Naylang

A REPL interpreter and debugger for the Grace programming language.

Borja Lorente

Director: José Luis Sierra Rodríguez

Abstract

A REPL interpreter and debugger for the Grace programming language.

Keywords: Intepreters,Programming Languages,Debuggers,Grace .

Contents

List of Figures

1. Introduction

Naylang is an open source REPL interpreter, runtime and debugger for the Grace programming language written in modern C++. It currently implements a subset of Grace described later, but as both the language and the interpreter evolves the project strives for feature-completeness.

Motivation

Grace is a language aimed to help novice programmers get acquainted with the process of programming, and as such it provides safety and flexibility in it's design.

However, this flexibility comes at a cost, and most of the current implementations of Grace are in themselves opaque and obscure. Since Grace is open source, most of it's implementations are also open source, but this lack of clarity in the implementation makes them hard to extend and modify by third parties and newcomers, severely damaging the growth opportunities of the language.

Objectives and Methodology

Naylang strives to be an exercise in interpreter construction not only for the creators, but also for any possible contributor. Therefore, the project focuses on the following goals:

- To provide a solid implementation of a relevant subset of the Grace language.
- To be as approachable as possible by both end users, namely first-time programmers, and project collaborators.
- To be itself a teaching tool to learn about one possible implementation of a language as flexible as Grace.

To that end, the project follows a Test Driven Development approach [], in which unit tests are written in parallel to or before the code, in very short iterations. This is the best approach for two reasons:

Firstly, it provides an easy way to verify which part of the code is working at all times, since tests strive for complete code coverage []. Therefore, newcomers to the project will know where exactly their changes affect the software as a whole, which will allow them to make changes with more confidence.

Secondly, the tests themselves provide documentation that is always up-to-date and synchronized with the code. This, coupled with descriptive test names, provide a myriad of **working code examples**. Needless to say that this would result in vital insight gained at a much quicker pace by a student wanting to learn about interpreters.

Tradeoffs

Since Naylang is designed as a learning exercise, clarity of code and good software engineering practices will take precedence over performance in almost every case. More precisely, if there is a simple and robust yet naïve implementation of a part of the system, that will be selected instead of the more efficient one.

However, good software engineering practices demand that the architecture of the software has to be modular and loosely coupled. This, in addition to the test coverage mentioned earlier, will make the system extensible enough for anyone interested to modify the project to add a more efficient implementation of any of it's parts.

In short, the project optimizes for **approachability** and **extensibility**, not for **execution time** or **memory usage**.

2. The Grace Programming Language

Introduction

Grace is an open source educational programming language, aimed to help the novice programmer understand the base concepts of Computer Science and Software Engineering. To that aim, Grace is designed to provide an intuitive and extremely flexible syntax while maintaining the standards of commercial-grade programming languages.

Key Features

Support for multiple teaching paradigms

Different teaching entities have different curricula when teaching novices. For instance, one institution might prefer to start with a declarative approach and focus on teaching students the basics of functional programming, while another one might want to start with a more imperative approach.

Despite being imperative at it's core, Grace provides sufficient tools to teach any curriculum, since methods are intuitively named and can be easily composed. In addition to that, lambda calculus is embedded in the language, with every block being a lambda function and accept arguments.

Safety and flexibility

Similar to other approachable high-level languages such as Python or JavaScript, Grace is garbage-collected, so that the novice programmer does not have to worry about manually managing object lifetimes. Furthermore, Grace has no mechanisms to directly manipulate memory, which provides a safe environment for beginners to learn.

Gradual typing

Grace is gradually typed, which means that the programmer may choose the degree of static typing that is to be performed. This flexibility is atomic at the statement level, and therefore any declaration may or may not be typed. For instance, we might have all of the following in the same code:

```
var x := 5           // x is inferred to be a Number, a native type of Grace
var y : Number := 6    // y is declared as a Number, a native type of Grace
var z : Rational := 7.0 // z is declared as a Rational, a user-defined type
                             // which may or may not inherit from Number
```

This mechanism brings to instructors the tools to teach types at the beginning of a course, leave them until the end, or explain them at the moment they deem appropriate.

However, this mechanism is not within the scope of the project, and for the moment Naylang will only have a dynamic typing mechanism, similar to JavaScript.

Multi-part method signatures

Method signatures have a few particularities in Grace. Firstly, a method signature can have multiple parts. A part is a Unicode string followed by a parameter list. That way, methods with much more intuitive names can be formed:

```
method substringFrom(first)to(last) {
   // Return a substring of the caller object from index "first" to index "last"
}
"hello".substringFrom(2)to(4) // Would return "llo"
```

This way there is a more direct correlation between the mental model of the student and the code.

To differentiate between methods, Grace uses the arity of each of the parts to construct a *canonical name* for the method. A canonical name is not more than the concatenation of each of the parts, substituting the parameter names for underscores. That way, the canonical name of the method above would be `substringFrom(_)to(_)`.

Two methods are different if and only if their canonical names are different. For example, `substringFrom(_)to(_)` is different from `substringFromto(_,_)`. As it is obvious, this mechanism imposes a differentiation by arity, and not by parameter types. Therefore, we could have this situation:

```
method substringFrom(first : Rational)to(last : Rational) {
    // Code
}

method substringFrom(first : Integer)to(last : Integer) {
    // Code
}
```

In this case, the second method is considered to be the same as the first, and it will cause a *shadowing error* for conflicting names. This design decision stems directly from the gradual typing, since there is no way to discern objects that are dynamically typed, and any object may be dynamically typed at any point. As a side effect, this method makes request dispatch considerably simpler.

Lexically scoped, single namespace

Grace has a single namespace for convenience, since novice projects will rarely be so large that they require separation of namespaces. It is also lexically scoped, so the declarations in a block are accessible to that scope and every scope inside it, but not to any outer scopes.

Lineups

Collections in Grace are represented as Lineups, which are completely polymorphic lists of objects that implement the Iterable interface.

Object-based inheritance

Everything in Grace is an object. Therefore, the inheritance model is more based on extending existing objects instead of instantiating particular classes. In fact, classes in Grace are no more than factory methods that return an object with a predefined set of methods and fields.

Unfortunately, this mechanism is also out of the scope of the project and will be left for future releases.

