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.

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.

Execution flow

Before discussing the parsing, the shape of the Abstract Syntax Tree and the implementation of objects, it is necessary to outline the general execution flow 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. These 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` may 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`, with `DebugEvaluator` extending the functionality by providing the necessary mechanisms for debugging. The implementation of the evaluation has been designed to be extensible and modular by default, which is described in Modularity.

// TODO: Add diagram

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.

To pass data between methods, the Naylang Parser Visitor utilizes two stacks. The first one stores partial AST nodes that are created as a result of parsing lower branches of the syntax tree, and are then added to the parent node (e.g. the initial value expression node in a variable declaration). A full description of this structure is found in a following section. The second stack stores raw strings, and is used in the construction of proper canonical names and identifiers for methods and fields.

Parsing Strategy

The strategy followed for parsing the source code was to override only the necessary methods to traverse the tree confortably. In general, for a node that depends on child nodes (such as an Assignment), the child nodes were visited and instatiated before constructing the parent node, as opposed as constructing an empty parent node and adding fields to it as the children were traversed. This approach has two major advantages:

- It corresponds with a postorder traversal of the implicit parse tree, which is more akin to most traditional parsing algorithms.

- As will be seen, it simplifies the design of AST nodes, since it eliminates the need to have mutation operators and transforms them into Data Objects (**???**).

Prefix and Infix Operators

Prefix and infix operators are a special case of syntactic sugar in Grace, since they allow for the familiar infix and prefix syntax (e.g. `4 + 5`). It is necessary to process these operators as special cases of the syntax, to convert them to valid AST nodes. As the Grace specification says (**???**), infix and prefix operators must be converted to explicit requests on an object.

In the case of **prefix operators**, the operation must be transformed to an explicit request in the right-hand receiver. In addition to that, the name of the method to call must be preceded with the `prefix` keyword. For instance, a call to the logical not operator `!x` would be transformed into the explicit request `x.prefix!`. As can be seen, a prefix operator does not take parameters.

For **infix operators** the transformation is similar, but in this case the receiver is the leftmost operand, while the right-side operand is passed in as a parameter. In addition, the canonical name of the method must be formed by adding one parameter to the method name, to account for the right-side operand. Therefore, the aforementioned `4 + 5` request would be translated to `4.+(5)`, or an explicit request for the `+(_)` method of the `4` object with `5` as a parameter.

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);
}
```

An example of the stack usage can be found in parsing user-defined methods, since these require Statement nodes for the body and Declarations for the formal parameters.

```
antlrcpp::Any NaylangParserVisitor::
    visitUserMethod(GraceParser::UserMethodContext *ctx)
    {
    // Parse the signature.
    // After this line, both the node stack and the string stack
    // contain the information regarding the formal parameter nodes
    // and the canonical name, respectively.
    ctx->methodSignature()->accept(this);

    // For the method's canonical name by joining each of the parts
    std::string methodName = "";
    for (auto identPart :
        popPartialStrs(
            ctx->methodSignature()->methodSignaturePart().size())) {
        methodName += identPart;
    }

    // Retrieve the formal parameters from the node stack
    int numParams = 0;
    for (auto part : ctx->methodSignature()->methodSignaturePart()) {
        numParams +=
            part->formalParameterList()->formalParameter().size();
    }
    auto formalParams = popPartialDecls(numParams);

    // Parse the method body
    ctx->methodBody()->accept(this);
    int bodyLength = ctx->methodBody()->methodBodyLine().size();
    auto body = popPartialStats(bodyLength);
    for (auto node : body) {
        notifyBreakable(node);
```

```
    }

    // Create the method node
    auto methodDeclaration =
        make_node<MethodDeclaration>(
            methodName, formalParams, body, getLine(ctx), getCol(ctx));

    // Push the new node into the stack as a declaration
    // for the caller method to consume
    pushPartialDecl(methodDeclaration);
    return 0;
}
```

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.

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.

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()
};
```

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()
};
```

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()
};
```

Execution Evaluator

The Execution Evaluator (or EE) is one of the most crucial components of Naylang. It is it's responsibility to traverse the AST created by the parser and interpret each node's meaning, executing the commands necessary to simulate the desired program's behavior. In a sense, it could be said that the Execution Evaluator is the engine of the interpreter.

