* and *Lexer* modules contain all the functions that analyse the lexical and syntactic structure of Processing code. Racket's `parser-tools` library [Owens, 2011] was used to implement the lexer and the parser, adapting parts of ProfessorJ's (discussed in section 2.3.3) lexical and grammatical definitions 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`s, 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` implements methods to: implement and manage custom scope rules; type-check the code and promote types; and generate semantically equivalent Racket code.

Types and Bindings Module The *Bindings* module provides auxiliary data structures needed to store and manage different Processing bindings. A custom `binding` class was created to abstract binding information, for example, a function binding will contain the modifiers, argument and return types. Additionally, a custom `scope` class was created to store all of these bindings and to handle Processing's scoping rules. Each `scope` has a reference to its parent `scope` and has a hash table that associates identifiers to a `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 some of ProfessorJ's (presented in section 2.3.3) type-checking functions to work with our compiler.

3.1.2 Language Module

The *Language* module encompasses all the modules that implement the macros, functions, and data types required by our generated Racket code, and that form Processing's runtime environment. These functions are provided by the *Processing* module, using the *Macros* and *Runtime Libraries* modules. The former contains necessary source transformations required to generate equivalent Racket code, while the latter, provides an interface that implements Processing's built-in classes and modelling primitives, using Rosetta to generate designs in several CAD applications.

3.2 Compilation Process

The implementation of the compiler for Processing follows the traditional pipeline approach (illustrated in figure 3.2), composed by three separated phases, namely parsing, code analysis, and code generation.

As illustrated, the compilation process starts by transforming Processing source code into an IR (in the form of an AST). Subsequently, the AST will analysed and type-checked, finally generating semantically equivalent Racket code. This code is then loaded into Racket's Virtual Machine (VM), where it is

executed and benefits from Racket's optimizations and just-in-time compiler. Each compilation phase will be described in greater detailed in the subsequent sections.

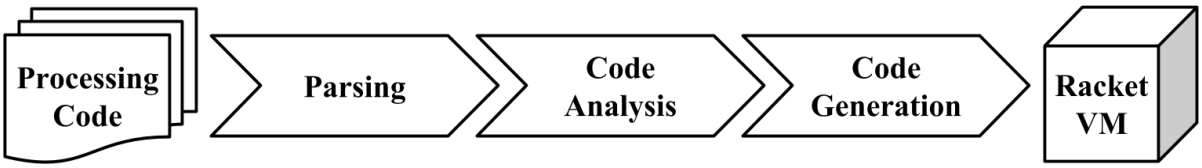


Figure 3.2: Overview of the compilation pipeline

3.2.1 Parsing Phase

The compilation process starts by the *Parsing* phase, which is divided in two main steps: the lexical analysis and the syntactical analysis. First, Processing source code is read from an `input-port`, using the scanner produced by our lexical specification. During this step, the input is transformed into `position-tokens` that contain their original source location, since DrRacket's IDE services require the original source locations to work correctly.

Secondly, these tokens are fed to a LALR parser which checks if the code is syntactically correct, finally producing an AST. Each node of the AST, is stored as Racket object that derives from a general AST superclass, saving relevant information needed throughout the compilation process, such as its source location or its current scope. Moreover, each AST subclass stores specific node-related information, needed for each different node's needs. For example, an `if` node will have references to the condition, consequent, and alternative nodes.

3.2.2 Code Analysis Phase

After the *Parsing* phase, a series of checks must be made to the generated AST. This is due to differences in Processing's and Racket's language definitions. Processing is a imperative, statement-based, and statically typed language, while Racket is functional and dynamically typed. As a result, custom tailored mechanisms were needed to translate Processing code to semantically equivalent Racket.

Firstly, we had to address the scope incompatibilities between Racket and Processing. On the one hand, Racket offers lexical scoping, thus if a variable is not found in a given scope, the enclosing scopes are searched until the outermost scope is reached. Also, when creating a new scope, new definitions introduced into that scope may shadow previously created definitions. For instance, the following form introduces a new scope `(let ((x 1)) ...)`, binding `1` to `x`. Any new lexical scope created inside the `let`'s body and that creates a new binding for `x`, will shadow the former definition.

On the other hand, Processing has a more limited approach, as it forbids redefinition of variables in inner local scopes. Analogously, a variable definition of `x` in an inner scope will not shadow previous definitions, but instead will generate an duplicate variable error. This is due to Processing's scoping

rules, which separates scopes into global and local, where each local scope inherits all the bindings of the enclosing scope.

Due to these restrictions, we had to develop a way knowing if a variable definition was valid in the scope or not, therefore we created our custom mechanism to implement Processing's scoping rules. So when a new definition is created, be it a function, variable, etc., the newly defined binding is added to the node's current scope along with its type information, making all the required checks to see if the definition is semantically correct with Processing. To compute bindings introduced by our code, we traverse the whole AST passing the current scope to each child. For nodes that create a new block (e.g. function definitions, conditional statements, loops, etc), we have to create a new local scope that holds a reference to its enclosing scope.

After checking that bindings introduced in the AST adhere to Processing's scoping rules, we have to perform an further analysis to ensure that our code's types are used correctly. As stated, Processing shares many of Java's typing rules, so parts of ProfessorJ's type-checking primitives were adapted in our Processing implementation. Similar to the bindings step, the type-checking procedure runs over the AST, repeatedly calling it over its child nodes until the AST is fully traversed. This procedure uses the binding information previously saved by the custom scoping mechanism to discover the types of each binding. During the type-checking, each node is checked for type correctness, and, in some cases, its types have to be promoted. In the event that types do not match, a type error is produced, signalling where the error occurred. For instance, consider the following function call `foo("number", 20)`. To correctly type-check this expression, first we would have to find out its return type by looking for the `foo` function binding in our custom scope. Afterwards, we would have to check if values passed as arguments are compatible with the types that were saved in the binding.

3.2.3 Code Generation Phase

After the AST is fully analysed and type-checked, we can generate semantically equivalent Racket code. This is achieved by calling the code generation procedure over the AST, producing an intermediate Racket representation that contains custom defined macros and functions to implement Processing's semantics. This intermediate Racket form is wrapped inside a `syntax-object` [Flatt and Findler, 2013], containing the quoted form of the code (an s-expression), the source location information (line number, column number and span) and lexical-binding information. Subsequently, the `syntax-objects` will be consumed by `read-syntax` at the reader level. Afterwards, Racket will expand our macros and load the generated code into Racket's VM and just-in-time compiler.

The rest of this section will describe the decisions regarding code generation for the most relevant nodes.

Literals

Processing has literal notations for integer, floating-point, character and String literals. These are mapped to Racket's numbers, characters, and strings. Generating code for each literal requires an

analysis of the defined type (i.e. the type associated to the literal in the original Processing code) and of the assigned type (i.e. the type that the type-checker assigns to the node). If the types are the same, the stored value is immediately returned, otherwise, the literal is promoted to the type assigned by the type-checker.

Identifiers & Multiple Namespaces

Processing identifiers are directly compiled to Racket identifiers, but with a caveat. Processing, unlike Racket, has multiple namespaces for variables, functions, and classes; while Racket has a single namespace. Therefore, to support multiple namespaces in Racket, identifiers were mangled with custom tags to avoid name collision. For example, variable `float foo = 10;` would be internally translated into `foo`, without any change. On the other hand, function definition `void foo() {...}`, due to name collisions, requires a distinguishing tag. As a result, we append `fn` to each function definition, therefore function `foo` is translated to `foo-fn`. The use of **'-'** as a separator allows us to solve the problem of name clashing with user defined Processing bindings, as **'-'** is not allowed in Processing names.

Furthermore, functions in Processing can be overloaded, so additional to the `fn` tag, specific tags are used to be able to distinguish each overloaded function. This was accomplished by using the argument's types to provide distinction between the function's names. As a result, the function definition `float foo (float x, float y){ ... }` would be translated to `foo-FF-fn`. However, if the function's arguments are not primitive types, the full name of the type is used as a tag instead.

Function Definitions

Function definitions in Processing are global and must be made visible to other Processing compilation units. Also, as Processing uses positional arguments and as function identifiers use the name mangling technique previously mentioned (thus supporting overloading), function definitions were implemented using Racket's `define`. The macro shown in figure 3.3 implements Processing's function semantics.

```
(define-syntax-rule
    (p-function (id arg ...) body)
    (begin
        (provide id)
        (define (id arg ...) body))))
```

Figure 3.3: Macro to generate Processing's functions

Furthermore, a support for `return` statements had to be developed. The `return` statement immediately returns to the point where the function was called, returning a given value. However, Racket's semantics differ from Processing's in this aspect, since every Racket form is an expression (i.e. produces a value) the return value of a function call in Racket is the value of the last expression of the function. To implement `return` statements in Processing, we used Racket's escape continuations [Flatt and Findler, 2011a], by capturing the function's body. Notwithstanding, continuations are expensive operations that

can be avoided for tail returns. Firstly, because they will be the last expression of the control flow, and secondly, because Racket returns the last expression of the function.