Subset of Grace in a Page

As mentioned earlier, some features of the language will be left out of the interpreter for now, and therefore we must define the subset of the language that Naylang will be able to interpret. Following is an excerpt from the official documentation (Noble, 2014), which provides examples of the features of the language covered:

// TODO: Write subsection

3. State of the art

Kernan

Kernan is currently the most feature-complete implementation of Grace (**???**). It is an interpreter written entirely in C#, and it features some similar execution and AST models as those implemented in Naylang. Specifically, the method dispatch and execution flow takes heavy inspiration from Kernan.

Kernan is publicly available from the Grace website (**???**).

Minigrace

Minigrace is the original Grace compiler (Black et al., 2012), which is written in Grace itself via bootstrapping with C (Grace, 2017). It does not include all the current language features, but it still serves as an excellent industrial-grade test case for the language.

Minigrace is currently hosted int GitHub (Homer, 2017).

GDB

The GNU Project Debugger has been the de facto debugger for C and C++ for many years, and thus it merits some time to study it. The main influence of GDB in Naylang will be the design of it's command set, that is, the commands if offers to the user. In particular, Naylang will focus on reproducing the functionality of the following commands: `run`, `continue`, `next`, `step`, `break`, `print` (**???**). Naylang will add another command, `env`, that allows the user to print the current evaluation scope. This set of core commands is simple yet highly usable, and can be composed to form virtually any behavior desired by the user. Support for commands such as `finish` and `list` will be added as future work.

To offer a controlled and pausable execution of a program, GDB reads the executable metada, and executes it pausing in the desired locations set by user-specified breakpoints. Nince Naylang is an intepreter and thus doesn't generate an executable, this information gathering technique is of course unusable by the project. Instead, Naylang will gather information from the AST directly to control the debugging flow.

Modularity in Interpreters

4. Implementation

The implementation of Naylang follows that of a completely interpreted language. First, the source is tokenized and parsed with ANTLRv4. Then, a visitor traverses the parse tree and generates and Abstract Syntax Tree from the nodes, annotating each one with useful information such as line numbers when necessary. Lastly, an evaluator visitor traverses the AST and executes each of the nodes.

In addition to the REPL commands, Naylang includes a debug mode, which allows to debug a file with the usual commands (run, continue, step in, step over, break). The mechanisms necessary for controlling the execution flow are embedded in the evaluator, as is explained later.

//TODO Class diagram [] Project Structure ——

The project is structured as a standard CMake multitarget project. The root folder contains a `CMakeLists.txt` file detailing the two targets for the project: The interpreter itself, and the automated test suite. Both folders have a similar structure, and contain the `.cpp` and `.h` files for the project. Other folders provide several necessary tools and aids for the project:

```
.(root)
  |-- cmake       // CMake modules for the ANTLRv4 C++ target
  |-- dists       // Build script for GCC
  |-- examples    // Examples of Grace Code to test the interpreter
  |-- grammars    // ANTLRv4 grammar files for the Lexer and Parser
  |-- interpreter // Sources to build the Naylang executable
  |-- tests       // Automated test suite
  '-- thirdparty
      '-- antlr   // ANTLRv4 Generator tool and runtime
```

Sources

The sources folder, `interpreter`, contains the sources necessary to build the Naylang executable. The directory is structured as a standalone CMake project, with a `CMakeLists.txt` file and a `src` directory at it's root. Inside the `src` directory, the project is separated into `core` and `frontends`. Currently only the console frontend is implemented, but this separation will allow for future development of other frontends, such as graphical interfaces. The `core` folder is structured as follows:

```
./interpreter/src/core/
|-- control // Controllers for the evaluator traversals
|-- model
```

```
|   |-- ast // Definitions of the AST nodes
|   |   |-- control
|   |   |-- declarations
|   |   '-- expressions
|   |       |-- primitives
|   |       '-- requests
|   |-- evaluators // Classes that implement traversals of the AST
|   '-- execution // Classes that describe various runtime components
|       |-- methods
|       '-- objects
'-- parser // Extension of the ANTLRv4-generated parser
```

Tests

For automated testing, the Catch header-only library was used (Nash, 2014). The interior structure of the `tests` directory **directly mirrors** that of `interpreter`, and the test file for each class is suffixed with `_test`. Thus, the test file for `./interpreter/src/core/parser/NaylangParserVisitor` will be found in `./tests/src/core/parser/NaylangParserVisitor_test.cpp`. Each file has one or more `TEST_CASE()`s, each with some number of `SECTION()`s. Sections allow for local shared and local initialization of objects.

Grammars and examples

There are two Grace-specific folders in the project:

- `grammars` contains the ANTLRv4 grammars necessary to build the project and generate `NaylangParserVisitor`. The grammar files have the `.g4` extension.
- `examples` contains short code snippets written in the Grace language and used as integration tests for the interpreter and debugger.

Build tools

Lastly, the remaining folders contain various aides for compilation and execution:

- `cmake` contains the CMake file bundled with the C++ target, which drives the compilation and linking of the ANTLR runtime. It has been slightly modified to compile a local copy instead of a remote one (Lorente, 2017).
- `thirdparty/antlr` contains two major components:
- A frozen copy of the ANTLRv4 runtime in the 4.7 version , `antlr-4.7-complete.jar` (Parr, 2017), to be compiled and linked against.
- The ANTLRv4 tool, `antlr-4.7-complete.jar`, which is executed by a macro in the CMake file described earlier to generate the parser and lexer

classes. Obviously, this is also in the 4.7 version of ANTLR. Visitor-based Evaluation ——

Before discussing the parsing, the shape of the Abstract Syntax Tree and the implementation of objects, it is necessary to outline the general flow of execution of Naylang.

At it's core, Naylang is designed to be an visitor-based interpreter (**???**). This means that the nodes of the AST are only containers of information, and every processing of the tree is done outside it by a Visitor class. This way, we can decouple the information about the nodes from the actual processing of the information, with the added benefit of being able to define arbitrary traversals of the tree for different tasks. Thess visitors are called *evaluators*, and they derive from the base class `Evaluator`. `Evaluator` has an empty virtual method for each type of AST node, and each AST node has an `accept()` method that accepts an evaluator. As can be seen, a subclass of `Evaluator` might include rules to process one or more of the node types simply by overriding the default empty implementation.

The main evaluator in Naylang is ExecutionEvaluator, and it is in charge of traversing the tree and executing the program defined in it, to provide an output. It has several noteworthy parts:

- **The scope** is what determines which fields and methods are accessible at a given time. It is a `GraceObject`, as will be discussed later, and the evaluator features several methods to modify it.
- **The partial** is the means of communicating between function calls. Any objects created as a result of interpreting a node (e.g. a `GraceNumber` created by a `NumberLiteral` node) are placed here, to be cosumed by the caller method.

