P2R
Implementation of Processing in Racket

Hugo Correia
hugo.f.correia@tecnico.ulisboa.pt

Instituto Superior Técnico
University of Lisbon

Abstract. Programming languages are being introduced in several areas of expertise, including design and architecture. Processing is an example of one of these languages that was created to teach architects and designers how to program. In spite of offering a wide set of features, Processing does not support the use of traditional computer-aided design applications, which are heavily used in the architecture industry.
Rosetta is a generative design tool based on the Racket language that attempts to solve this problem. Rosetta provides a novel approach to design creation, offering a set of programming languages that generate designs in different computer-aided design applications. However, Rosetta does not support Processing. Therefore, the goal is to add Processing to Rosetta's language set, offering architects that know Processing, an alternative solution that supports computer-aided design applications.
In order to achieve this goal, a source-to-source compiler that translates Processing into Racket will be developed. This will also give the Racket community the ability to use Processing in the Racket environment, and, at the same time, will allow the Processing community to take advantage of Racket's libraries and development environment.
In this report, an analysis of different language implementation mechanisms will be presented, focusing on the different steps of the compilation phase, as well as higher-level solutions, including Language Workbenches. In order to gain insight of source-to-source compiler creation, relevant existing source-to-source compilers are presented. Finally, a solution is proposed describing how this implementation will be accomplished.

1 Introduction

The evolution of technology has given mankind the ability to create more robust and efficient solutions that have changed the way problems are solved. These techniques are influencing areas like design and architecture, where artists can now produce unique constructions with the assistance of computer-aided design (CAD) applications.

Nevertheless, this process of designing through CAD applications resorts to repetitive manual interventions, that could be accomplished through automated solutions. Therefore, Generative Design (GD) is a technique that is being considered to help artists create their designs. In GD, the designer starts by producing an algorithm that computes a new sketch. Subsequently, the algorithm is used by a computational system to create a sketch and present it to the designer. A parameterized algorithm enables designers to create multiple designs, by making variations of the parameter's values. This variation provides an incremental feedback loop, which enables the use of performative architecture solutions. These approaches allow artists to test different parameter configurations, providing them a low-cost solution to obtain optimized designs. For instance, building costs, and energy efficiency constraints can be applied to obtain more advantageous designs.

On the other hand, the algorithms used in GD are specified through programming languages (PL). This has compelled architects and designers to learn programming. Consequently, PLs and integrated development environments (IDE) are being developed to fulfil the architects and designers needs. Processing (discussed in Section 1.1), is a PL that was created to teach architects and designers how to program, focusing on more creative and visual approaches.

1.1 Processing

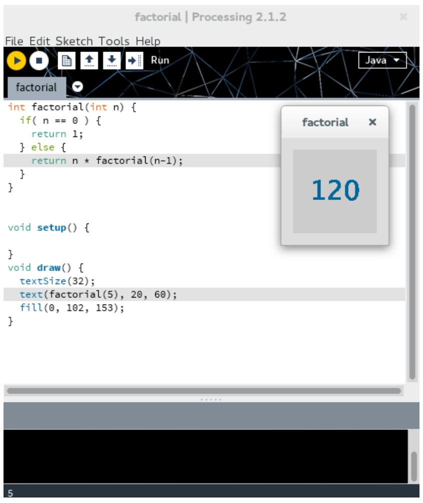
Processing[1] is a full-feature PL and IDE (Fig. 1) developed by the MIT media labs. Processing was created to teach programming to students, artists, and designers, without any previous programming experience [27]. Greatly inspired by the Java PL, it inherits its object-oriented features and C-style curly brace syntax. These features act as a stepping-stone for other languages that a novice programmer might want to learn in the future.

Processing provides different programming modes that enable the user to create new sketches in different languages and platforms. The standard mode is the Java mode, which is suited for beginners but scales up to professional Java development, with the use of tools like Eclipse IDE[2]. Alternatively, there is also an Android mode that aims to deliver an easy way of creating mobile applications. On the other hand, a separate project exists, ProcessingJS[3], that enables Processing code to be run in a web browser through HTML5 and WebGL, thus bringing a new paradigm of programming to the web.

Processing offers a simple IDE with good documentation, that is supported by a large open-source community and by a significant academic critical mass. This has been crucial for the language adoption over the years. Moreover, Processing is taught in some bachelors and masters degrees, having a positive outcome in the student's learning process [13].