Control Flow Statements

As Racket and Processing have the same evaluation order of their arguments (left to right), Processing's statements can be directly mapped into Racket forms. For instance, Processing's `if` form can be translate to a Racket `if` form.

```
(define-syntax p-if
    (syntax-rules ()
        [(_ condition alternative)
            (when condition alternative)]
        [(_ condition alternative consequent)
            (if condition alternative consequent)]))
```

Figure 3.4: Macro to generate Processing's `if` statements

Figure 3.4 expands either to Racket's `if` or to `when`, in the cases that there is no consequent branch. On the other hand, loops such as `while` and `for`, are implemented using a named `let` form. The code example shown in figure 3.5 demonstrates the implementation of `while` using a `let`.

Moreover, to implement `continue` and break in these loops, similarly to the `return` statements in function definitions, we used escape continuations. In the case of `continue`, the body of each loop is wrapped in a `let/ec` form. For `break`, we wrap `let/ec` to the whole loop definition. Figure 3.6 illustrates the generated code of a `for` loop, using `let/ec` for `break` and `continue`.

As escape continuations are expensive operations, these loops can benefit from an optimization, by detecting if a `break` or `continue` exists in the loop's body, therefore not generating the `let/ec` if no `break` of `continue` statement are encountered.

```
(let while-loop ()
    (when test
        body
        (while-loop)))
```

Figure 3.5: Processing's `while` loop implemented in Racket using a named `let`

```
(let/ec break
    init
    (let for-loop ()
        (when test
            (let/ec continue body)
            increment
            (for-loop))))
```

Figure 3.6: Processing's `for` loop implemented in Racket using a named `let` and `let/ec`

Static & Active Mode

Processing's static mode (illustrated in figure 3.7) provides a straightforward way of quickly writing Processing scripts by implementing a list of statements to produce a design. On the other hand, Processing code is considered to be in *Active* Mode (figure 3.8) when a function or a method is defined in the current compilation unit.

```
// setup
background(255);
smooth();
noStroke();


// draw ellipse
fill(255,0,0);
ellipse(20,20,16,16);
```

Figure 3.7: Processing example in *Static* mode

```
void setup() {
    background(255);
    smooth();
    noStroke();
}

void draw() {
    fill(255,0,0);
    ellipse(20,20,16,16);
}
```

Figure 3.8: Processing example in *Active* mode

To correctly support Processing's distinctions between *Active* and *Static* mode, custom checks were added for some of the parser's rule, signalling that the current compilation unit is in *Active* mode. Afterwards, as in *Active* mode global statements are restricted (e.g. if, while, for, print, etc), all global statements are wrapped in a macro that at compile checks if the global statement is valid or not in the current mode, thus implementing Processing's *Static* and *Active* mode semantics.

Draw & Setup

Each Processing *sketch* in active mode has a custom execution semantics. Processing users can define two functions to aid their design process: the `setup` and `draw` functions. In `setup`, he user can define the initial environment properties, while in `draw` executes the code that draws the design. A custom macro (illustrated in figure 3.9) was developed to implement Processing's `setup` and `draw` semantics.

```
(define-syntax (p-initialize stx)
    (syntax-case stx ()
        [(_)
         (with-syntax
             ([setup (datum->syntax stx 'setup-fn)]
              [draw  (datum->syntax stx 'draw-fn)])
            (cond
                [(and (identifier-binding #'setup 0)
                      (identifier-binding #'draw 0))
                 #'(begin (setup) (draw))]
                [(identifier-binding #'setup 0) #'(setup)]
                [(identifier-binding #'draw 0)  #'(draw)]
                [else #'(void)]))]))
```

Figure 3.9: Macro to generate Processing's `setup` and `draw` mechanisms

The `p-initialize` macro ensures that the `setup` and `draw` functions are called, if defined by the user, and it is implemented by using Racket's `identifier-binding` to check if the `setup-fn` and `draw-fn` are bound in the current environment, generating code to execute them if so.

Classes

An initial implementation of classes has been developed, mapping Processing classes into Racket classes and reusing some of ProfessorJ's ideas. Instance methods are translated directly into Racket methods, therefore `public void foo()` would be translated to `(define/public (foo-fn) ...)`. On the other hand, we foresee issues with static methods and constructors. Static methods can be 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` would be translated to `(define (Foo.foo-fn)...)`. Also, as Processing can have multiple constructors, we can solve this problem by adapting `new` to find the appropriate constructor to initialize the object.

The work on Processing's class system is still very rudimentary, as it is lacking inheritance and interfaces. Furthermore, many of the type-checking rules have not been developed for these language concepts. The implementation of these features have been purposely put off, as they are mostly used by advanced Processing users and not by the beginner Processing programmer.

3.3 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 primitives that provide common design abstractions aiding users during their development process. The following sections describe how parts of the runtime were implemented in greater detail.

3.3.1 Primitive Operations & Built-in Classes

Most of Processing's primitive operations are available directly in Racket, such as `sin`, `cos`, `ceil`, etc. Our implementation maps these operations directly to the corresponding Racket form. Nonetheless, built-in classes such as `PVector` (which abstracts 2D or 3D vectors, useful to describe positions or velocities) are implemented directly in Racket, allowing for greater interoperability with the language. To implement these primitives, we analysed the original Processing environment and other Processing implementations, namely Processing.js (section 2.3.1).

3.3.2 Drawing Primitives

Processing's drawing paradigm is heavily based on OpenGL's semantics, using the traditional push/pop-matrix style. As previously stated, we used Rosetta's 2D and 3D modelling primitives and abstractions. This not only enables us to generate designs in an OpenGL context (similar to Processing's original environment), but also gives us access to several CAD back-ends, such as AutoCAD or Rhinoceros 3D. Nonetheless, custom interface adjustments are required to implement many of these drawing primitives, as not every Processing primitive maps directly into Rosetta's.

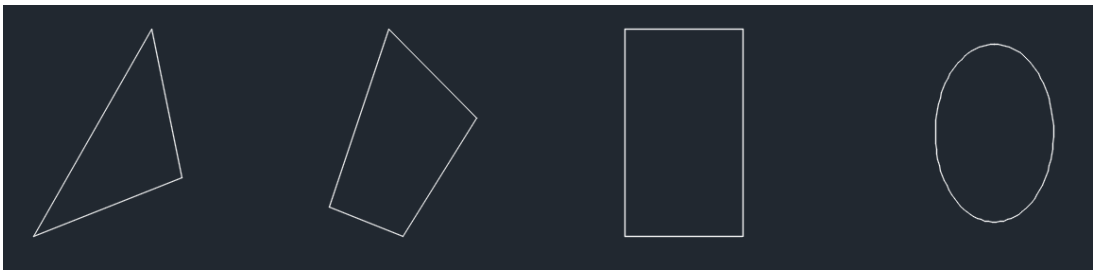


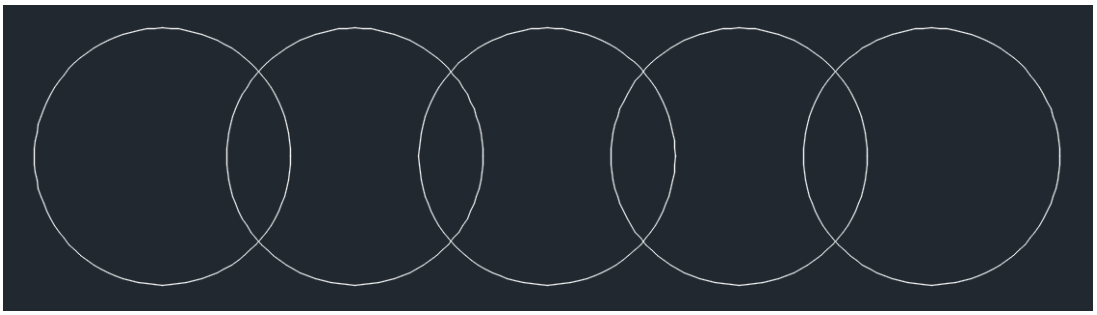
Figure 3.10: Generating 2D shapes in AutoCAD using P2R

Many of Processing's 2D primitives are directly available in Rosetta, therefore primitives such as `line`, `ellipse`, `arc`, `rect`, `quad`, and `triangle` were easily implemented, requiring only small adjustments. For example, the `triangle` primitive built using Rosetta's `polygon` (receives a list of points, creating a polygonal shape), that in this case receives the triangle's three points. Figure 3.10 shows the use of `triangle`, `quad`, `rect`, and `ellipse` in AutoCAD.

To have a better feel how these primitives can be used in P2R, consider the following drawing and its respective code, which are illustrated in figure 3.11. Note that the `backend` function provided by Rosetta, allows the user to define in which CAD back-end the design will be generated. In this simple example, we illustrate the use of a `for` loop to generate a series of circles, using Processing's `ellipse` primitive. Clearly this simple example can easily be modified to generate additional circles (modifying the loop's test condition) or generated a façade of circles (by adding an additional inner loop to generate circles along the y-axis).

Processing provides a rather poor set of 3D modelling primitives, only providing the `box` and `sphere` primitives. If users that want to model other complex forms must explicitly define them by using `beginShape` and `endShape` primitives, which enables them to define a set of vertices and how they are connected (i.e. using lines, triangles, or quad strips).