Lexing and Parsing

This step of the process was performed with the ANTLRv4 tool (Parr, 2013), specifically the C++ target (Harwell, 2016). ANTLRv4 generates several lexer and parser classes for the specified grammar which contain methods that are executed every time a rule is activated.

These classes can then be extended to override those rule methods and execute arbitrary code, as will be shown later. This method allows instantiation of the AST independently from the grammar specification.

The Naylang Parser Visitor

For this particular program, the Visitor lexer and parser were chosen, since ANTLRv4's default implementation allowed for a preorder traversal of the parse tree, but offered enough flexibility to manually modify the traversal if needed. One might, for example, prefer to visit the right side of the assignment before moving onto the left side to instantiate particular types of assignment depending on the assigned value. To that end, the `NaylangParserVisitor.cpp` class was created, which extends `GraceParserBaseVisitor`, a class designed to provide the default implementation of a parse tree traversal.

The class definition along with the overriden method list can be found in `interpreter/src/core/parser/NaylangParserVisitor.h`. Note that ANTLRv4 names the visitor methods `visit<RuleName>` by convention. For example, `visitBlock()` will be called when the `block` rule is matched in parsing.

The Naylang Parser Stack

During the AST construction process, information must be passed between parser function calls. A function call parser rule must have information about each of the parameters available, for example. To that end, the parser methods generated by ANTLR have a return value of type `antlrcpp::Any`. This however was not usable by the project, since sometimes more than one value needed to be returned, and most of all, converting from `Any` to the correct node types proved impossible.

Therefore, a special data structure was developed to pass information between function calls. The requirements were:

- It must hold references to Statement nodes.
- It must be able to return the n last inserted Statement pointers, in order of insertion.
- It must be able to return those references as either Statements, Expressions or Declarations, the three abstract types of AST nodes that the parser handles.

The resulting structure declaration can be found in `interpreter/src/core/parser/NaylangPa`
It uses template metaprogramming to be able to specify the desired return type from the caller and cast the extracted elements to the right type. Note that a faulty conversion is possible and the structure does not enforce any type invariants other than those statically enforced by the compiler. Therefore, the invariants must be implicitly be preserved by the client class.

The parser class uses wrapper functions for convenience to predefine the most common operations of this structure. For example:

```
// NaylangParserVisitor.h
std::vector<StatementPtr> popPartialStats(int length);

// NaylangParserVisitor.cpp
std::vector<StatementPtr> NaylangParserVisitor
        ::popPartialStats(int length) {
    return _partials.pop<Statement>(length);
}
```

Left-Recursion and Operator Precedence

Grace assigns a three levels of precedence for operators: `*` and `/` have the highest precedence, followed by `+` and '-', and then the rest of prefix and infix operators along with user methods are executed.

Usually, for an EBNF-like (Standard, 1996) grammar language to correctly assign operator precedence, auxiliary rules must be defined which clutter the grammar with unnecessary information. ANTLRv4, however, can handle left-recursive rules as long as they are not indirect (Parr, 2013). It does this by assigning rule precedence based on the position of the alternative in the rule definition. This way, defining operator precedence becomes trivial:

```
// Using left-recursion and implicit rule precendence.
expr  : expr (MUL | DIV) expr
      | expr (PLUS | MINUS) expr
      | explicitRequest
      | implicitRequest
      | prefix_op expr
      | expr infix_op expr
      | value
      ;
```

As can be seen, the precedence is clearly defined and expressed where it matters the most (the first two lines). Grace's specification does not define a precedence for any other type of expression, so the rest is left to the implementer.

A slightly more annotated version of this rule can be found in the parser grammar, under the `expression` rule.

Abstract Syntax Tree

As an intermediate representation of the language, a series of classes has been developed to denote the different aspects of the abstract syntax. Note that even though the resulting number of classes is rather small, the iterative process necessary to arrive to the following hierarchy took many iterations, due to the sparse specification of the language semantics (Grace, 2016) and the close ties this language has with the execution model. This created a loop where design decisions in the execution model required changes in the AST representation, and vice versa. The following diagram represents the current class hierarchy:

// TODO: add class diagram []

The design of the abstract syntax representation hierarchy is subject to change as new features are implemented in the interpreter.

GraceAST class

// TODO: Complete when merged with NodeFactory and removed the nodelink crap

Pointers

In the representation of the different parts of the abstract syntax, often a node has to reference other nodes in the tree. Since that memory management of tree nodes was not clear at the beginning of the project, a series of aliases were created to denote pointers to the different major classes of nodes available. These aliases are named `<Nodeclass>Ptr` (e.g. `ExpressionPtr`). For the current representation of the language, only three classes need these pointers specified: Statement, Declaration and Expression. These three classes of pointers give the perfect balance of specificity and generality to be able to express the necessary constructs in Grace. For instance, a variable declaration might want an ExpressionPtr as it's value field, while a method declaration might want DeclarationPtrs for it's formal parameters and high-level StatementPtrs for it's body.

Currently, the aliases are implemented as reference-counted pointers (`std::shared_ptr<>` (**???**)). However, as the project has moved towards a centralized tree manager (`GraceAST`), the possibility of making that clas responsible for the memory of the nodes has arised. This would permit the aliases to switch to weak pointers (**???**) or even raw pointers in their representation, probably reducing memory management overhead.

Statement Nodes

The Statement nodes are at the top of the hierarchy, defining common traits for all other nodes, such as source code coordinates. Control structures, such as IfThen and While, are the closest to pure statements that there is. It could be said that Return is the purest of statements, since it does not hold any extra information.

Declaration Nodes

The declaration nodes are nodes that do not return a value, and bind a specific value to an identifier. Therefore, all nodes must have a way of retrieving their names so that the fields can be created in the corresponding objects. We must distinguish between two types of declarations: **Field Declarations**, and **Method Declarations**.

Field Declarations

Field declarations represent the intent of mapping an identifier to a value in the current scope. Depending in the desired mutablity of the expression, these declarations will be represented with either Constant Declarations or Variable Declarations. These two nodes only differ in their evaluation, and their internal representation is identical. They both need an identifier to create the desired field, and optionally an initial value to give to that field.

```
class VariableDeclaration : public Declaration {

    std::string _identifier;
    ExpressionPtr _initialValue;

public:

    VariableDeclaration(
        const std::string &identifier,
        ExpressionPtr intialValue,
        int line, int col);

    VariableDeclaration(
        const std::string &identifier,
        int line, int col);

    // Accessors and accept()
};
```

Every Field Declaration is a **breakable statement** (see Debugging).