Processing was created to teach programming within a visual context, using a software sketchbook of 2D and 3D designs. Currently, it includes an OpenGL renderer that designers and architects can use to have a quick overview of their designs (an example of a Processing design is presented in Fig. 2). Clearly, these designs are easier and quicker to implement, yet for more complex modelling

[1] `http://www.processing.org/`
[2] `http://www.eclipse.org/`
[3] `http://processingjs.org/`



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


void setup() {

}
void draw() {
  textSize(32);
  text(factorial(5), 20, 60);
  fill(0, 102, 153);
}
```

Fig. 1. Processing IDE drawing the result of the factorial of 5

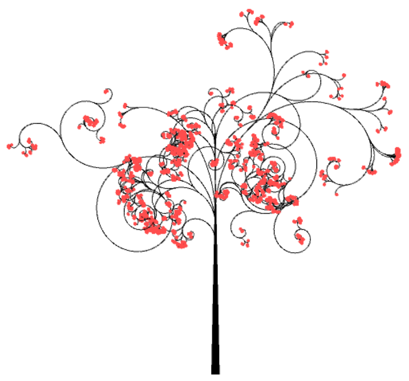


Fig. 2. Example of a design produced in Processing [2]

tasks, Processing fails to offer a viable solution. In addition, CAD applications already provide powerful modelling capabilities that allow designers and architects to interactively edit, transform, and manipulate their modelled designs. Therefore, having a CAD environment in Processing is a crucial aspect for these communities. Moreover, available CAD applications already offer some PLs to model new designs, yet these solutions are targeted for advanced users and are not suited for pedagogical purposes.

On the other hand, a more robust approach, Rosetta (presented in Section 1.2), is being developed. Instead of providing one PL with an unique rendering system, Rosetta connects multiple PLs to several CAD back-ends, where the designer can take advantage of the modelling capabilities and drawing primitives of each CAD. This enables the designer to test different PL and CAD combinations, choosing the solution that better fits the designer's needs. Therefore, the combination of Processing with Rosetta offers a pedagogical PL to Rosetta's CAD back-ends, as well as a full-featured CAD environment for Processing that architects and designers can explore.

1.2 Rosetta

Rosetta [19] is a GD tool designed to provide an alternative way of doing design through a programming-based approach. Instead of providing developers a strict methodology to create new designs, Rosetta offers a set of PL and CAD back-ends to its users, that can be combined in many ways according to their needs and personal preferences.

Rosetta is composed of front-ends, where the user can program in one of the available PLs, and back-ends, where the design is drawn and then presented to the user. Rosetta is built on Racket (discussed in Section 3.2). Fig. 3 summarizes Rosetta's architecture.

Rosetta's front-ends currently support Racket, AutoLisp and Javascript, but an additional language, Python [26], is being developed. As for its back-ends, Rosetta can render designs in AutoCAD, Rhinoceros 3D and OpenGL. In addition, Rosetta implements a large set of modelling primitives that are directly available in most CADs or that can be emulated for CADs that do not provide them.

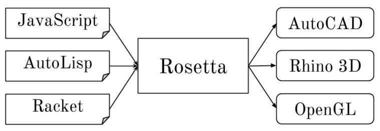


Fig. 3. Overview of Rosetta's Architecture

Although Rosetta is already a powerful option for GD, it evidently lacks the support of Processing. Therefore, it is deemed important for Rosetta to have Processing as one of its front-ends, providing another pedagogical language to Rosetta and allowing Processing users to take advantage of all the benefits of a professional CAD environment.

This report starts by describing the main goals of this work (Section 2), then presents the state-of-the-art of language creation and translation (Section 3), followed by the proposed solution (Section 4), and, finally, the evaluation method for the proposed solution is explained (Section 5).

2 Goals

The fundamental goal is to create a working implementation of Processing in Racket. This allows Processing to be added as a new Rosetta front-end, providing a new programming paradigm that users can use to implement their designs. In addition, Processing users will be able to render their designs through a CAD application, which is a main concern for the architecture community. Racket offers a wide range of libraries that would be useful to Processing users, ergo an interoperability mechanism will be developed.