As the current Processing environment is rather poor in these modelling operations, we augment it with a set of additional of 3D modelling primitives provided by Rosetta (illustrated in figure 3.12), such as `cylinder`, `cone`, `pyramid`, or `torus`.



```
backend(autocad);
for(int i = 0; i < 5; i++)
    ellipse(i*75, 0, 50, 50);
```

Figure 3.11: Generating circles in AutoCAD using P2R

Figure 3.12: Example of 3D modelling primitives in P2R

Furthermore, Processing offers users a set of transformation primitives such as `rotate`, `scale`, and `translate` to transform and manipulate their shapes. Using Rosetta, we can expand this set of transformations with primitives such as `union`, `intersection`, `subtraction`, `loft`, or `sweep`, which are heavily used in architectural work. For example, figure 3.13 illustrates how easily a `box` and a `sphere` can be joined using the `union` operation. Similarly, figure 3.14 and 3.15 show how they can be used with `intersection` and `subtraction`.

Although Processing provides some abstractions, such as `PVector` to encapsulate vectors, it does not have an appropriate abstraction for coordinates. Usually, users solve this issue by passing points around in a array or even by passing each coordinate point information individually, resulting in long function headers and additional verbosity in the program. On the other hand, Rosetta has custom mechanisms to abstract coordinate systems, namely cartesian (`xyz`), polar (`pol`), and cylindrical (`cyl`) which can be used and combined interchangeably. As a result, these abstractions (`xyz`, `pol`, and `cyl`) are made available in our system, so that users can take advantage of them in their designs, enabling them to use an intuitive and simple mechanism to manipulate and build their drawings.

Figure 3.16 illustrates how one can easily switch from a 2D (figure 3.11) to a similar 3D representation. The reader will node that the main difference is the primitive used (in this case a `torus`). Also, note that in this case the `xyz` primitive is used to encapsulate the coordinate of the torus.

3.3.3 Types in the runtime

Processing is a statically typed language, hence the runtime libraries have type information of their bindings. However, as our runtime is built directly in Racket, which is dynamically typed, at compile time the type-checking mechanism does not have the type information of these runtime bindings. Moreover, these bindings have to be available for our custom scoping mechanism to work. As a result, we had to develop a mechanism to provide types in our runtime, making the bindings available to our scopes, and,

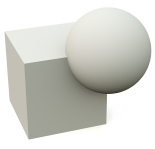


Figure 3.13: The `union` of a `sphere` with a `box`

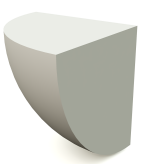


Figure 3.14: The `intersection` of a `sphere` with a `box`

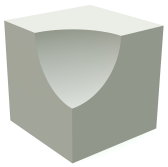
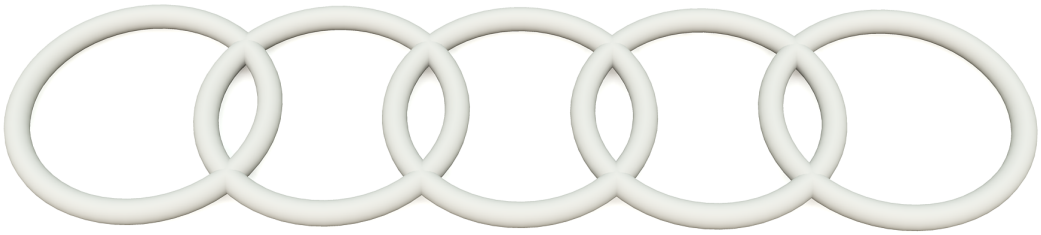


Figure 3.15: The `subtraction` of a `sphere` to a `box`



```
backend(autocad);
for(int i = 0; i < 5; i++)
    torus(xyz(i*75,0,0), 50, 5);
```

Figure 3.16: Generating a tours chain in `AutoCAD` using P2R

at the same time, making it work with our type-checking mechanism.

Our approach was to consider that any binding that was provided by the runtime would use an opaque type that suppresses the type-checking mechanism. This is achieved by altering the type hierarchy to support this opaque type, so that the argument and return types of runtime functions work seamlessly with our type-checker. Furthermore, we had to dynamically load the bindings that were available in our runtime, to make them accessible in our custom scope system.

Moreover, for the implementation of Processing's functions and primitives in Racket, we created a custom macro that associates type information to each defined binding. This way the type-checker can correctly verify if types are compatible, and allows the scoping mechanism to work with the runtime bindings. We used both mechanisms in conjunction, by defining our runtime bindings with our macro, and using the opaque type to abstract certain return and argument types. Using these mechanisms, we have the advantage of also emulating function overloading, due to our name mangling procedure. To illustrate this mechanism, consider figure 3.17, which shows how we can add type information to the `triangle` primitive. Observe, that we have added the return and argument type information to the `triangle` primitive, and, at the same time, taking advantage Rosetta's `polygon` primitive.

```
(deifine-types (triangle [float x1] [float y1]
                         [float x2] [float y2]
                         [float x3] [float y3]
                         -> float)
    (polygon (xy x1 y2) (xy x2 y2) (xy x3 y3)))
```

Figure 3.17: Implementing the `triangle` primitive in our runtime using `define-types`

Alternatively, instead of using a custom macro, we could of written Processing's APIs in `typed/racket` [Tobin-Hochstadt et al., 2011], as types can be associated to Racket definitions. Yet, this alternative was not used due to possible type incompatibilities with Processing's and `typed/racket`'s type hierarchy. Secondly, because the gain of using `typed/racket` would not be significant as Rosetta and the compiler are written in untyped Racket.

3.4 Interoperability

One of the advantages of developing a source-to-source compiler is the possibility of accessing 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 [Tobin-Hochstadt et al., 2011] and powerful hygienic macro system [Flatt, 2002] 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 [Flatt and Findler, 2011d], 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. In Racket, any character may appear in an identifier (excluding white-spaces and special characters, namely `( ) { } [ ] ; | ' ‘`). 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!` is 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 Processing code. Table 3.1 summarizes the translation mechanisms applied to imported names. Notwithstanding, as developers may not be satisfied with our 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 and save the exported bindings. As Racket is dynamically typed, we use a special type for arguments and return types, that the type-checker skips, thus delaying errors to occur at runtime. To illustrate the interoperability mechanism consider the `foo-bar` module shown in figure 3.18, which provides `foo-bar` and `foo?`, and the Processing code illustrated in figure 3.19.

As illustrated in figure 3.19, the function `checkFoo` uses the `foo-bar` procedure from the `foo-bar.rkt` module. Note that our translation procedure has been applied to all provided bindings from the `foo-bar.rkt` module. 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

Racket Identifier	Processing Identifier
fooBar	fooBar
fooBar!	fooBarBang
fooBar#	fooBarSharp
fooBar%	fooBarPercent
foo*Bar	fooStarBar
foo+Bar	fooPlusBar
foo.Bar	fooDotBar
foo/Bar	fooSlashBar
foo<Bar	fooLessBar
fooBar=	fooBarEqual
foo>Bar	fooGreaterBar
fooBar~	fooBarTilde
foo@bar	fooAtBar
fooBar?	fooBarP
foo:Bar	fooColonBar

Table 3.1: Racket to Processing translation rules for identifiers

of the module into their mangled form. To compute this list, we used Racket's `module->exports` primitive to provide us a list of bindings exported by the module. However, this information does not suffice, as we need to know the arity of each exported binding to be able to produced binding compatible with our generated code. Therefore, we analysed each exported binding by `module->exports`, and retrieved its arity using `procedure-arity`. 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, by using Racket's `filtered-in` primitive to translate the original binding to the new mangled form.

```
#lang racket

(provide foo-bar)

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

Figure 3.18: The `foo-bar` module in Racket

```
#lang processing

require "foo-bar.rkt";