Method Declarations

Method declarations represent a subroutine inside a grace Object. While their evaluation might be complex, the abstract representation of a method is rather straightforward. Sintactically, a method is comprised of a canonical identifier (**??**), a list of formal parameter definitions (to be later instantiated in the method scope) and a list of statements that comprises the body of the method.

```
class MethodDeclaration : public Declaration {

    std::string _name;
    std::vector<DeclarationPtr> _params;
    std::vector<StatementPtr> _body;

public:
    MethodDeclaration(
            const std::string &name,
            const std::vector<DeclarationPtr> &params,
            const std::vector<StatementPtr> &body,
            int line, int col);

    // Accessors and accept()
};
```

Expression Nodes

Control Nodes

Control nodes represent the control structures a user might want to utilize in order to establish the execution flow of the program. Nodes like conditionals, loops and return statements all belong here. Note that, due to the high modularity of Grace, only the most atomic nodes have to be included to make the language Turing-complete, and every other type of control structure (for loops, for instance) can be implemented in a prelude, in a manner transparent to the user (**???**) (**???**).

Conditional Nodes These nodes form the basis of control flow, and are what makes the foundation of the language. This class includes the IfThen and IfThenElse node definitions:

```
class IfThenElse : public Statement {

    ExpressionPtr _condition;
    std::vector<StatementPtr> _then;
    std::vector<StatementPtr> _else;

public:

    IfThenElse(
            ExpressionPtr condition,
            std::vector<StatementPtr> thenExp,
            std::vector<StatementPtr> elseExp,
            int line, int col);

    // Accessors and accept()
};
```

Both nodes have a similar structure, with an expression node as the condition, and blocks of statements to be executed if the condition is met.

// TODO: Move to evaluation

When evaluating a conditional node, the condition node is evaluated first. Then, if the condition returns `true`, the `then` statements are evaluated. If it is not met, the `else` statements will be evaluated if there are any (IfThenElse nodes), otherwise nothing will be done (IfThen nodes).

Loop Nodes

Loop nodes are the nodes used to execute an action repeated times. In this case, only one node type is necessary, the While node.

```
class While : public Statement {

    ExpressionPtr _condition;
    std::vector<StatementPtr> _body;

public:
    While(
        ExpressionPtr condition,
        const std::vector<StatementPtr> &body,
        int line, int col);

    // Accessors and accept()
};
```

While loops accept a boolean expression as a condition and a list of statements as a body.

Return Nodes

Return is the most basic control structure, and serves to express the desire of terminating the execution of the current method and optionally return a value from it. As such, the only information they hold is the value to be returned.

```
class Return : public Statement {

    ExpressionPtr _value;

public:

    // Explicit value return
    Return(
        ExpressionPtr value,
        int line, int col);

    // Implicit value return
    Return(int line, int col);

    // Accessors and accept()
};
```

Assigment

Assignments are a special case node. Since, as will be explained later, objects are maps from identifiers to other objects, the easiest way of performing an assignment is to modify the parent's scope. That is, to assign value A to field X of scope Y (`Y.X := A`) the easiest way is to modify Y so that the X identifier is now mapped to A. Note that a user might omit identifier Y (`X := A`), in which case the scope is implicitly set to `self` (the current scope). Therefore, writing `X := A` is syntactically equivalent to writing `self.X := A`.

The ramifications of this decission are clear. A special case must be defined both in the parser and in the abstract syntax, to allow the retrieval of the field name and optionally the scope in which that field resides:

```
class Assignment : public Statement {
public:
  // Explicit scope constructor
  Assignment(
    const std::string &field,
    ExpressionPtr scope,
    ExpressionPtr value);

  // Implicit scope constructor
  Assignment(
    const std::string &field,
    ExpressionPtr value);

  // Accessors and accept()
};
```

Expressions

Expression nodes are nodes that, when evaluated, must return a value. This includes many of the usual constructs such as primitives (BooleanLiteral, NumberLiteral...), ObjectConstructors and Block constructors. However, it also includes some unusual classes called `Requests`.

Primitives

Primitives are the expressions that, when evaluated, must return objects in the a base type of the language. In general, a primitive node is only responsible for holding the information necessary to build an object of it's type, and they correspond directly with native type constructors. For instance, a NumberLiteral node will only need to hold it's numeric value, which is all that's necessary to create a GraceNumber object. Of course, this makes the evaluation of these nodes straightforward, and they will always be leaves of the AST. As an example, this is the defininiton of the primitive node used for strings.

```
class StringLiteral : public Expression {

    std::string _value;

public:

    StringLiteral(
        const std::string &value,
        int line, int col);

    // Accessors and accept()
};
```

The list of primitives includes: NumberLiteral, StringLiteral, BooleanLiteral and Lineup.

Requests

In Grace everything is an object, and therefore every operation, from variable references to method calls, has a common interface: A Request made to an object. Syntactically, it is impossible to differentiate a parameterless method call from a field request, and therefore that has to be resolved in the interpreter and not the parser. Hence, we need a representation wide enough to incorporate all sorts of requests, with any expression as parameters.

```
class RequestNode : public Expression {
protected:
    std::string _name;
    std::vector<ExpressionPtr> _params;

public:

    // Request with parameters
    RequestNode(
        const std::string &methodName,
        const std::vector<ExpressionPtr> &params,
        int line, int col);

    // Parameterless request (can be a field request)
    RequestNode(
        const std::string &methodName,
        int line, int col);

    // Accessors and accept()
};
```

There are two types of Requests:

Implicit Requests are Requests made to the current scope, that is, they have no explicit receiver. These requests are incredibly flexible, and they accept almost any parameter. The only necessary parameter is the name of the method or field requested, so that the evaluator can look up the correct object in the corresponding scope. Optional parameters include a list of expressions for the parameters passed to a request (in case it's a method request), and code coordinates.

```
class ImplicitRequestNode : public RequestNode {

public:
    // Constructors inherited from superclass

    ImplicitRequestNode(
        const std::string &methodName,
        const std::vector<ExpressionPtr> &params,
        int line, int col);

    ImplicitRequestNode(
        const std::string &methodName,
        int line, int col);

    // Accessors and accept()
};
```