Racket and Processing both offer an IDE to its users, therefore this implementation must take this into consideration. A Processing developer that will use the translator must have available a similar set of features that are already provided by the original Processing IDE, such as syntax highlighting, debugging, profiling, and OpenGL support.

Since the goal is to translate Processing into Racket, the original Processing code must be converted to semantically equivalent Racket code. Therefore, a source-to-source compiler will be developed to fulfil this need. Different available methodologies to implement language translation will be further discussed in this report.

3 Related Work

Creating a new language implementation is a complex endeavour that can be achieved through different approaches, including compilation, interpretation or language translation. Translating a high-level language such as Processing into another high-level language like Racket is most commonly designate source-to-source translation or compilation. Since the goal is to implement a source-to-source compiler, an analysis of the state-of-the-art in compilation processes must be made. The following sections will focus on how source-to-source compilers are developed, by analysing available language construction and translation methods, and by exploring proven implementations of source-to-source compilers.

3.1 Compiler Design

A compiler's architecture is composed by two distinct parts: the front-end and the back-end.

The front-end is designated as the analysis phase, where a thorough dissection is applied to the source code producing an intermediate representation (IR) that will serve as an outline of the original code [1]. The front-end is implemented by a pipeline 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 purpose, 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 machine code. This phase is commonly designated as the synthesis phase of the compiler. Generally, the back-end aggregates optimization and code generation steps. The main concern of these steps is to produce correct code, that also runs efficiently.

The following sections will present possible solutions to implement the compiler's front-end. In order 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 PL a token can be an identifier or an operator. 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 in order 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 rules' 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.

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 [1]. 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 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, thus they produce LL(K) parsers.

A common problem when writing a parser is presenting good error messages to the developer when an error occurs; as they are normally hand-built, the developer can easily create specific error messages. In addition, due to its manual implementation, these parsers can easily be optimized in some of their 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 [23].

Another issue is that theses parsers 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:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_m$$

Or indirect, when a rule expands a non-terminal symbol that in turn will expand the same rule:

$$A \rightarrow B\alpha \mid ...$$

$$B \rightarrow A\alpha \mid ...$$

In order to solve these problems, left-recursion can be removed from the grammar by re-writing it. However as many PLs employ left-recursive rules in their grammars, this ends up being a costly task. This lead users to consider more robust alternatives, such as Lex/Yacc or ANTLR, that provide a more time-saving and expressive way of analysing input and producing an IR.

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` [20].

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 [16]. 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`. Fig.4 shows how Lex and Yacc are used in conjunction.

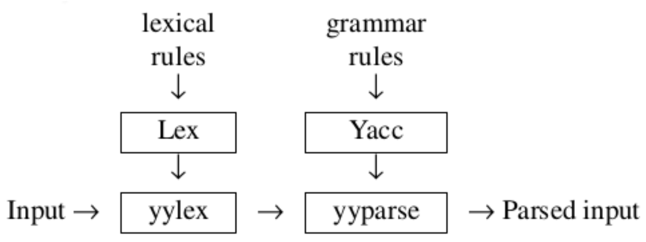


Fig. 4. Lex with Yacc [20]

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. Shift/reduce occurs when the parser cannot decide if another token should be consumed or if the rule should be immediately reduced.

A common example that illustrates this conflict is `if` statements:

```
if_stmt: "if" expr "then" stmt
       | "if" expr "then" stmt "else" stmt
       ;
```

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 design. An example is when the developer accidentally puts empty productions in grammar rules:

```
rule1: // empty                 rule2: // empty
     | expr                          | expr
     ;                               ;
```

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 [16].

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 [24].

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 (AST) and generates a human-readable recursive-descent parser [25]. 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 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.

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 [4], Coco/R [21], or JavaCC [31] are possible alternatives to implement parser generators.

Bison is a popular choice mainly due to its reuse of Yacc syntax, thus supporting any Yacc compitable 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 GLR 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 lookaheads 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 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[4]
Coco/R	LL(1)	Internal	E-BNF
Bison	LALR(1), GLR	External	Yacc
JavaCC	LL(K)	Internal	E-BNF

Table 1. Overview of the different parsing alternatives

3.2 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. Language workbench (LW), a term introduced by Martin Fowler [11], attempt to ease 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 [32] (LOP). 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).

Therefore, the goal is to use LWs to easily implement general-purpose languages (GPL) and DSLs that provide a productive infrastructure that has all the features of a modern IDE. One of the issues with available PLs is that they are not expressive enough, in the sense that, when reasoning about a problem,

[4] Extended Backus-Naur Form

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 [7]:

– **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 [12]). 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, additional features may be available in some LWs. They may support specific language validations (structure, names, and types), testing, and debugging support for the created languages.

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, DSLs are not widely used in programming, mainly due to the difficulty of combining languages and creating intelligent tools to manipulate them. LW offer the developer flexibility in building these tools, to exactly fulfil their needs.

Relevant LW systems are presented in the following sections.

MPS

Meta Programming System (MPS) is a LW created by JetBrains, greatly inspired by Charles Simonyi's Intentional Software [29]. MPS enables the user to design

DSLs and start using them straight away. It offers a simple way of working with DSLs, where even 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 [3], and the program level, that provides the 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 their 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 Languages`, are some examples of languages that belong to the set of `Language Definition` languages. For example, `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, i.e. 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 a structures in the target language. Therefore the translations can be from model-to-text or model-to-text, that enable the transformation of DSLs to other DSLs or to GPLs.

Editor Services The use of a projectional editor, enables the generation of projections for each syntax node and for 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 concepts that are well known 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. As 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 the core features of most GPLs.

Spoofax

Spoofax [17, 18] 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 plugin system, offering a large choice of plugins that can help the developer in many language development tasks such as version control or issue tracking. It 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 at the same time 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 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 an analysis that decorates the tree with semantic information to be used by the editor services. Afterwards, a normalized representation is used 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 the 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.

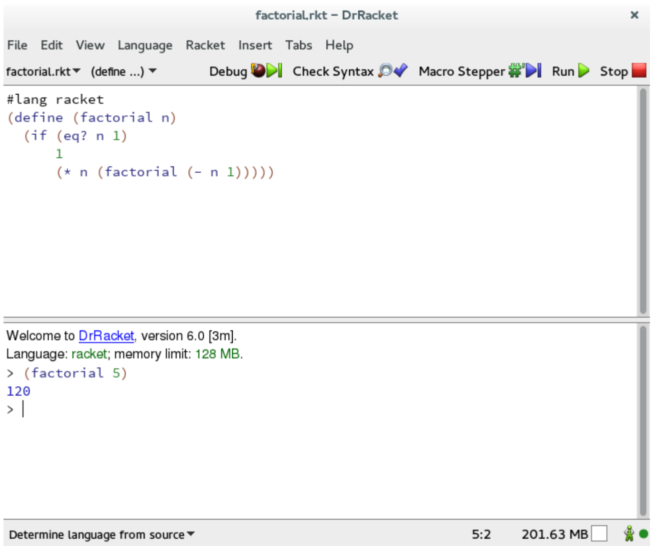
Editor Services As Spoofax is based on the Eclipse IDE, it supports features that Eclipse offers like outline view of the source, syntax highlighting, code folding, syntax completion, bracket management, error/warning marking, etc. These editor services are automatically derived from the syntax definition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

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

Racket

Racket [9] is a descendant of the Scheme language, thus belonging to 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, Fig. 5) 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


factorial.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
factorial.rkt (define ...)
Debug
Check Syntax
Macro Stepper
Run
Stop

```
#lang racket
(define (factorial n)
  (if (eq? n 1)
      1
      (* n (factorial (- n 1)))))
```