void checkFoo(String s) {
    println(fooBar(s));
}
```

Figure 3.19: The `checkFoo` function in Processing using `foo-bar`

3.5 Integration with DrRacket

The Racket language offers its users a pedagogical IDE, DrRacket. As Racket encourages users to take advantage of its language creation mechanisms, naturally it also allows users to configure and adapt its IDE services to work with new languages. Therefore, as our solution is built on the Racket platform, and as Processing developers are already familiar with the Processing Development Environment (PDE), adapting DrRacket to support Processing would provide a similar development environment to Processing developers. In turn, this would permit developers to easily make the transition to our system. The following chapter will describe the effort needed to adapt and configure DrRacket for the Processing language.

3.5.1 Comparing both IDEs

Analysing both IDEs, we observe that they share many features. Table 3.2 compares the different IDE services supported by the PDE and DrRacket.

	Debugger	Profiler	Syntax Checking	Syntax Highlighting	REPL
PDE	Yes	Yes	No	Yes	No
DrRacket	Yes	Yes	Yes	Yes	Yes

Table 3.2: Comparison between the Processing Development Environment and DrRacket

Analysing table 3.2, we observe that both environments share many features, including a debugger, profiler, and syntax highlighting. The PDE allows the use of an external mode to do debugging, while Racket offers an internal debugger. As for profiler support, in the case of Processing, external tools have to be installed in order to support profiling, contrasting with the Racket environment, where a library already exists.

DrRacket provides additional IDE features to its users, supporting syntax checking and a Read-Eval-Print-Loop (REPL). Syntax checking provides mechanisms to check if the program is written using correct syntax, tracking variable definition and their occurrences throughout the code. On the other hand, the REPL is a common feature that is present in many LISP descendants, providing users a mechanism to test specific parts of their code. A REPL is a good mechanisms for beginners to quickly test their ideas and learn from their mistakes.

3.5.2 Adapting Processing to DrRacket

DrRacket uses Racket's syntax-objects [Flatt and Findler, 2013] to refer to the original source code, by encapsulating the code's s-expression, source location information and lexical-binding information. As a result, during the *Parsing* phase, each generated token saves the location of the piece of code it refers to. This information is kept throughout the compilation process by storing it in each `ast-node`. Finally, the syntax-objects produced by our *Code Generation* phase are constructed using this source location

information. This way, DrRacket's features which rely on the source location information, such as the syntax checking mechanism will work for Processing.

To enable syntax highlighting, we had to define a `language-info` function that allows developers to configure a module language with arbitrary properties to be used by external tools. For syntax highlighting, DrRacket uses the `color-lexer` property, which uses a custom scanner to consume the text and return the appropriate symbol to colourize the consumed token. This was implemented by reusing a significant part of our lexer specification, adapting it produce tokens that contain values from Racket's colour scheme. As a result, we provide a more comfortable coding experience for users with DrRacket, where users can now distinguish elements such as keywords, identifiers, comments, literals, etc.

Lastly, a relevant feature that Racket offers is a REPL, which is common in many LISP descendants. This feature was 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 as a result. Therefore, we created a new parser rule to implement REPL interactions, adding it to the 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 handling.

Chapter 4

Evaluation

The main reason for the development of our Processing to Racket compiler was to promote its use in the context of architectural problems. This chapter illustrates the main results of our implementation. Firstly, by illustrating how Processing code can access different Computer-Aided Design (CAD) back-ends (presented in section 4.1) and how Processing's 3D modelling tools were augmented for architectural work (illustrated in section 4.2). Secondly, section 4.3 shows how DrRacket can provide a similar development environment to Processing users. Examples of how external Racket libraries can be explored by Processing code are presented in section 4.4. Finally, although performance was not one of our goals, in section 4.5 we provide a performance comparison of our implementation versus the original Processing environment.

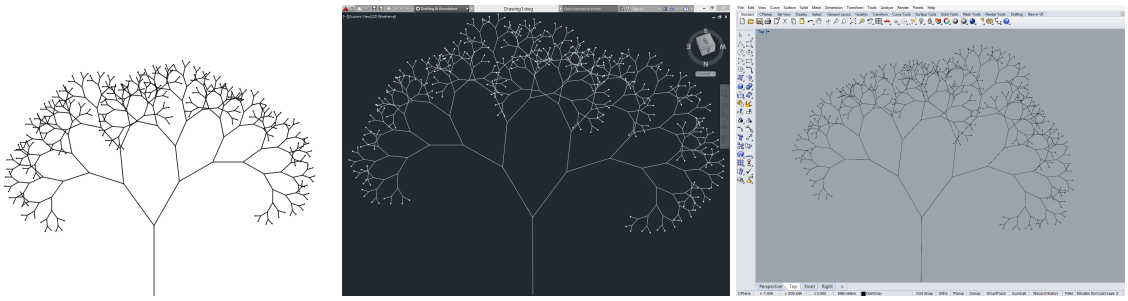
4.1 Connecting Processing with CAD applications

The main goal of our implementation was to allow Processing scripts to be used in CAD applications. To illustrate how our implementation can explore the different CAD environments, consider the Processing example illustrated in figure 4.1.

```
float max = 0.79, min = 0.69, da = PI/6, db = PI/5;
void tree2D(float x, float y, float l, float a) {
    float x2 = x - l * cos(a), y2 = y - l * sin(a);
    line(x, y, x2, y2);
    if (len < 10) {
        ellipse(x2 , y2, 0.6, 0.6);
    } else {
        tree2D(x2, y2, random(min, max) * l, a + da);
        tree2D(x2, y2, random(min, max) * l, a - db);
    }
}
```

Figure 4.1: Processing code to generate a 2D fractal tree

This example generates a fractal tree that gradually reduces the length of each tree branch, using Processing's `line` and `ellipse` primitives to create its branches and leaves. Note that the rate by which



```
tree(0, 0, 100, PI/2);
```

```
backend(autocad);
tree(0, 0, 100, PI/2);
```

```
backend(rhinoceros);
tree(0, 0, 100, PI/2);
```

Figure 4.2: Fractal tree generated in Processing, AutoCAD, and Rhinoceros 3D (from left to right)

the length of each branch is controlled by a randomly generated number.

Our goal was to use the same Processing code, and, with it, generate the same drawing in different CAD applications. Figure 4.2 presents the use of `tree2D` in the original Processing environment, and then AutoCAD and Rhinoceros 3D. Observe that the only change made to the original Processing invocation of `tree2D` was to add an additional primitive to indicate what rendering back-end to use. Also note that the generated drawings have slight variations, due to the use of the `random` function to produce shorter tree branches.

Notwithstanding, these 2D illustrations have limited applicability for architectural work. They can however be a base idea to explore other designs. For instance, figure 4.3 illustrates an example of 3D tree, showing how we can quickly migrate from a simple design to a more complex and architecturally useful creation. In this case, the branches were modelled using the `coneFrustum` primitive, while the endings were modelled using a `box`, thus creating a tree inspired column. Note that the `coneFrustum` primitive is not available in the original Processing environment.

```
void tree3D(float x, float y,float z, float l, float a1, float a2, float r) {
    float x2 = x + l * cos(a1) * sin(a2),
          y2 = y + l * sin(a1) * cos(a2),
          z2 = z + l * cos(a2),
          r2 = r * 0.55;
    coneFrustum(xyz(x, y, z), r, xyz(x2, y2, z2), r2);
    if (l < 7) {
        box(xyz(x2 - 3, y2 - 3, z2 - 0.5), 6, 6, 1);
    }  else {
        tree3D(x2, y2, z2, l * 0.7, a1 + PI * 1/4, PI/4, r2);
        tree3D(x2, y2, z2, l * 0.7, a1 + PI * 3/4, PI/4, r2);
        tree3D(x2, y2, z2, l * 0.7, a1 + PI * 5/4, PI/4, r2);
        tree3D(x2, y2, z2, l * 0.7, a1 + PI * 7/4, PI/4, r2);
    }
}
```

Figure 4.3: Processing code to generate a 3D tree

These examples can be generated for different CAD applications, using the `backend` primitive. Figure 4.4 shows the execution of `tree3D` for AutoCAD and Rhinoceros 3D, showing the potential that our

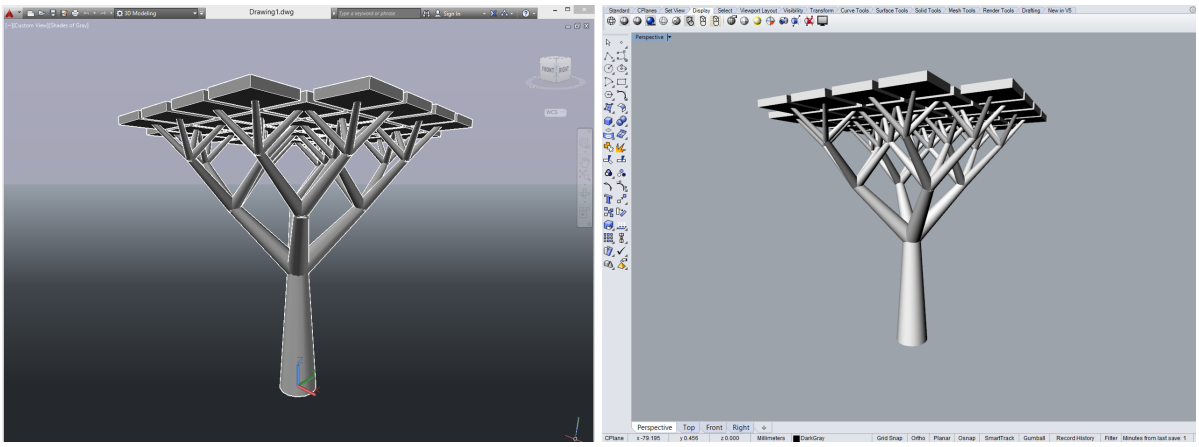


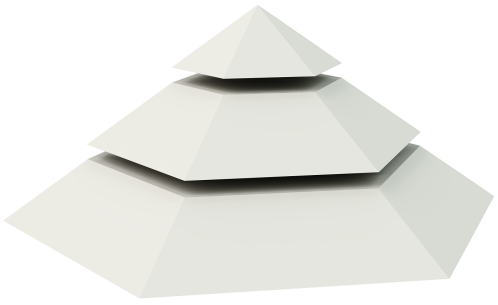
Figure 4.4: 3D tree generated in AutoCAD and Rhinoceros 3D (from left to right)

Processing implementation brings to architecture work. With little effort, architects can run Processing scripts in a CAD or Building Information Modelling (BIM) application. Moreover, architects can quickly migrate from a 2D drawing idea to a 3D model, and then explore it for architectural purposes.

4.2 Augmenting Processing's 3D modelling primitives

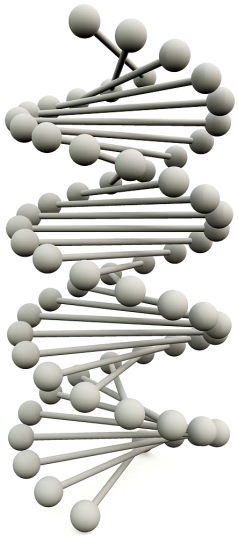
Processing provides a very poor set of 3D primitives (namely `box` and `sphere`), which are insufficient for architectural modelling. Therefore, we augmented Processing with a large set of primitive shapes (e.g., cylinder, pyramid, torus, etc), boolean operations (e.g., union, intersection, and difference), and many more (e.g., extrusion, sweeping, and lofting), reducing the effort needed to generate complex designs.

Figure 4.5 shows a quick model of a simple pyramid tower using the `pyramid` and `pyramidFrustum` primitives in our system. Note that both of these primitives have been made available by our implementation. The primitive `pyramidFrustum` takes as input the number of sides, a base point, a base radius, the initial angle, the top point, and the top radius. Similarly, the `pyramid` function receives the number of sides, base point, base radius, initial angle, and top point.