Explicit Requests are Requests made to a specified receiver, such as invoking a method of an object. These Requests are little more than a syntactic convenience, since they are composed of two Implicit Requests (one for the receiver, one for the actual request).

```
class ExplicitRequestNode : public RequestNode {

    ExpressionPtr _receiver;

public:

    // Constructors call the super() constructor.

    ExplicitRequestNode(
        const std::string &method,
        ExpressionPtr receiver,
        const std::vector<ExpressionPtr> &params,
        int line, int col);

    ExplicitRequestNode(
        const std::string &method,
        ExpressionPtr receiver,
        int line, int col);

    // Accessors and accept()
};
```

Following are some examples of different code snippets, and how they will be translated into nested Requests (for brevity, IR and ER will be used to denote ImplicitRequest and ExplicitRequest, respectively):

```
x;              // IR("x")
obj.val;        // ER(IR("obj"), "val"))
add(4)to(3);    // IR("add(_)to(_)", {4, 3})
4 + 3;          // ER(4, "+(_)", 3)
```

Note that, even in the case of an expression not returning anything, it will always return the special object `Done` by default.

ObjectConstructor Nodes

In Grace (similarly to JavaScript), a user can at any point explicitly create an object with the `object`keyword, followed by the desired contents of the object. this operation is represented in the abstract syntax with an ObjectConstructor node, which evaluates to a user-defined GraceObject.

Since an object can contain virtually any Grace construct, and ObjectConstructor is nothing more than a list of statements that will be evaluated one after the other.

```
class ObjectConstructor : public Expression {

    std::vector<StatementPtr> _statements;

public:
    ObjectConstructor(
        const std::vector<StatementPtr> &statements,
        int line, int col);

    // Accessors and accept()
};
```

Block Nodes

Blocks are a very particular language feature in Grace. Block expressions create block objects, but also define lambda expressions. Therefore, from the representation's point of view, a block must hold information very similar to that of a method declaration, with formal parameters and a body.

```
class Block : public Expression {

    std::vector<StatementPtr> _body;
    std::vector<DeclarationPtr> _params;

public:

    Block(
        std::vector<StatementPtr> _body,
        std::vector<DeclarationPtr> _params,
        int line, int col);

    // Accessors and accept()
};
```

Methods and Dispatch

One of the advantages of Grace is that it integrates native methods and user-defined methods seamlessly in it's syntax. As a consequence, the implementation must be able to handle both types of methods indistinctly from each other. Hence, the `Method` class was created. This class represents a container for everything that is needed to define a Grace method, namely, a list of **formal parameters** in the form of **declarations**, and a list of **statements** that conforms the **body** of the method. The canonical name of a method is used in determining which of an object's methods to use, and not in the execution of the method itself. Hence, it is not necessary to include it in the representation. Since Grace blocks are lambda expressions, it is also possible to instantiate a `Method` from a `Block`:

```
class Method {
  std::vector<DeclarationPtr> _params;
  std::vector<StatementPtr> _code;
public:
  Method(BlockPtr code);
  Method(const std::vector<DeclarationPtr> &params, const std::vector<Statemen
  // ...
};
```

Dispatch

Since every method has to belong to an object, the best way to implement dispatch is to have objects dispatch their own methods. Since user-defined methods contain their code in the AST representation, an object needs an evaluator to evaluate the code, and thus it must be passed as a parameter. In addition, the **effective parameter** values must be precalculated and passed as Grace object, not AST nodes:

```
virtual GraceObjectPtr dispatch(
  const std::string &methodName,
  ExecutionEvaluator &eval,
  const std::vector<GraceObjectPtr> &paramValues);
```

The object then retrieves the correct `Method`, forms a `MethodRequest` with the parameters, and calls `respond()` on the desired method, returning the value if applicable.

Self-evaluation

The only responsibility of `Methods` is to be able to `respond()` to requests made by objects. A `MethodRequest` is in charge of holding the **effective parameters** for that particular method call.

```
virtual GraceObjectPtr respond(
  ExecutionEvaluator &context,
  GraceObject &self,
  MethodRequest &request);
```

How this method is implemented is up to each subclass of `Method`. Native methods, for example, will contain C++ code that emulates the desired behavior of the subprogram. `Method` counts with a default implementation of `respond()`, which is used for user-defined methods, and uses the given context to evaluate every line of the method body:

```
GraceObjectPtr Method::respond(
  ExecutionEvaluator &context, GraceObject &self, MethodRequest &request) {

    // Create the scope where the parameters are to be instantiated
    GraceObjectPtr closure = make_obj<GraceClosure>();

    // Instantiate every parameter in the closure
    for (int i = 0; i < request.params().size(); i++) {
        closure->setField(params()[i]->name(), request.params()[i]);
    }

    // Set the closure as the new scope, with the old scope as a parent
    GraceObjectPtr oldScope = context.currentScope();
    context.setScope(closure);

    // Evaluate every node of the method body
    for (auto node : _code) {
        node->accept(context);
    }

    // Get return value (if any)
    GraceObjectPtr ret = context.partial();
    if (ret == closure) {
        // The return value hasen't changed. Return Done.
        ret = make_obj<GraceDoneDef>();
    }

    // Restore the old scope
    context.setScope(oldScope);
    return ret;
}
```

Native methods

Native methods are a special case of `Methods` in that they are implemented using native C++ code. Most of these operations correspond to the operations necessary to handle native types (such as the `+` operator for numbers). Native methods do not require a context to be evaluated, and therefore they define a simpler interface for the subclasses to use, for conveniance.

```
class NativeMethod : public Method {
public:
  // Pure abstract method to be implemented by subclasses
  virtual GraceObjectPtr respond(
    GraceObject &self, MethodRequest &request) = 0;

  // Note that subclasses can still override this implementation
  virtual GraceObjectPtr respond(
    ExecutionEvaluator &context, GraceObject &self, MethodRequest &request) {
    return respond(self, request);
  }
};
```

Each native method is a subclass of `NativeMethod`, and implements it's functionality in the body of the overriden `respond()` method. For convenience, each subclass of GraceObject that implements native types defines them inside it's header, as inner classes. This is specially useful when a method requires access to the internal structure of an object, since inner classes have access to them by default. Object and Execution Model ——

In Grace, everything is an object, and therefore the implementation of these must be flexible enough to allow for both JavaScript-like objects and native types such as booleans, numbers and strings.