```
Welcome to DrRacket, version 6.0 [3m].
Language: racket; memory limit: 128 MB.
> (factorial 5)
120
> 
```

Determine language from source
5:2
201.63 MB

Fig. 5. DrRacket computing the factorial of 5

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 programs. So in addition to being a PL, we can view Racket as a framework for creating and extending PLs [10].

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 are ASTs that contain syntactic data as well as metadata 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.

Alternative Language Workbenches

In addition to the presented alternatives, other popular systems exist such as SugarJ [6], Rascal [30] and Xtext [5]. 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 (GLR) 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.

3.3 Source-to-Source Compilers

Source-to-source compilers, sometimes designated as transcompilers, are a type of compiler that translates a high-level language into another high-level language, while maintaining the same level of abstraction. By being able to translate a high-level PLs to other high-level languages, the developer can use 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, perform source optimizations on the existing code or for implementing DSLs. Another advantage to be considered, is that as high-level language code is being generated, it might be possible that the code can be understood and debugged by the developer.

Relevant source-to-source compilers to analyse are ProfessorJ - A Java to Scheme translator, and ProcessingJS - A Processing to JavaScript translator. The later, because of the similarity between Java and Processing, and of Scheme with Racket, the former, because of similarities between Racket and JavaScript, namely its dynamic typing and functional paradigm. These source-to-source compilers will be further explained in this Section.

ProfessorJ

ProfessorJ [14] was developed to be a language extension for Scheme's pedagogical development environment, DrScheme [8]. The aim was to provide a smoother learning curve for students that needed to learn Java. Furthermore, the teaching staff identified that the error messages that were presented were incomprehensible to the students, so they constantly required the aid of a teacher to help them decipher the errors. As a result, the teachers were constantly wasting time with specific issues that could be better used to address the underling fundamental concepts that they were trying to teach.

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.

In order to accomplish this, 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 [15]. 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.

Although some challenges arose during the development of the translator, namely inner classes and static methods, the similarities between some of Scheme's and Java's language constructs eased the development process of the translator.

However some issues needed to be solved. For instance, due to Java having many namespaces and Scheme having a single namespace, required the consideration of name-mangling techniques in order to avoid name collision of methods or shadowing of fields. 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.

In Java, native methods are supported by using C. In ProfessorJ, Scheme is used as the native language. A stub method is generated for each method in the class, which in turn will call the appropriate native function.

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 on the Java code.

ProcessingJS

ProcessingJS is a JavaScript implementation of Processing for the web created by John Resig. Through ProcessingJS, the user can create data visualizations, digital art, video, games, etc. using web standards and without plugins. Code is written in the Processing language and then included into the web page automatically by ProcessingJS. It runs in any HTML5 compatible browser such as Firefox, Safari, Chrome or Internet Explorer. It works by translating Processing code found in a web page into JavaScript and renders it using the $< canvas >$ element on the page.

ProcessingJS starts by parsing Processing code through a hand-written parser, that involves the usage of a lot of regular expressions, and at the end a JavaScript equivalent is produced. The author claims that it works "fairly well", in the sense that it is able compile code available at `processing.org`, yet it is a poor implementation of a parser and most certainly has limitations. Aside from the parser, it enables de embedding of other web technologies into Processing sketches.

In the initial implementation the author only implemented a basic 2D API, that has been further extended [28] to support 3D structures and further improvements, being released in 2010 as ProcessingJS 1.0.

4 Proposed Architecture

The proposed architecture addresses two main issues. Firstly, how Processing code is going to be translated to Racket code (Presented in Section 4.1) and secondly, how the integration with DrRacket is going to be accomplished (Discussed in Section 4.2). Additionally, the development process that will be used to build the translator is an important concern. Therefore, a summary of how the system will be developed is presented below (Section 4.3).