```
pyramidFrustum(6, xyz(0,0,0), 15, 0, xyz(0,0,4), 10);
pyramidFrustum(6, xyz(0,0,5), 10, 0, xyz(0,0,9), 5);
pyramid(6, xyz(0,0,10), 5, 0, xyz(0,0,14));
```

Figure 4.5: Building a Pyramid Tower using P2R



```
#lang processing

float r = 15, h = 2;

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

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

    if(a > 0)
        helix(z+h, a-PI/8);
}
```

Figure 4.6: Double helix generated from Processing code in AutoCAD

This example shows how easy it is for users to create new 3D models by using our set of primitives. On the contrary, to implement the same example in the original Processing environment, we would have to use Processing's `beginShape` and `endShape` operations, which require the user to detail all of the vertices of the pyramid, describing how each surface is constructed and connected.

Another example is the double helix, presented in figure 4.6. This model is created by using a recursive function that repeatedly creates 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), that takes advantage of the `cylinder` and `xyz` primitives, to render the helix in AutoCAD.

To illustrate what our compiler is doing behind the scenes, figure 4.7 presents the intermediate Racket code produced by our compiler. The first point worth mentioning is that all function identifiers are renamed to support multiple namespaces. We can see that `helix` identifier is translated to `helix-FF-fn`, where the **F** indicates that the function has 2 arguments that are of type `float`. Functions and macros such as `p-mul`, `p-sub`, or `p-call`, are defined in the *Language* modules (presented in section 3.1.2), implementing Processing's semantics.

Moreover, other runtime bindings such as `cos-F-fn` or `sphere-OF-fn`, illustrate the usage of our mechanism of adding types to runtime bindings (described in section 3.3.3). Variable definitions are translated using the `p-declaration` macro, that generates a Racket `define-values` form using a sequence of `identifier` and `value` pairs. Mathematical operators (`p-mul`, `p-add`) are implemented as Racket functions, yet, do not overflow as in Processing. Observe that all function definitions have their body wrapped in a `let/ec` form, injected to support `return` statements within functions.

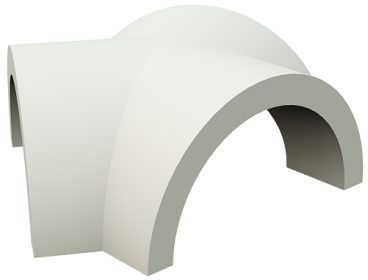
Another example that demonstrates the advantages of using our implementation's transformation primitives in Processing is presented in figure 4.8. This example represents the `union` of three rotated cylinders, followed by the `subtraction` of another `union` of cylinders with a smaller radius. Finally, the lower part is removed by subtracting a `box` to the model. This particular example is much more difficult to express using only the operations available in the original Processing language. Observe that in this

```
(p-declaration (r 15.0) (h 2.0))
(p-function (helix-FF-fn z a)
  (let/ec return
    (p-block
      (p-declaration (x1 (p-mul r (p-call cos-F-fn a)))
                     (y1 (p-mul r (p-call sin-F-fn a))))
      (p-declaration (x2 (p-mul r (p-call cos-F-fn (p-add PI a))))
                     (y2 (p-mul r (p-call sin-F-fn (p-add PI a)))))
      (p-call sphere-OF-fn (p-call xyz-FFF-fn x1 y1 z) 2)
      (p-call cylinder-OFO-fn
         (p-call xyz-FFF-fn x1 y1 z) 0.5 (p-call xyz-FFF-fn x2 y2 z))
      (p-call sphere-OF-fn (p-call xyz-FFF-fn x2 y2 z) 2)
      (p-if (p-gt a 0)
        (p-call helix-FF-fn (p-add z h) (p-sub a (p-div PI 8.0)))))))
```

Figure 4.7: Generated Racket code for `helix`

example we are using the `pol` abstraction for polar coordinates, that in this case are more adequate to specify the cylinders' position.

Finally, figure 4.9 presents a concrete example of an idealized building rendered in Firefox using Processing.js (described in section 2.3.1). This example is implemented with the aid of external Processing libraries, namely *Peasycam* and *HE_Mesh2014* (the code for this example can be seen at `www.openprocessing.org/sketch/20880`). To demonstrate how easily we can achieve similar results using our system, we developed the same example using our Processing implementation, generating the model in AutoCAD (shown in figure 4.10). For this model, the only 3D modelling operations used were `box` and `subtraction`, using no external libraries.



```
backend(autocad);
Object p0 = xyz(0,0,0);

Object outer =
    union(cylinder(p0, 10, pol(10,1/3*TWO_PI)),
          cylinder(p0, 10, pol(10,2/3*TWO_PI)),
          cylinder(p0, 10, pol(10,3/3*TWO_PI)));
Object inner =
    union(cylinder(p0, 8, pol(10,1/3*TWO_PI)),
          cylinder(p0, 8, pol(10,2/3*TWO_PI)),
          cylinder(p0, 8, pol(10,3/3*TWO_PI)));
Object hollow = subtraction(outer, inner),
       box = box(xyz(-15,-15,-10), xyz(15,15,0));