GraceObject

For the implementation, a generic `GraceObject` class was created, which defined how the fields and methods of objects were implemented:

```
class GraceObject {
protected:
    std::map<std::string, MethodPtr> _nativeMethods;
    std::map<std::string, MethodPtr> _userMethods;
    std::map<std::string, GraceObjectPtr> _fields;

    GraceObjectPtr _outer;

public:
  // ...
};
```

As can be seen, an object is no more than maps of fields and methods. Since every **field** (object contained in another object) has a unique string identifier, and methods can be differentiated by their canonical name (**??**), a plain C++ string is sufficient to serve as index for the lookup tables of the objects.

`GraceObject` also provides some useful methods to modify and access these maps:

```
class GraceObject {
public:
    // Field accessor and modifier
    virtual bool hasField(const std::string &name) const;
    virtual void setField(const std::string &name, GraceObjectPtr value);
    virtual GraceObjectPtr getField(const std::string &name);

    // Method accessor and modifier
    virtual bool hasMethod(const std::string &name) const;
    virtual void addMethod(const std::string &name, MethodPtr method);
    virtual MethodPtr getMethod(const std::string &name);

    // ...
};
```

Native types

Grace has several native types: `String`, `Number`, `Boolean`, `Iterable` and `Done`. Each of these is implemented in a subclass of `GraceObject`, and if necessary stores the corresponding value. For instance:

```
class GraceBoolean : public GraceObject {
    bool _value;
public:
    GraceBoolean(bool value);
    bool value() const;

    // ...
};
```

Each of these types has a set of native methods associated with it (such as the `+(_)` operator for numbers), and those methods have to be instantiated at initialization. Therefore, `GraceObject` defines an abstract method `addDefaultMethods()` to be used by the subclasses when adding their own native methods. For example, this would be the implementation for Number:

```
void GraceNumber::addDefaultMethods() {
    _nativeMethods["prefix!"] = make_native<Negative>();
    _nativeMethods["==(_)"] = make_native<Equals>();
    // ...
    _nativeMethods["^(_)"] = make_native<Pow>();
    _nativeMethods["asString(_)"] = make_native<AsString>();
}
```

There are some other native types, most of them used in the implementation and invisible to the user, but they have no methods and only one element in their type class, such as `Undefined`, which throws an error whenever the user tries to interact with it.

Casting

Since this subset of Grace is dynamically typed, object casting has to be resolved at runtime. Therefore, `GraceObjects` must have the possibility of casting themselves into other types. Namely, we want the possiblity to, for any given object, retrieve it as a native type at runtime. This is accomplished via virtual methods in the base class, **which error by default**:

```
// GraceObject.h

// Each of these methods will throw a type exception called
virtual const GraceBoolean &asBoolean() const;
virtual const GraceNumber &asNumber() const;
virtual const GraceString &asString() const;
// ...
```

These functions are then overriden with a valid implementation in the subclasses that can return the appropriate value. For example, `GraceNumber` will provide an implementation for `asNumber()` so that when the evaluation expects a number from a generic object, it can be given. Of course, for types with just **one possible member in their classes** (such as `Done`) and objects that **do not need more data** than the base `GraceObject` provides (such as `UserObject`), no caster method is needed, and a boolean type checker method is sufficient. These methods return false in `GraceObject`, and are overriden to return true in the appropriate classes:

```
// GraceObject.h

// These methods return false by default
virtual bool isNumber() const;
virtual bool isClosure() const;
virtual bool isBlock() const;
// ...
```

This approach has two major benefits:

- It allows the evaluator to treat every object equally, except where a specific cast is necessary, such as the result of evaluating condition expression of an `if` statement, which must be a `GraceBoolean`. Therefore, the type checking is completely detached from the AST and, to an extent, the evaluator. The evaluator only has to worry about types when the language invarints require so.
- It scales very well. For instance, if a new native type arised that could be either a boolean or a number, it would be sufficient to implement both caster

methods in an appropriate subclass.

Note that this model is used for runtime dynamic typing and, since Grace is a gradually-typed language, some of the type-checking work will have to be moved the the AST as the possibility of proper static typing is implemented.

Heap and Garbage Collection

Debugging

Frontend

One of the design goals of Naylang is to serve as a teaching example of an interpreter. This requires that the execution core (parsing, AST and evaluation) be as isolated as possible from the interaction with the user, with aims to facilitate the student to discern the essential parts of interpreters from the nonessential I/O operations.

Currently, all the user interaction is handled by the `ConsoleFrontend` class, which is in charge of receiving commands from the user and calling one of it's `ExecutionMode`s to handle the commands.

Execution modes (such as REPL or Debug) are in charge of feeding data to and controlling the flow of the interpreters. Each mode has it's own commands, which are implemented using the Command pattern. It can be easily seen how any one of these pieces can be easily swapped, and seemingly relevant changes such as adding a graphical frontend are as simple as replacing `ConsoleFrontend`.

Here is the list of available commands in Naylang:

```
// Global commands (can be called from anywhere)
>>>> debug <file>
  // Start debugging a file
>>>> repl
  // Start REPL mode
>>>> quit
  // Exit Naylang

// REPL mode
>>>> load (l) <filepath>
  // Open the file, parse and execute the contents
>>>> exec (e) <code>
  // Execute an arbitrary code in the current environment
>>>> print (p) <expr>
  // Execute an expression and print the result,
  // without modifying the environment.
// Debug mode
ndb> break (b) <line>
  // Place a breakpoint in a given line
ndb> run (r)
  // Start execution from the beginning of the file
ndb> continue (c)
  // Resume execution until end of file or a breakpoint is reached
ndb> env (e)
  // Print the current environment
ndb> step (st)
  // Step to the next instruction, entering new scopes
ndb> skip (sk)
  // Step to the next instruction, skipping scope changes and calls
```

// TODO: UML diagram

5. Modular Visitor Pattern

6. Testing Methodology

Testing and automated validation were important parts of the development of Naylang. Even though Grace had a complete specification, some of the general design approaches were not clear from the beginning, as is mentioned in the discussion about the Abstract Syntax Tree. Therefore, there was a high probability that some or all parts of the system would have to be redesigned, which was what in fact ended up occurring. To mitigate the risk of these changes, the decision was made to have automatic unit testing with all the parts of the system that could be subject to change, so as to receive exact feedback about which parts of the system were affected by any change.

This decision has in fact proven to be of great value in the later stages of the project, since it makes a thousand-line project manageable.