As previously described, the Execution Evaluator (as do all other subclasses of Evaluator) follows the Visitor pattern to encapsulate the processing associated with each node. This particular subclass overrides every node processing, since each one has some sematics associated with it.

Structure

An important part of the EE is the mechanism used to share information between function calls. For instance, there has to be a way for the evaluator to access the number created after traversing a NumberLiteral node. For that, the EE has two mechanisms:

- **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 scope can be modified and interchanged depending on the needs of the programs. For example, executing a method requires creating a subscope that contains variables local to the method, and discarding it after it is no longer needed.
- **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. For instance, when evaluating an `Assignment` the evaluator needs access to the object generated by evaluating the value node. The phrases "return" and "place in the partial" are used interchangeably in the rest of the section.

```
class ExecutionEvaluator : public Evaluator {

    GraceObjectPtr _partial;
    GraceObjectPtr _currentScope;

public:

    ExecutionEvaluator();

    virtual void evaluate(BooleanLiteral &expression) override;
    virtual void evaluate(NumberLiteral &expression) override;
    virtual void evaluate(StringLiteral &expression) override;
    virtual void evaluate(ImplicitRequestNode &expression) override;
    virtual void evaluate(ExplicitRequestNode &expression) override;
```

```
    virtual void evaluate(MethodDeclaration &expression) override;
    virtual void evaluate(ConstantDeclaration &expression) override;
    virtual void evaluate(Return &expression) override;
    virtual void evaluate(Block &expression) override;
    virtual void evaluate(ObjectConstructor &expression) override;
    virtual void evaluate(VariableDeclaration &expression) override;

    // Accessors and mutators
};
```

Evaluations

The following section details how each node class is evaluated. This categorization closely resembles that of the AST description, since the structure of the syntax tree strongly conditions the structure of the evaluator.

Expressions

In Naylang's abstract syntax, expressions are nodes that return a value. In terms of the evaluation, this translates to expressions being nodes that, when evaluated, place an object in the partial. This object can be new (e.g. when evaluating a primitive) or it can be a reference (e.g. when evaluating a field request). Note that method requests are also in this category, since in Grace every method returns a value (`Done` by default).

Primitives

The primitive expressions are the easiest to evaluate, since they are always leaves of the syntax tree and correspond directly to classes in the object model. Therefore, evaluating a primitive expression requires no more than creating a new object of the correct type and placing it in the partial, as shown in the example.

```
void ExecutionEvaluator::evaluate(NumberLiteral &expression) {
    _partial = make_obj<GraceNumber>(expression.value());
}
```

ObjectConstructor Nodes

The evaluation of Object Constructor nodes requires some additional setup by the evaluator. The final objective is to have **a new object** in the partial, with the field and method values specified in the constructor. Since an Object Constructor node is a list of valid Grace statement nodes, the easiest way to ensure that the new object has the correct contents is to evaluate each statement inside the constructor sequentially.

However, if no previous work is done, the results of those evaluations would be stored in the current scope of the evaluator, and not in the new object. Therefore, we must ensure that when evaluating the contents of the constructor, we are doing so in the scope of the new object. Therefore, the following algorithm has been used to evaluate the Object Constructor nodes:

```
void ExecutionEvaluator::evaluate(ObjectConstructor &expression) {
    // Store the current scope to restore it later
    GraceObjectPtr oldScope = _currentScope;

    // Create the target object and set it as the current scope
    _currentScope = make_obj<UserObject>();

    // Evaluate every statement in the constructor in the context
    // of the new object
    for (auto node : expression.statements()) {
        node->accept(*this);
    }

    // Place the result on the partial
    _partial = _currentScope;

    // Restore the previous scope
    _currentScope = oldScope;
}
```

Implicit Requests

Implcit Request are the most complex nodes to evaluate, since they can represent a number of intents. Said nodes can be either field requests or method calls (with or without parameters), and thus the evaluation has to include several checks to determine it's behavior.