4.1 Processing to Racket Code

This section describes how Processing code is going to be translated into Racket code. First, an overview of the main modules that are going to be developed is explained, followed by how they are used throughout the compilation process and how interoperability with Racket is going to be achieved.

Module Architecture

A clear division is made at this point in the architecture, where two main modules exist: the `Translator` module, that will implement all the necessary mechanisms to generate compilable Racket code, and the `Runtime` module, that will have all the necessary runtime operations that the generated code will use. Fig. 6 provides an overview of the system's modules.

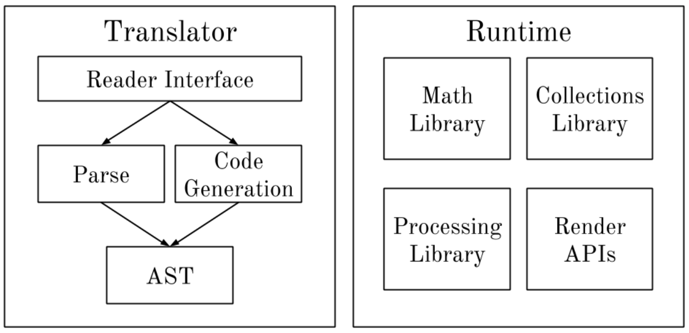


Fig. 6. Overview of the main modules of the system.

Reader Interface Module is required to provide an interface with the Racket language extension mechanism. Racket defines languages by prepending a `#lang <lang>` declaration at the beginning of each file. For instance a Processing source file would start with `#lang processing`.

In order to implement this mechanism, a new reader module must be created for Processing that requires the implementation of two main functions: `read` and `read-syntax`. The `read-syntax` function receives the name of the source file and an input-port, returning a list of syntax-objects. These syntax-objects represent the Racket code from the input-port with the additional source location information. The `read` function only receives the input-port as argument and returns a list of S-expressions. Normally, this function uses `read-syntax` to obtain the syntax-objects and then extracts each datum, i.e. plain S-expression, from it. This module uses the *Parse* and *Code Generation* modules to parse the input-port and generate Racket code.

Parse Module contains the lexer and parser definitions that are implemented using Racket's `parser-tools` [22]. This provides a lexical analyser and parser generator based on Lex and Yacc, where lexical and grammar specifications are written in plain Racket.

This module will benefit from ProfessorJ's parser and lexer definitions for the Java language. Since Processing is very similar to Java, ProfessorJ's syntactic and semantic definitions can be reused to implement Processing. Moreover, DrRacket's editor services need the original source location information (line number and column position) in order to work correctly. `parser-tools` provide a token structure that captures this information, preserving it throughout the compilation process.

In order to produce a representation of the original Processing code, the *Parse* module uses the AST module to create the IR of the source code.

An alternative parsing strategy would be to use a recursive-descent parser. However, since the grammar is of a considerable size and as ProfessorJ offers a complete Java parser, this option was abandoned in order to focus on other important areas of the translator.

AST Module is used to produce a representation of the code. The parser will generate AST nodes that represent specific concepts of the language, thus creating a tree of nodes. Each of these nodes will be implemented as a Racket object that will have all the necessary information for the generation of Racket Code.

Code Generation Module generates the final Racket code that will be fed into the Racket Virtual Machine. The translation of Processing code to semantically equivalent Racket code will be accomplished by traversing the produced ASTs, applying specific macro transformation to each AST node. For specific cases where macros do not suffice, the translation will be accomplished through the use of Racket procedures. Finally, the Racket Virtual Machine(VM) loads the generated code, offering performance optimizations, automatic memory management, and a just-in-time compiler.

Runtime Module includes the necessary runtime functions that are present in Processing. For instance, Processing's Collection and Math libraries must be made available to the programmers.

Compilation Process

In order to compile Processing to Racket, a set of implementation options must to be made. Fig. 7 provides an overview of the compilation process.

First, the Racket code is read by a lexical analyser that produces a stream of tokens, that is then consumed by a LALR(1) parser producing a list of ASTs. Afterwards, equivalent Racket code is going to be generated by traversing the AST. In the end, Racket code is ready to be processed by Racket's VM.

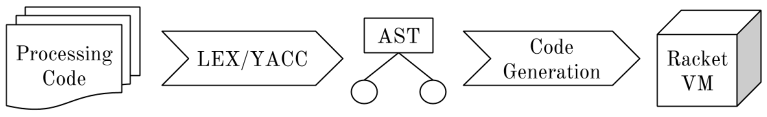


Fig. 7. Overview of the compilation process.

Interoperability

A mechanism for achieving interoperability will be developed, where Racket modules can be used inside Processing code. The implementation effort of this feature is mitigated because Racket supports the composition of different language modules. The solution is to include Racket modules through the use of Processing's `import` keyword, creating a convention that would clearly distinguish the Racket module that is being imported from the normal Processing modules. In Racket one of the following import cases can occur:

– A Racket module that is already installed in the system can be imported. To solve this, the following syntax would be used:

```
import Racket <module> [ as <alias> ];
```

The `<module>` is the name of the installed module. The optional `<alias>` field can be used to avoid name collision. Therefore, to use a `function` defined in an imported module the developer would use `<module>.function`.

– A module can be imported from a file, using a specific path:

```
import Racket "path/to/module" as <alias>;
```

Thus a string will contain the path to the module. As a result the `<alias>` would be mandatory.

– External PLaneT[5] modules can be used:

```
import Planet "maintainer/planet-module:version";
```

In this case, to call a `function` from the `planet-module`, the following syntax will be used: `planet-module.function`.

Integration with Rosetta

Processing's drawing paradigm closely resemble OpenGL's, thus these features could be implemented through the use of Racket's OpenGL APIs. However, Rosetta provides multiple CAD back-ends which include OpenGL. Architects that already know Processing will have the ability to easily change the back-end that is being used, offering additional flexibility to their design process.

On the other hand, Processing does not support some primitives that are available in Rosetta, thus this implementation would add a new set of drawing primitives to the Processing's tool set.

[5] `http://planet.racket-lang.org/`

4.2 IDE development

One of the main goals of this work is to provide a similar development environment in Racket that is already available to Processing developers. Table 2 provides a comparison between both systems. Racket offers additional editor services to the developer, which is a positive factor that justifies the development of these services for Processing in the Racket environment.

GUI (Graphical User Interface) support is essential for this implementation. Processing renders its designs through OpenGL, so in order to support this in Racket, there are two available options: Racket's OpenGL APIs or Rosetta. Racket's OpenGL APIs would have the advantage that the Processing community could directly use the Racket environment to visualize designs without using Rosetta.

On the other hand, as Rosetta provides support to CAD back-ends, as well as an OpengGL back-end, the designer and architecture communities can take advantage of CAD modelling capabilities, that are lacking in Processing.

Both of these solutions could be supported, yet as Rosetta provides the best of both worlds, this solution will resort to Rosetta to visualize its designs.

Debugger is a fundamental feature for programmers to find errors in their implementations. An external mode is available for debugging inside the Processing environment but has not been integrated into its core. Racket supports an internal library that provides debugging for its programs that uses generated syntax-objects to reference the original Processing code.

Profiler support is provided in both environments. However, 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.

Syntax Checking provides mechanisms to check if the program is written using correct syntax. Racket uses the S-expressions as the underlying representation for these tools, so little work is needed to support this mechanism.

Syntax Highlighting is present in both environments. Racket provides a module to define how the language's syntax highlighting will be presented in DrRacket.

Read-eval-print-loop (REPL) is a common feature that is present in the Lisp family. It provides users a mechanism to test specific parts of their code and it is a good way for novices in the language to quickly test their ideas and learn from their mistakes. However, this mechanism is not available in Processing, hence it would be good for Processing users to be able to use a REPL to quickly create and test their designs.

[6] External debug mode - `http://debug.martinleopold.com/`
[7] Using JVM profiler tools.
[8] Using Racket's OpenGL & GUI libraries.

	GUI	Debugger	Profiler	Syntax Checking	Syntax Highlighting	REPL
Processing	Yes	Yes[6]	Yes[7]	No	Yes	No
Racket	Yes[8]	Yes	Yes	Yes	Yes	Yes

Table 2. Comparison between Processing's and Racket's IDEs

4.3 Development Process

Some implementation concerns have to be taken into consideration. The goal is to gradually implement more features of the language in the compiler. To achieve this, each new feature must be tested before implementing the next feature, ergo the aim is to have tests that confirm the robustness of the translator.

In addition, the use of a version control system, such as GitHub[9], is important because it provides a set of tools such as documentation tracking. Afterwards, the intention is to share this work with the Racket and open-source communities. Using GitHub to manage the development process will help the dissemination of the translator and offers an opportunity to receive external contributions from the community. The planning of the implementation of the system is available in Appendix A.

5 Evaluation Methodology