Tests as an educational resource

Naylang aims to be more than just a Grace interpreter, but to also be an approachable FOSS (**???**) project for both potential collaborators and programming language students the same. Having a sufficiently big automated test suite is vital to make the project amiable to newcomers, for the following reasons:

- Automated tests provide **complete, synchronized documentation** of the system. Unlike written documentation or comments, automated tests do not get outdated and, if they are sufficiently atomic and well-named, provide working **specification and examples** of what a part of the system does and is supposed to be used. A newcomer to the project will find it very useful to dive into the test suite even before looking at the implementation code to find up-to-date explainations of a module and it's dependencies.
- Automated tests force the implementer to **modularize**. Unit testing requires that the dependencies of the project be minimized, so as to make testing each part individually as easy as possible. Therefore, TDD encourages a very decoupled design, which makes it easy to reason about each part separately (**???**).
- Automated tests make it **easy to make changes**. When a student or potential collaborator is planning to make changes, it can be daunting to modify any of the existing source code, in fear of a functionality regression. Automated tests aid with that, and encourage the programmer to make changes by reassuring the sense that any undesired changes in functionality will be immediately reported, and the amount of hidden bugs created will be minimal.

As an example, if newcomers wanted learn about how Naylang handles Assignment, they can just dive into the `Assignment_test.cpp` file to see how the Assignment class is initialized, or search for usages of the Assignment class in the `ExecutionEvaluator_test.cpp` file to see how it's consumed and evaluated, or even search it in `NaylangParserVisitor_test.cpp` to see how it's parsed. Then, if they wanted to extend Assignment to enforce some type checking, they could write their own test cases and add them to the aforementioned files, which would guide them in the parts of the system that have to be modified to add that capability, and notify them when they break some functionality.

Test-Driven Development (TDD)

Since the goal was to cover as much code as possible with test cases, the industry-standard practice of Test-Driven Development was used. According to TDD, for each new addition to the codebase, a failing test case must be added first. Then, enough code is written to pass the test case. Lastly, the code is refactored to meet coding standards, all the while keeping all the tests passing. This way, every part of code

TDD may feel slow at first, but as the end of the project approached the critical parts of the project were covered in test cases, which provided with immense agility to develop extraneous features such as the frontends.

As a result of following the TDD discipline, the length of the test code is very similar to that of the implementation code, a common occurrence in projects following this practice (**???**).

The Framework

Naylang is a relatively small (less than 10.000 lines of code), threadless and lightweight project. Therefore, the testing framework choice was influenced mainly in favor of ease-of-use, instead of other features such as robustness or efficiency. With that end in mind, Catch (Nash, 2014) presented itself as the perfect choice for the task, for the following reasons:

- **Catch is header only**, and therefore including it in the build system and Continuous Integration was as trivial as adding the header file to every test file.
- **Catch allows for test suites**, by providing two levels of separation (`TEST_CASE()` and `SECTION()`). This way, the test file for a particular component of the system (e.g. `GraceNumber_test.cpp`) usually contains a single `TEST_CASE()` comprised of several `SECTION()`s. That way, it's easy to identify the exact point of failure of a test. Some of the bigger files have more than one test case, where required (e.g. `NaylangParserVisitor_test.cpp`).

- **Allows for exception-assertions** (named `REQUEST_THROWS()` and `REQUEST_THROWS_WITH()`), in addition to regular truthy assertions (named `REQUEST()`). For a language interpreter, many of the runtime errors occur when the language user inputs an invalid statement, and therefore are out of the hands of the implementor. It is imperative to provide graceful error handling to as many of these faults as possible, and therefore it is also necessary to test them. This exception-assertions provide the tools to test the runtime errors correctly.
- **Test cases are debuggable**, meaning that, since all Catch constructs are macros, the content of test cases themselves is easily debuggable with most industrial-grade debuggers, namely GDB. The project takes advantage of this feature by writing a failing test case every time a bug is found by manual testing. This way **as many debug passes as needed** can be done **without having to reproduce the bug** by hand each time, which considerably reduces debugging time.

Testing the Abstract Syntax Tree

The Abstract Syntax Tree was the first thing implemented, and thus it was the component where most of the up-front decisions about the testing methodology were made. Luckily, the nodes themselves are little more than information containers, and thus their testing is straightforward, with most of the test files following a similar pattern. A typical test file for a node contains a single test case with the name of the node, and several sections divided in two categories:

- **Constructor tests** provide examples and descriptions of what data a node expects to receive and in what order.
- **Accessor tests** indicating what data can be accessed of each node type, and how.

Following is one of the more complicated examples:

```
TEST_CASE("ImplicitRequestNode Expressions", "[Requests]") {

    // Initialization common to all sections
    auto five = make_node<NumberLiteral>(5.0);
    auto xDecl = make_node<VariableDeclaration>("x");

    // Constructor sections
    SECTION("A ImplicitRequestNode has a target identifier name, parameter exp
        REQUIRE_NOTHROW(ImplicitRequestNode req("myMethod", {five}););
    }

    SECTION("An ImplicitRequestNode with an empty parameter list can request v
        REQUIRE_NOTHROW(ImplicitRequestNode variableReq("x"););
        REQUIRE_NOTHROW(ImplicitRequestNode methodReq("myMethod"););
    }
    // Accessor sections
    SECTION("A ImplicitRequestNode can return the identifier name and paramete
        ImplicitRequestNode req("myMethod", {five});
        REQUIRE(req.identifier() == "myMethod");
        REQUIRE(req.params() == std::vector<ExpressionPtr>{five});
    }
}
```

As mentioned above, the nodes do not have any internal logic to speak of, and are little more than data objects (**??**). Therefore, these two types of tests are sufficient.

Testing the Evaluation

Testing the Parser

Integration testing

Testing Frontends

7. Conclusions

Having reached the end of the development cycle for th

Challenges

Goal review

Future work

Bibliography

Black, A.P., Bruce, K.B., Homer, M. and Noble, J. (2012), "Grace: The absence of (inessential) difficulty", in *Proceedings of the Acm International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, New York, NY, USA, pp. 85–98.

Grace. (2016), "Grace language specification", available at: http://gracelang.org/documents/grace-spec-0.7.0.html (accessed 2 May 2017).

Grace. (2017), "Minigrace webpage", available at: http://gracelang.org/applications/grace-versions/minigrace/ (accessed 5 May 2017).

Harwell, S. (2016), "ANLR4-c++", available at: http://www.soft-gems.net/index.php/tools/49-the-antlr4-c-target-is-here (accessed 2 May 2017).