subtraction(hollow, box);
```

Figure 4.8: Structure made from boolean operations applied to rotated cylinders

The previously presented examples, demonstrate how our implementation has enlarged the original Processing environment, supporting a set of additional primitives and transformations that help architects take advantage of our system for architectural work. Moreover, we can easily reach the same visual results of the original Processing environment by using our implementation and its additional primitives, without resorting to external libraries.

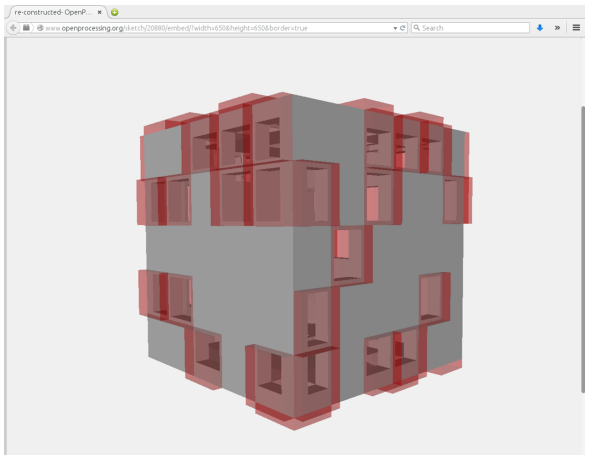


Figure 4.9: Perforated building generated in Firefox using Processing.js

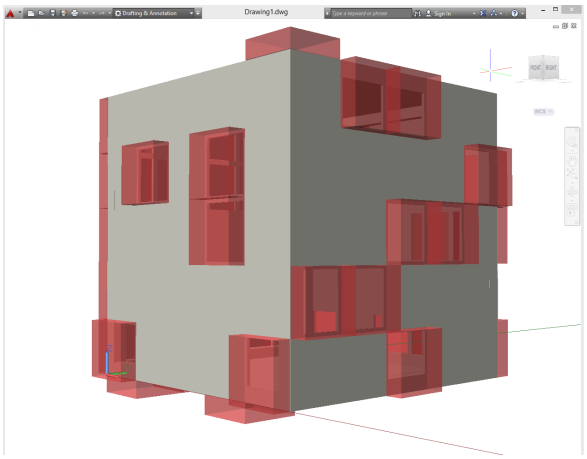


Figure 4.10: Perforated building generated in AutoCAD using P2R

4.3 Integration with DrRacket

An important feature that we found essential for any implementation of the Processing language was the existence of an Interactive Development Environment (IDE) tailored to the language. Moreover, as Racket offers an IDE (DrRacket), we adapted it to support the Processing language.

Figure 4.11 shows both DrRacket, using our Processing implementation and the Processing Development Environment (PDE). Taking a closer look, we immediately observe that both environments are similar, offering a tabbed editor equipped with syntax highlighting. Note that in DrRacket (on the left), the `#lang` declaration allows users to specify what language of the Racket ecosystem to use in the editor.

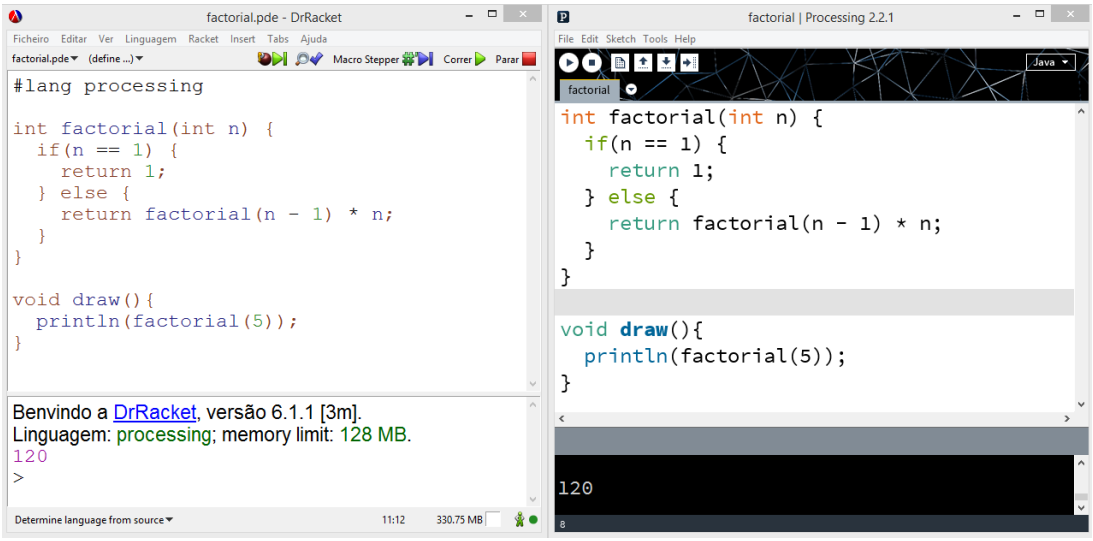


Figure 4.11: Similarities between DrRacket using Processing and the PDE

To further illustrate the similarities of both development environments, consider figure 4.12. This example illustrates how both environments present error messages to users. As we can observe, both offer a similar mechanism to highlight errors when they occur and both present an appropriate message related to the error at hand.

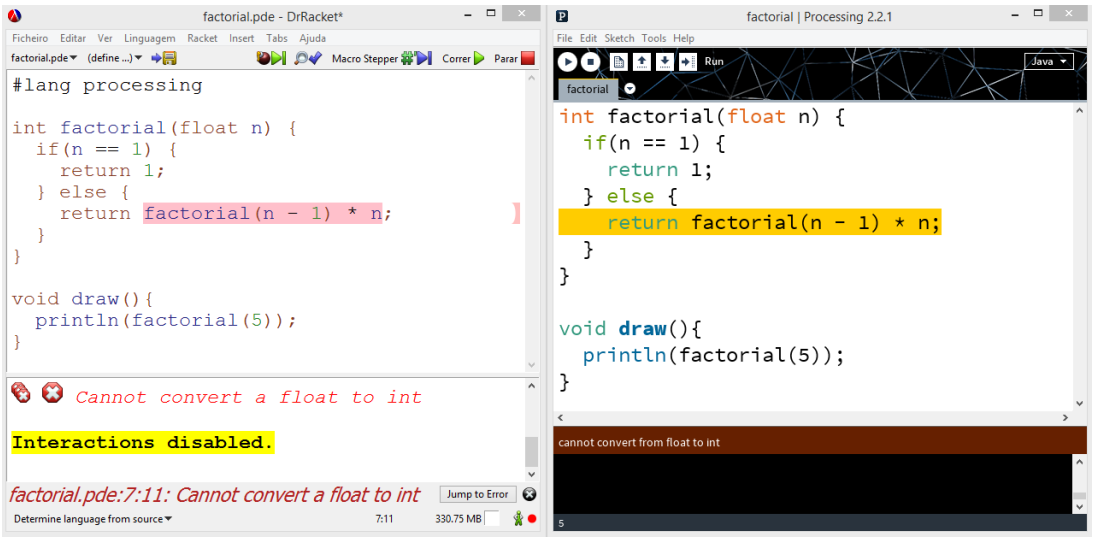


Figure 4.12: Error messages in DrRacket and the PDE

Finally, in figure 4.13 we illustrate the new capabilities that the adaptation of DrRacket to the Processing language brings to our users. Firstly, the definition tracking mechanism available in DrRacket was easily adapted to support Processing code. This mechanism allows users to have visual representation of how their definitions are used in their code, presenting an useful features for beginner programmers.

Secondly, we adapted DrRacket's Read-Eval-Print-Loop (REPL) interactions to work with Processing code, therefore offering an interactive approach to programming using the Processing language, a feature which is unavailable in other Processing environments. The approach was to support Processing's syntax in the REPL, however, as Processing is a statement base language, it does not produce an expression as a result. For instance, executing a function call (e.g `factorial(5)`) would not return a value. To do so, we would need to use a function such as `println` to produce a value through a side-effect. Thus, to obtain the result of `factorial(5)`, we would have to wrap it in `println(factorial(5))`, that then would print out the result (in this case 120).

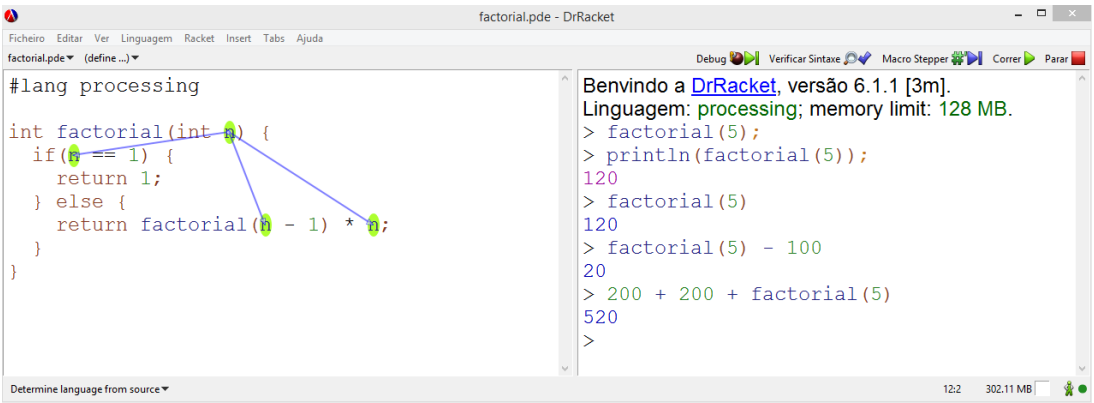


Figure 4.13: Definition tracking and REPL interactions with Processing code in DrRacket

As this is not the normal behaviour of a REPL, we adapted the parser to treat these REPL interactions in a different manner. Our change was to add support for expressions such as `factorial(5) + 200`, which would immediately return the actual result (in this case 320), offering a more REPL like behaviour

to the Processing language.

As illustrated, we provide a nearly identical IDE for Processing users in DrRacket, with additions that are not present in original Processing environment, such as a REPL. In our view, this is a very helpful feature for beginners, as it allows users to immediately and interactively test out new design ideas. Furthermore, DrRacket also provides mechanisms to customize the editor to users needs (new editor modes, custom syntax highlighting, etc), and things like a debugger and profiler are also available for users to explore. Finally, as DrRacket allows users to quickly transition from one language to another by changing the `#lang` declaration, users have the ability of learning a development environment that can be used to explore other programming languages.

4.4 Interoperability with Racket

Developing a source-to-source compiler has the advantage of allowing us to explore libraries written in another language. We demonstrate how Racket libraries can be access in our Processing code, namely by using libraries that were previously built for 3D modelling.

Consider the Processing code presented in 4.14. The `mosaics` procedure generates a grid 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.

```
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);
}
```

Figure 4.14: Processing code to generate mosaics in P2R

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 in our Processing code. Secondly, we `require` `draw.rkt`, which allows us to access `fullArc` and `frame`. These functions enabling us to generate the arcs and produce the enclosing bound-

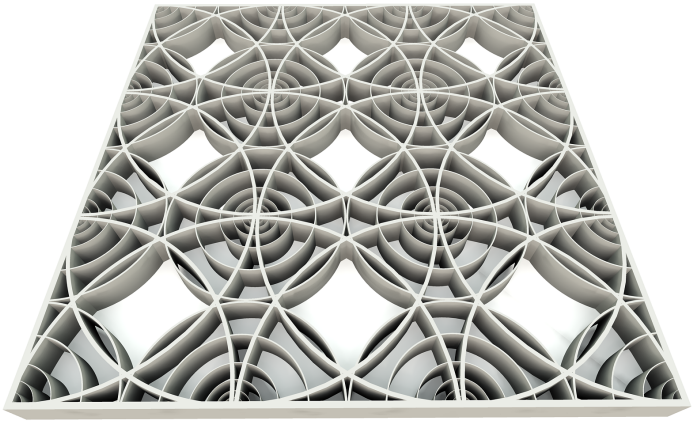
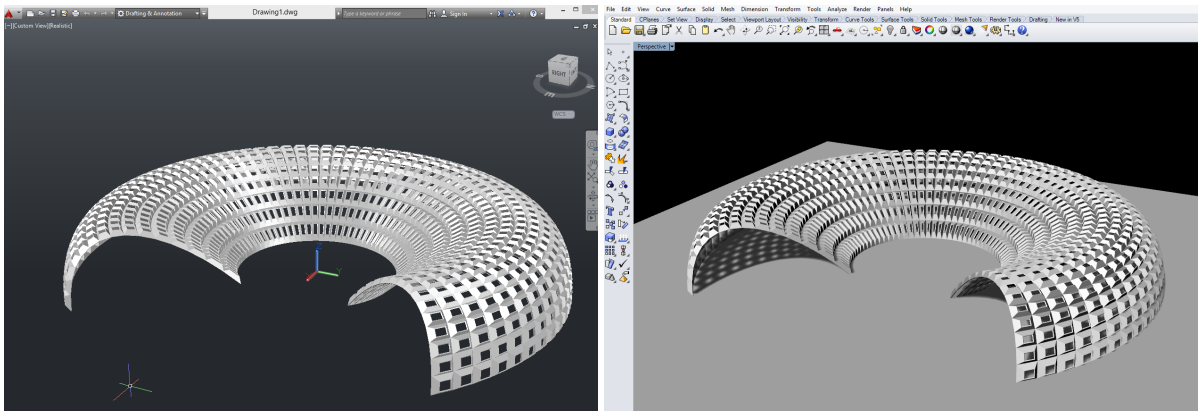


Figure 4.15: Mosaic pattern generated in AutoCAD

ary. This example shows how we can use previously created drawing libraries in our implementation. Figure 4.15 illustrates the generation of a mosaic pattern in AutoCAD.

Finally, we demonstrate another example (shown in figure 4.16) 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 `require`s `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.