The proposed evaluation for this work will be the formal correctness of the translator's generated code, that is, once given a Processing program, the translator must produce the same result as it would with the original Processing compiler. In addition, a Processing user must have development environment similar to the original Processing IDE.

Performance will not be a major concern because translating high-level languages is, in general, a slow process. Furthermore, as the source language runs on the JVM, that applies sophisticated optimizations, and as Processing core concern is not performance, this will not be a focus of this work's evaluation.

Therefore, the evaluation will focus on the formal correctness of the translator, the similarity of the produced designs, and the designer's effort to produce the same designs in both systems.

6 Conclusion

The demonstration of the pertinence of this system has been presented throughout this report. Creating an implementation of Processing for the Racket community, that allows architects and designers to render their designs in CAD applications, is a solution that will greatly benefit both worlds. The ability of

[9] `http://www.github.com`

providing new design paradigms to the developers and IDE tools that are supplied by the DrRacket environment, offers a strong reason for the architecture community to take advantage of this solution.

The implementation of the compiler follows the traditional pipeline approach, composed by syntactic, semantic, and code generation phases, only differing in the generated target language that, in this case, is a high-level language. Processing code will be transformed into Racket using these compilation steps. In the end, the generated Racket code will be loaded into Racket's VM. After an analysis of the common solutions used in language implementation, the choice of Racket for the development is due to the necessity of integration with Rosetta, as well as its language composition mechanism and editor services.

In the end, the goal is to have Processing available in Racket, offering a solution with all of DrRacket's features, as well as the ability to create new designs using Rosetta's CAD back-ends.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Anastasis Chasandras. My tree. `http://www.openprocessing.org/sketch/10318`, May 2014.

[3] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[4] Charles Donnelly and Richard Stallman. Bison. the yacc-compatible parser generator. 2013.

[5] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[6] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

[7] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. *Software Language Engineering*, pages 197–217, 2013.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme.

[9] M Flatt and RB Findler. Plt. the racket guide. `http://docs.racket-lang.org/reference/`, May 2014.

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

[11] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

[12] Martin Fowler. Projectional editing. `http://martinfowler.com/bliki/ProjectionalEditing.html`, May 2014.

[13] Pia Fricker, Christoph Wartmann, and Ludger Hovestadt. Processing: Programming instead of drawing. *Proceedings of eCAADe 2008*, pages 525–530, 2008.

[14] Kathryn E Gray and Matthew Flatt. Professorj: a gradual introduction to java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177. ACM, 2003.

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

[16] Stephen C Johnson. Yacc: Yet another compiler-compiler. 1975.

[17] Karl Trygve Kalleberg and Eelco Visser. Spoofax: An extensible, interactive development environment for program transformation with stratego/xt. 2007.

[18] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.

[19] António Leitão, Luís Santos, and José Lopes. Programming languages for generative design: a comparative study. *International Journal of Architectural Computing*, 10(1):139–162, 2012.

[20] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator. 1975.

[21] Hanspeter Moessenboeck. Coco/r: A generator for fast compiler front-ends. 1990.

[22] Scott Owens. Parser tools: lex and yacc-style. `http://docs.racket-lang.org/parser-tools/`, May 2014.

[23] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf, 1st edition, 2009.

[24] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

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

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

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

[28] Andor Salga, Daniel Hodgin, Anna Sobiepanek, Scott Downe, Mickael Medel, and Catherine Leung. Processing.js: Sketching with $< canvas >$. In *ACM SIGGRAPH 2011 Talks*, page 15. ACM, 2011.

[29] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

[30] Tijs van der Storm. The rascal language workbench. Technical report, CWI Technical Report SEN-1111, CWI, 2011.

[31] Sreeni Viswanadha, Sriram Sankar, et al. Javacc - the java parser generator. `https://javacc.java.net/`, May 2014.

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

A Appendix: Planning

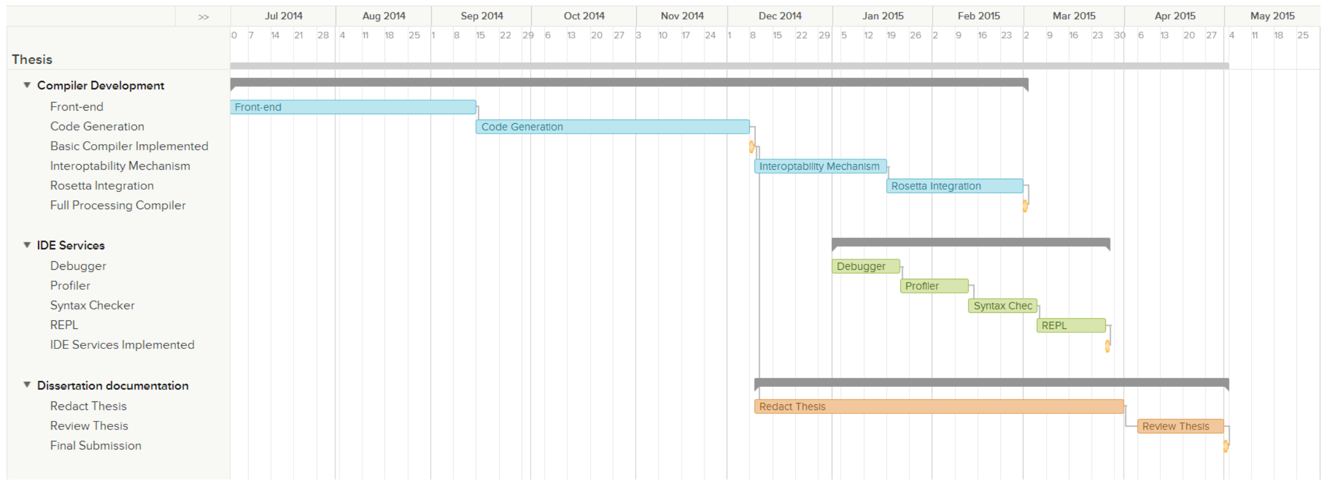


Fig. 8. Planned Schedule for the Dissertation