However, Grace provides a useful invariant to design the evaluation of requests: All identifiers are unique within a scope or it's outer scopes (**???**). As a consequence, for any given object, the sets of field and method identifiers **have to be disjoint**. Therefore, it does not make a difference the order in which we check whether a request is a field request or method call. In the case of Naylang, a decision was made to check whether a request was a field request first, and default to interpreting it as a method request if it wasn't.

Once a request is found to represent a **field request**, it's evaluation becomes simple. Requests are expressions, and thus must place a value in the partial. Implicit requests are requests made to the current scope, and thus it is sufficient to retrieve the value of the field in the current scope.

Evaluating a **method call** requires slightly more processing. First, the values of the effective parameters must be computed by evaluating their expression nodes. These values are then stored in a list that will ultimately be passed to the method

object. After that, a request has to be made to the current scope to `dispatch()` the method named in the request, and the return value is stored in the partial. The dispatch and method evaluation mechanism is further discussed in Methods and Dispatch.

```
void ExecutionEvaluator::evaluate(ImplicitRequestNode &expression) {

    // Evaluate the node as a field request if possible
    if (expression.params().size() == 0) {
        if (_currentScope->hasField(expression.identifier())) {
            _partial = _currentScope->getField(expression.identifier());
            return;
        }
    }

    // Otherwise, evaluate it as a method call
    std::vector<GraceObjectPtr> paramValues;
    for (int i = 0; i < expression.params().size(); i++) {
        expression.params()[i]->accept(*this);
        paramValues.push_back(_partial);
    }

    _partial = _currentScope->dispatch(expression.identifier(), *this, paramVa
}
```

Explicit Requests

Explicit Requests are similar to Implicit Requests, the only difference being that Explicit Requests can make requests to scopes other than the current one. An additional step must be added to compute the effective scope of the request (which was always `self`in the case of Implicit Requests). Then, the requests will be done to the newly retrieved object instead of the current scope.

```
void ExecutionEvaluator::evaluate(ExplicitRequestNode &expression) {
    expression.receiver()->accept(*this);
    auto receiver = _partial;

    // Note the use of receiver instead of _currentScope

    if (expression.params().size() == 0) {
        if (receiver->hasField(expression.identifier())) {
            _partial = receiver->getField(expression.identifier());
            return;
        }
    }

    std::vector<GraceObjectPtr> paramValues;
    for (auto param : expression.params()) {
        param->accept(*this);
```

```
        paramValues.push_back(_partial);
    }
    _partial = receiver->dispatch(expression.identifier(), *this, paramValues)
}
```

This evaluation contains duplicate code that could certainly be refactorized, but it was left as-is in benefit of clarity by providing completely independent evaluation functions.

Block Nodes

Block nodes are similar to Object Constructor nodes in that they place a new object with effectively arbitrary content in the partial. The only difference is that, while Object Constructor nodes immediately evaluate every one of the statements, a Block node is inherently a lambda method definition, and thus the body of the method cannot be evaluated until all the effective parameters are known.

Therefore, the evaluation of a Block in grace consists of forming an anonymous method with the contents of the Block node and creating a `GraceBlock` object with that method as it's `apply()` method, to be evaluated whenever it is requested.

```
void ExecutionEvaluator::evaluate(Block &expression) {
    auto meth = make_meth(expression.params(), expression.body());
    _partial = make_obj<GraceBlock>(meth);
}
```

Declaration Nodes

Declarations, from the EE's point of view, are nodes that add to the current scope in some way - be it adding new fields, or new methods. In general very little processing is done in declarations, and the do not modify the partial directly.