```
require "elliptic-torus.rkt";
float aMin = QUARTER_PI, aMax = 7 * QUARTER_PI;
ellipticTorus(xyz (0,0,0), 0.005, 0.03, 0.5, aMin, aMax, 0, TWO_PI);
```

Figure 4.16: Elliptic torus generated in AutoCAD and Rhinoceros 3D

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

4.5 Performance

Although performance was not our main goal, we analysed our implementation's performance, comparing it with the standard processing environment. As our fundamental goals were to augment Processing with additional CAD back-ends and the ability to explore 3D primitives for architectural work, no performance optimizations were made to our generated code. Therefore, our expectation was that our implementation would perform poorly when comparing with the original Processing environment.

These foreseen performance problems are explained by the following reasons: firstly, because the Processing language is implemented on top of the Java platform, which generates highly optimized code; secondly, we are limited to the speed of our target language (i.e. Racket), which is dynamically typed and therefore has a greater performance overhead when comparing to Java (which is statically typed, i.e. additional optimizations can be made base on type information); finally, our generated code relies on expensive operations to implement Processing's semantics, such as *escape-continuations*, which are one of the main constraints for performance.

For this reason, to analyse the performance of our implementation, we created three different test cases to show how our system performs. The first test case, illustrated in figure 4.17, computes the value of `fibonacci(40)`; the second test case, presented in figure 4.18, computes the value of the `ackermann` function for inputs 3 and 7; lastly, the third test, computes the value of the factorial of 1000, which is shown in figure 4.19.

Our analysis uses the Processing language as the benchmark for comparison. We ran tests for the Racket language to see how our implementation performed against it, as the performance of our system should, in theory, be as fast as Racket. To get a notion of how our implementation performs compared to other Processing implementations, we analysed the performance of Processing.js. Finally, as we knew that `escape-continuations` were a major source of overhead, we adapted our code generation procedures to remove them. This way we were able to visualize the overhead that these constructs introduce in our implementation. This removal was accomplished in a naive way, not respecting Processing's semantics. Nevertheless, it provides a useful insight of a possible optimization to explore in our implementation. This optimization would consist in creating a control-flow analysis mechanism to remove redundant tail returns (i.e. removing unnecessary `escape-continuations`).

Table 4.1 shows the performance comparison of each test scenario against the Processing benchmark, summarizing the results obtained by our analysis. These tests were executed on a machine equipped with an AMD Phenom II X4 955 processor, using 4 GB of Ram, and running a 64-bit GNU/Linux distribution of a ArchLinux. Tests to Processing.js were executed in Firefox v37.

Analysing the execution of `fibonacci` (figure 4.17), it comes to no surprise that Processing is the faster execution. As we can clearly observe, our implementation is much slower (in this case, approximately 250 times slower) than the original Processing implementation. Moreover, analysing our version without continuations, we conclude that the bottleneck is in the use of continuations. Without them, the implementation is 2.5 times slower than Processing, and slightly faster than Processing.js (which is three times slower than Processing), presenting an acceptable performance. When comparing to Racket, we

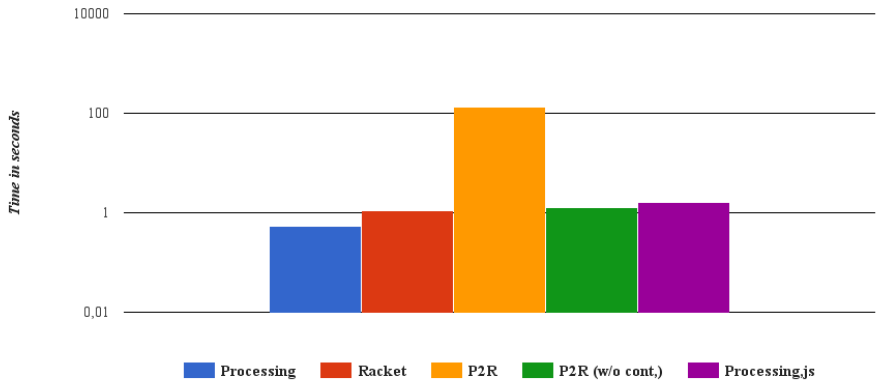


Figure 4.17: Computing the value of `fib(40)`

find that our implementation (without continuations) is still slightly slower than Racket, evidencing there is room for additional optimizations to our generated code.

Our second example (presented in figure 4.18) shows the execution of the `ackermann` function. Again, in this case, we find that our implementation is slower (55 times) than Processing. However, by removing continuations, we have a better performance than Processing and Processing.js, while having a similar performance to Racket.

Finally, we analyse the computation of the `factorial` function (figure 4.19). Note that the `factorial` function, as the input increases, quickly produces larger numbers that require arbitrary-precision arithmetic (i.e. bignum arithmetic) to be correctly represented. On the one hand, Racket already supports this type of numbers, and, therefore, our implementation takes advantage of this fact as well. On the other hand, as Processing is built on Java, its numbers have limited precision, thus to correctly implement the `factorial` function for Processing, we had to use `BigIntegers`. Although one may argue that, in this way, we are not respecting Processing's semantics and that we are paying a larger price to do arithmetic for small numbers, we argue that as the Processing language is meant for non-programmers

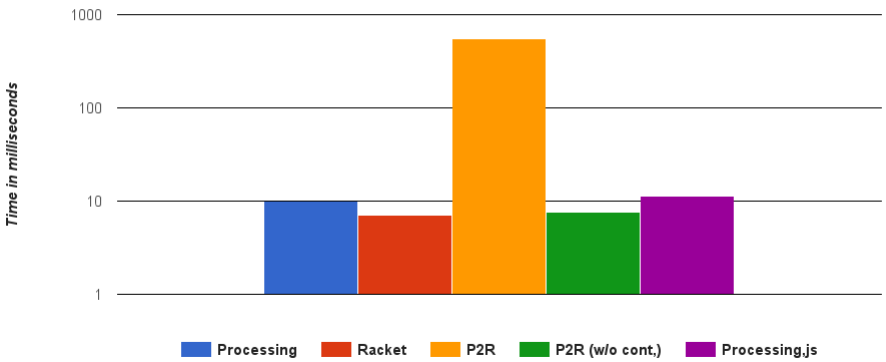


Figure 4.18: Computing the value of `ackermann(3,7)`

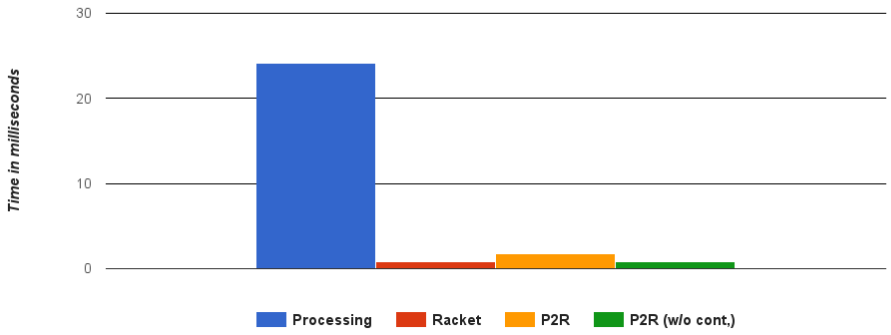


Figure 4.19: Computing the value of `factorial(1000)`

that just want to do design (and not think about arithmetic precision), it is a reasonable price to pay.

With these decisions in mind, analysing figure 4.19, we observer that Processing, in this case, performs worse than in other test scenarios. Comparing with our implementation (with continuations) it is 14 times slower, and is 33 times slower than Racket. Unfortunately, we could not get a result for Processing.js, as Firefox produces an error when running this example.

	Racket	P2R	P2R(w/o cont.)	Processing.js
`fib(40)`	2,1	250,0	2,4	3,0
`ackermann(3,7)`	0,7	55,5	0,8	1,1
`factorial(1000)`	0,033	0,072	0,033	—

Table 4.1: Performance comparison between the Processing benchmark and the different test scenarios. Slower than the Processing benchmark for values >1, while values <1 represent faster executions.

Although there is much room for improvement in our system's performance, we have a clear idea of where we have performance overhead. Our implementation (with continuations) performed poorly in all test scenarios, while the naive implementation (without continuations) performed as well as the Racket base line, clearly showing that `continuations` are an issue, and that, in some cases, can be avoided.

Ultimately, the desired performance for our implementation should be as fast as the equivalent in Racket. However, preserving semantic representation of the Processing language in Racket has a cost, which will be difficult to escape entirely. Moreover, as our implementation depends on the connection to CADs (which uses interprocess communication), there will always be a major bottleneck to communicate with the different CADs.

Chapter 5

Conclusions

Implementing Processing for traditional Computer-Aided Design (CAD) applications benefits architects and designers by allowing them to take advantage of Processing's visual capabilities. Our implementation follows the common compiler pipeline architecture, generating semantically equivalent Racket code and loading it into Racket's Virtual Machine (VM). We developed Processing's runtime and standard library in Racket, allowing for greater interoperability with Racket's language ecosystem and Rosetta's features.

Augmenting Processing with new design paradigms and abstractions, namely 3D modelling primitives (torus, cone, cylinder, etc.) and transformations (union, subtraction, loft, etc.), presents a strong reason for the architecture community to take advantage of our solution. Our implementation clearly empowers Processing's environment with modelling features that help Processing users create new designs using a more expressive modelling approach. Moreover, architects can easily access and combine several modelling primitives, enabling the creation of new designs that would be much harder to achieve in the original Processing environment.

The Processing Development Environment (PDE) is an essential feature of the Processing language, as it reduces beginners' programming learning curve. Our implementation offers a similar solution, by adapting DrRacket to support our implementation, providing a simplified editor and development environment, which is almost identical to the PDE. Additionally, we provide an important additional feature, the Read-Eval-Print-Loop (REPL), which is unavailable to other Processing implementations, providing users with a mechanism to quickly and interactivity test out new ideas.