Homer, M. (2017), "Minigrace source code", available at: https://github.com/gracelang/minigrace (accessed 5 May 2017).

Lorente, B. (2017), "Antlr4 for c++ with cmake: A practical example", available at: http://blorente.me//Antlr-,-C++-and-CMake-Wait-what.html (accessed 2 May 2017).

Nash, P. (2014), "A modern, c++-native, header-only, framework for unit-tests, tdd and bdd c++ automated test cases in headers", available at: https://github.com/philsquared/Catch (accessed 2 May 2017).

Noble, J. (2014), "//grace in one page", available at: http://gracelang.org/applications/documentation/grace-in-one-page/ (accessed 2 May 2017).

Parr, T. (2013), *The Definitive Antlr 4 Reference*, 2nd ed., Pragmatic Bookshelf.

Parr, T. (2017), "4.7", available at: https://github.com/antlr/antlr4/tree/c8d9749be101aa24947aebc706ba8ee8300e84ae (accessed 2 May 2017).

Standard, E.S.S. (1996), "Ebnf: Iso/iec 14977: 1996 (e)", *URL Http://Www. Cl. Cam. Ac. Uk/Mgk25/Iso-14977. Pdf*, Vol. 70.

A. Appendix A: Grammars

ANTLR 4 grammars used for parsing Grace in Naylang.

Lexer Grammar

```
lexer grammar GraceLexer;
tokens {
    DUMMY
}

WS : [ \r\t\n]+ -> skip ;
INT: Digit+;
Digit: [0-9];

METHOD: 'method ';
VAR_ASSIGN: ':=';
VAR: 'var ';
DEF: 'def ';
PREFIX: 'prefix';
OBJECT: 'object';

COMMA: ',';
DOT: '.';
DELIMITER: ';';
QUOTE: '"';
EXCLAMATION: '!';
RIGHT_ARROW: '->';
OPEN_PAREN: '(';
CLOSE_PAREN: ')';
OPEN_BRACE: '{';
CLOSE_BRACE: '}';
OPEN_BRACKET: '[';
CLOSE_BRACKET: ']';

CONCAT: '++';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
```

```
  POW: '^';
  EQUAL: '=';

  TRUE: 'true';
  FALSE: 'false';

 // Should be defined last, so that reserved words stay reserved
  ID: LETTER (LETTER | '0'..'9')*;
  fragment LETTER : [a-zA-Z\u0080-\uFFFF];
```

Parser Grammar

```
  parser grammar GraceParser;

  options {
      tokenVocab = GraceLexer;
  }

  /*
   * Parser Rules
   */
  program: (statement)*;
  statement: expression DELIMITER | declaration; //| control;

  declaration : variableDeclaration
              | constantDeclaration
              | methodDeclaration
              ;

variableDeclaration: VAR identifier (VAR_ASSIGN expression)? DELIMITER;
 constantDeclaration: DEF identifier EQUAL expression DELIMITER;
  methodDeclaration: prefixMethod
                   | userMethod
                   ;

  prefixMethod: METHOD PREFIX (EXCLAMATION | MINUS)  methodBody;
  userMethod: METHOD methodSignature methodBody;

  methodSignature: methodSignaturePart+;
methodSignaturePart: identifier (OPEN_PAREN formalParameterList CLOSE_PAREN)
 formalParameterList: formalParameter (COMMA formalParameter)*;
  formalParameter: identifier;

  methodBody: OPEN_BRACE methodBodyLine* CLOSE_BRACE;
methodBodyLine: variableDeclaration | constantDeclaration | expression DELIM
```

```
// Using left-recursion and implicit operator precendence. ANTLR 4 Reference,
expression  : rec=expression op=(MUL | DIV) param=expression      #MulDivExp
        | rec=expression op=(PLUS | MINUS) param=expression    #AddSubExp
        | explicitRequest                                  #ExplicitReqExp
        | implicitRequest                                  #ImplicitReqExp
        | prefix_op rec=expression                          #PrefixExp
        | rec=expression infix_op param=expression          #InficExp
          | value                                           #ValueExp
              ;

explicitRequest : rec=implicitRequest DOT req=implicitRequest #ImplReqExplRe
           | rec=value DOT req=implicitRequest        #ValueExplReq
                 ;

  implicitRequest : multipartRequest               #MethImplReq
           | identifier effectiveParameter #OneParamImplReq // e.g. `print "H
           | identifier              #IdentifierImplReq //variables or 0 para
                 ;
  multipartRequest: methodRequestPart+;
methodRequestPart: methodIdentifier OPEN_PAREN effectiveParameterList? CLOSE
effectiveParameterList: effectiveParameter (COMMA effectiveParameter)*;
  effectiveParameter: expression;
  methodIdentifier: infix_op | identifier | prefix_op;

  value   : objectConstructor #ObjConstructorVal
          | block             #BlockVal
          | lineup            #LineupVal
          | primitive         #PrimitiveValue
          ;

 objectConstructor: OBJECT OPEN_BRACE (statement)* CLOSE_BRACE;
block: OPEN_BRACE (params=formalParameterList RIGHT_ARROW)? body=methodBodyL
  lineup: OPEN_BRACKET lineupContents? CLOSE_BRACKET;
  lineupContents: expression (COMMA expression)*;

  primitive   : number
              | boolean
              | string
              ;

  identifier: ID;
  number: INT;
  boolean: TRUE | FALSE;
  string: QUOTE content=.*? QUOTE;
  prefix_op: MINUS | EXCLAMATION;
  infix_op: MOD | POW | CONCAT;
```

B. Appendix B: How was this document made?

Author

Note: the process described in this Appendix was devised by Álvaro Bermejo, who published it under the MIT license in 2017 [@persimmon].

Process

This document was written on Markdown, and converted to PDF using Pandoc.

Document is written on Pandoc's extended Markdown, and can be broken amongst different files. Images are inserted with regular Markdown syntax for images. A YAML file with metadata information is passed to pandoc, containing things such as Author, Title, font, etc… The use of this information depends on what output we are creating and the template/reference we are using.

Diagrams

Diagrams are were created with LaTeX packages such as tikz or pgfgantt, they can be inserted directly as PDF, but if we desire to output to formats other than LaTeX is more convenient to convert them to .png files with tools such as `pdftoppm`.

References

References are handled by pandoc-citeproc, we can write our bibliography in a myriad of different formats: bibTeX, bibLaTeX, JSON, YAML, etc…, then we reference in our markdown, and that reference works for multiple formats