Field Declarations

Field Declarations are the nodes that, when processed, insert a field with an initial value in the current scope. The processing of these nodes is quite simple, since they delegate the initial value processing to their respective children. After retrieving the initial value, evaluating them is a matter of extending the current scope to include the new field:

```
void ExecutionEvaluator::evaluate(VariableDeclaration &expression) {
    // If an explicit initial value is defined, initialize the
    // variable to that. Otherwise, initialize it to an empty object.
    if (expression.value()) {
        expression.value()->accept(*this);
        _currentScope->setField(expression.name(), _partial);
    } else {
```

```
        _currentScope->setField(expression.name(), make_obj<UserObject>());
    }
}
```

Note that the evaluation of Field declarations assumes that the scope of the evaluator is the desired one at the time of evaluation.

Method Declarations

The evaluation of a Method Declaration has the aim of extending the method tables of the current scope to contain a new user-defined method. As it is the case with Blocks, the body of the Method Declaration will not be evaluated until a Request for it is evaluated and effective parameters are provided.

To evaluate a Method Declaration, a new Method has to be created with the formal parameters and body of the declaration, and it must be added to the current scope:

```
void ExecutionEvaluator::evaluate(MethodDeclaration &expression) {
    MethodPtr method = make_meth(expression.params(), expression.body());
    _currentScope->addMethod(expression.name(), method);
}
```

Control Nodes

Control structures in Grace are identical in behavior to their C++ counterparts, which makes the evaluation of control nodes incredibly intuitive, by using the methods natively available in the implementation language.

When evaluating a conditional node for example, 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). It is clear how this can be accomplished with C++'s native `if () {} else {}` construct.

```
void ExecutionEvaluator::evaluate(IfThenElse &expression) {
    expression.condition()->accept(*this);
    auto cond = _partial->asBoolean().value();
    if (cond) {
        for (auto exp : expression.thenPart()) {
            exp->accept(*this);
        }
    } else {
        for (auto exp : expression.elsePart()) {
            exp->accept(*this);
        }
    }
}
```

Analogous implementation is necessary for the While nodes.

```
void ExecutionEvaluator::evaluate(While &expression) {
    expression.condition()->accept(*this);
    auto cond = _partial->asBoolean().value();
    while (cond) {
        for (auto exp : expression.body()) {
            exp->accept(*this);
        }

        // Re-evaluate condition
        expression.condition()->accept(*this);
        cond = _partial->asBoolean().value();
    }
}
```

Since the method scope management is implemented in the Method class, the only responsibility of the Return node is to serve as a stopping point (leaf) in the execution tree. Note that the value of the return node is an expression, and thus the return value will be implicitly stored in the partial when returning from this function.

```
void ExecutionEvaluator::evaluate(Return &expression) {
    expression.value()->accept(*this);
    return;
}
```

Assigment

The aim of evaluating an Assignment node is to modify a field in the current scope to reference a new object.

The first step in evaluating an Assignment node is to retrieve the new value we want the field to contain, by evaluating the value branch of the node. The value branch is an expression, and thus the result of the call will ultimately be located in the partial, from which we can retrieve it and store it to assign to the new field later.

An Assignment can be performed on a field of the curent scope or a field in any of the objects contained in the scope. Therefore, the second step in evaluating an Assignment node is to set the scope to the one where the target field is located, in a manner analogous to the evaluation of the Object Constructors. For this, it is necessary to evaluate the scope fields of the node, and set the scope to the resulting value. Note that they will always be requests, and almost always they will have the form of field request chains (e.g. `self.obj.x`).

Finally, the only remaining thing is to modify the desired field to hold the new value and restore the original scope.

```
void ExecutionEvaluator::evaluate(Assignment &expression) {
    // Calculate the desired value and save it
    expression.value()->accept(*this);
```

```
    auto val = _partial;

    // Calculate the target object and set the EE's scope
    auto oldScope = _currentScope;
    expression.scope()->accept(*this);
    _currentScope = _partial;

    // Modify the correct field to have the new value
    _currentScope->setField(expression.field(), val);

    // Restore the old scope
    _currentScope = oldScope;
}
```

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 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, `GraceObject`s 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.