Translating a high-level language to another enables the possibility of accessing libraries that are written in different languages. Therefore, to take advantage of Racket's multi-language ecosystem, we developed an interoperability mechanism to access any library of the Racket environment. Furthermore, the Processing language and its design approach is also being explored with other programming languages (e.g. Ruby, Python, and JavaScript). Our system also encompasses this feature, as it allows us to explore and combine Processing with any of the different languages provided by Rosetta and the Racket environment, namely Scheme, Python, JavaScript, and AutoLISP.

Having Processing in Racket allows us to connect with CAD applications, taking advantage of new

3D modelling primitives tailored for architectural work while using, at the same time, an Interactive Development Environment (IDE) tailored for the Processing language. Moreover, we have the ability of accessing libraries that are written in any language of the Racket ecosystem. For all the reasons mentioned above, our system offers compelling motives for the architecture community to explore our system in their architectural endeavours.

5.1 Future Work

Our approach was to implement parts of Processing that users need to write simple scripts. At the same time, we made available 3D modelling primitives that allow Processing to be used in the context of architectural work. Nonetheless, there are many features to be explored in the future:

- Firstly, although a large part of the language was implemented in our compiler, some parts still need to be addressed. We have developed an initial version of Processing's class system, yet many of the type-checking mechanisms were not implemented, requiring future work. Moreover, inheritance and interfaces were also left for future work.

- Although our compiler produces custom error messages, these relate to errors during the compilation process. We are still lacking Processing's exception system in our compiler. Our future goal is to map Processing's exception system into Racket's, providing a seamless experience in terms of exception handling.

- With our implementation, Processing's 3D primitives and transformation were greatly augmented. However, we still have to address and implement some of its 2D primitives. Furthermore, many of Processing's scripts have heavy interaction with keyboard and mouse events. As a result, these interactions with the chosen CAD environment could also be explored in the future.

- We have still not found a convincing solution to map Processing's colouring and textures into the different materials and layers which many CAD systems use. Moreover, Processing is also used to create new typography, thus this also could be a feature to explore in the future.

- As demonstrated by our performance analysis, we can quickly improve our implementation by performing optimizations to the code generated by our compiler. In future, we could implement a control-flow analysis mechanism to remove unnecessary continuations, which are a major source of overhead. Moreover, many other optimizations could be made to our compiler, namely seeing if generating Typed Racket [Tobin-Hochstadt et al., 2011] would bring any performance benefits to our implementation. However, premature optimization is the source of all evil; therefore, these decisions would have to be based on a further performance analysis, primarily to understand the overheads that each CAD connection has.

- The ability to use an editor mode adequate for Processing in DrRacket still needs to be addressed, as the default editor mode is specifically designed for s-expression based languages. Another

feature that would be interesting to explore in our implementation would be to have live coding (available in Ruby-Processing), thus creating a more interactive environment for users to explore. This would require a mechanism to constantly re-evaluate definitions in DrRacket.

- Having graphical REPL interactions would also be an interesting feature to have in our system. This would allow users to immediately have visual feedback of shapes and custom created geometry in the REPL, not requiring a full CAD interface to be loaded.
- Finally, it would be useful to be able to use Processing's libraries in our implementation. These provide features such as network access, sliders, buttons, PDF support, etc., which allow users to explore and experiment with different design creation processes, leading to more innovative and creative designs.

Bibliography

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2^{nd} edition, 2006. ISBN-13: 978-0321486813, ISBN-10: 0321486811.

J. Ashkenas. Ruby-processing. `https://github.com/jashkenas/ruby-processing`, 2015. Accessed: 2015-04-28.

A. Bader-Natal. Sketchpad. `http://sketchpad.cc/`, 2011. Accessed: 2015-04-28.

S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1 (2), 2004.

C. Donnelly and R. Stallman. *Bison: the Yacc-compatible parser generator*, volume 1. Free Software Foundation, 1993.

S. Efftinge and M. Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. *Software Language Engineering*, pages 197–217, 2013.

J. Feinberg, J. Gilles, and B. Alkov. Python for processing. `http://py.processing.org/`, 2014. Accessed: 2015-04-28.

R. B. Findler. Drracket: The racket programming environment. 2013.

R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. Drscheme: A pedagogic programming environment for scheme. In *Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 369–388, London, UK, 1997. Springer-Verlag. ISBN 3-540-63398-7. URL `http://dl.acm.org/citation.cfm?id=646452.692958`.

M. Flatt. Composable and compilable macros:: You want it when? *SIGPLAN Not.*, 37(9):72–83, Sept. 2002. ISSN 0362-1340. doi: 10.1145/583852.581486. URL http://doi.acm.org/10.1145/583852.581486.

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

M. Flatt and R. Findler. The racket guide, chapter 10.3 continuations. http://docs.racket-lang.org/guide/conts.html?q=continuations, 2011a. Accessed: 2014-05-05.

M. Flatt and R. Findler. The racket guide. http://docs.racket-lang.org/guide/, 2011b. Accessed: 2014-05-02.

M. Flatt and R. Findler. The racket guide, chapter 17 creating languages. http://docs.racket-lang.org/guide/languages.html, 2011c. Accessed: 2015-03-22.

M. Flatt and R. Findler. The racket guide, chapter 6.4 imports: requires. http://docs.racket-lang.org/guide/module-require.html, 2011d. Accessed: 2014-05-05.

M. Flatt and R. Findler. The racket guide, chapter 16.2.1 syntax objects. http://docs.racket-lang.org/guide/stx-obj.html, 2013. Accessed: 2015-04-27.

M. Fowler. Language workbenches: The killer-app for domain specific languages. http://martinfowler.com/articles/languageWorkbench.html, 2005. Accessed: 2014-04-20.

M. Fowler. Projectional editing. http://martinfowler.com/bliki/ProjectionalEditing.html, 2008. Accessed: 2015-03-22.

K. E. Gray and M. 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.

K. E. Gray and M. Flatt. Compiling java to plt scheme. In *Proc. 5th Workshop on Scheme and Functional Programming*, pages 53–61, 2004.

S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

L. C. Kats and E. Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869497. URL http://doi.acm.org/10.1145/1932682.1869497.

M. E. Lesk and E. Schmidt. Lex: A lexical analyzer generator, 1975.

J. Lopes and A. 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.

J. Louis-Rosenberg. Objexport. http://n-e-r-v-o-u-s.com/tools/obj/, 2013. Accessed: 2015-04-29.

J. Maeda. *Design by Numbers*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0262133547.

H. Moessenboeck. *Coco/R: A Generator for Fast Compiler Front-Ends*. Citeseer, 1990.

S. Owens. Parser tools: lex and yacc-style parsing. `http://docs.racket-lang.org/parser-tools/`, 2011. Accessed: 2014-09-22.

T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1^{st} edition, 2010. ISBN-13: 978-1934356456, ISBN-10: 193435645X.

T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2^{nd} edition, 2013. ISBN-13: 978-1934356999, ISBN-10: 1934356999.

T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

J. Resig, B. Fry, and C. Reas. Processing. js, 2008.

C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 132–141. ACM, 2011.

T. van der Storm. The rascal language workbench. Technical report, CWI Technical Report SEN-1111, CWI, 2011.

S. Viswanadha, S. Sankar, et al. Javacc - the java parser generator. `https://javacc.java.net/`, 2009. Accessed: 2015-03-22.

M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.