Memory Management

Grace is a garbage-collected language (**???**), and therefore there must be some mechanism to automatically control memory consumption during the evaluation.

Reference-counting

The first proposed solution to this problem was to have reference-counted objects, so that when an object would be referenced by either the evaluator or one of the objects in the subscopes of the evaluator. That way, every object accesible from the evaluator would have at least one reference to it, and would get destroyed when it went out of scope.

In this implementation, a factory function can be defined to create objects. With the help of C++ templates, a single static function is sufficient to instatiate any subclass of GraceObject.

```
template <typename T, typename... Args>
static std::shared_ptr<T> make_obj(Args&&...args) {
    return std::shared_ptr<T>{new T{std::forward<Args>(args)...}};
}
```

This function can be called from anywhere in the project (usually the evaluators and test cases), and the function will know which arguments the class constructor needs.

```
auto num = make_obj<GraceNumber>(5.0);
```

This implementation was sufficiently functional and easy to implement to facilitate the development of the evaluator and the object model. However, reference-counting as a memore management strategy has a number of fatal flaws, the worse of them being the *circular reference problem* (**???**). When reference-counting objects, it is possible to form cycles in the reference graph. If such a cycle were to form, then the objects inside the cycle would always have at least one other object referencing them, and thus would never get deallocated.

Heap and ObjectFactory classes

The next step was to use one of the well-researched memory management algorithms. With that in mind a Heap class was created to simulate a real dynamic memory store, and implement garbage collection over that structure. The Heap would have the responsibility of controlling the lifetime of an object or, as it is said in C++, *owning* that object's memory lifespan.

It is the responsibility of the Heap to manage an object's memory, but this management should be transparent to the type of the object itself. The Heap should only store GraceObjects, without worrying about the type of object it is. Therefore, including object factory methods in the Heap would be unadvisable. Instead, a façade was created to aid in the object creation process, called ObjectFactory.

The responsibility of this class is to provide a useful interface for the evaluator to create objects of any type wihtout interacting with the Heap directly. As an added benefit, this implementation of ObjectFactory could keep the interface for object creation described above, so that minimal existing code modifications were needed.

Integration

In order to integrate the newly created Heap with the evaluation engine, some minor changes need to be made.

Since now the Heap is managing the memory, the evaluator can stop using reference-counted pointers to reference objects. Instead, it only needs raw pointers to memory managed by the Heap. The same happens with the pointers held by GraceObjects. Since every object reference uses the GraceObjectPtr wrapper, this change is as simple as changing the implementation of the wrapper:

```
// What was
typedef std::shared_ptr<GraceObject> GraceObjectPtr;

// Is now
typedef GraceObject* GraceObjectPtr;
```

Since the interface provided by `std::shared_ptr<>` is similar to that of raw pointers, most of the code that used GraceObjectPtrs will remain untouched.

The second change to integrate the Heap into the project is to have each evaluator hold an instance of Heap. There should be only one instance of an execution evaluator per programming session, and therefore it is reasonable that every instance of the evaluator will have an instance of the Heap.

// TODO: Add UML diagram

Lastly, the GraceObject class needs to be extended to allow the retrieval of all the fields to ease traversal, and to include a `visited` flag so that the algorithm knows which objects to delete.

```
class GraceObject {
protected:
    std::map<std::string, MethodPtr> _nativeMethods;
    // ...

public:
    bool _visited;
    // ...

    std::vector<GraceObjectPtr> fields();
};
```

Garbage Collection Algorithm

In order to implement garbage collection in the Heap, an appropriate algorithm had to be selected from the myriad of options available. When reviewing the different possibilities, the focus was set on finding the simplest algorithm that could manage memory without memory leaks. This criteria was informed by the desire of making Naylang a learning exercise, and not a commercial-grade interpreter. As a result, the **Mark and Sweep** garbage collection algorithm was selected (**???**), since it is the most straightforward to implement.

In this algorithm, the Heap must hold references to all objects created in a list. Every time memory liberation is necessary, the Heap traverses all the objects accessible by the current scope of the evaluator with a regular graph search. Whenever it reaches an object that was not reached before, it marks it as "accesible". After that, every node that is not marked as accessible is deemed destroyable, and it's memory is deallocated.

Since this implementation of the Heap only simulates the storage of the objects, and does not make claims about it's continuity, heap fragmentation cannot happen. Therefore, no strategy is needed to defragment the memory.

Note that the Heap is implemented in such a way that the garbage-collection functionality is blocking and synchronous, and thus it can be called at any point in the evaluator. This would enable, for example, to implement an extension of the evaluator to include garbage collection triggers at key points of the exection, using the Modular Visitor Pattern.

Implementation

The internal design of the Heap class is vital to ensure that the objects are stored in an efficient manner, and that the garbage collection itself does not hinder the capabilities of the evaluator too greatly.

Object storage

The requirements for object storage in the Heap must be taken into consideration when selecting a data structure for object storage.

Of course, all objects must be **accessible at any point** in the execution, but this is accomplished with pointers returned at object creation and not by looking up in the Heap storage itself. Therefore, a structure with the possibility for fast lookup (such as an `std::map` (**???**)) is not necessary. Furthermore, it can be said that the **insertion order is not important**.

The mark and sweep algorithm needs to **traverse** the stored objects at least twice every time the garbage collection is triggered: Once to mark every object as not visited, and another time after the marking to check whether or not it is still accesible. Therefore, the storage must allow the possibility of traversal, but it

does not need to be extremely efficient since a relatively small number of passes need to be made.

Lastly, the storage must allow to **delete elements at arbitrary locations**, since at any point any object can go out of scope and will need to be removed when the collector triggers. This is perhaps the most performance-intensive requirement, since several object deletions can be necessary for each pass.

The two first requirements make it clear that a linear storage (array, vector or linked list) is needed, and the last requirement pushes the decision strongly in favor of a linked list. Luckily, C++ already has an implementation of a doubly-linked list (**???**), which the Heap will be using.

With the container selected, the only remaining thing is to establish which of C++'s mechanisms will be used to hold the object's lifespan. The concept of *memory ownership* was introduced in a previous section, and it was established that the Heap is responsible for *owning* the memory of all runtime objects (**???**). In modern C++, memory ownership is expressed by means of a *unique pointer*, that is, a smart pointer that has exactly one reference. The object that holds that reference then is responsible for keeping the memory of the referenced object. When the container object goes out of scope or is destroyed, the destructor for the contained object is immediately called, liberating the memory (**???**). In the case of Naylang, this menas that the object will be destroyed either when it is extracted from the list, or when the list itself is destroyed.

With this information, the Heap storage can be designed as a **linked list** of *cells*, wherein each *cell* is a `unique_ptr` to an instance of one of the subclasses of `GraceObject`.

// TODO: Include diagram

Mark and Sweep algorithm

The implementation of the algorithm itself is rather straightforward, since it is nothing more complicated than performing several traversals in the object storage:

```
void Heap::markAndSweep() {
    for (auto obj : _storage) {
        obj->_visited = false;
    }

    auto scope = _eval->currentScope();
    scope->_visited = true;
    visit(scope);

    int index = 0;
    std::vector<int> toDelete;
    for (auto obj : _storage) {
        if (!obj->_visited) {
            toDelete.push_back(index);
```

```
        }
        index++;
    }

    for (auto ndx : toDelete) {
        _storage.erase(ndx);
    }
}

void Heap::visit(GraceObject* scope) {
    for (auto field : scope->fields()) {
        field->_visited = true;
        visit(field);
    }
}
```

Memory capacity and GC triggers

Ideally, the garbage-collection mechanism would be transparent to the evaluator, meaning that no explicit calls to the collection algorithm should be done from the evaluation engine. Rather, it is the Heap itself who must determine when to trigger the GC algorithm. To this end, the Heap is initialized with three values:

- An absolute capacity, which acts as a upper bound for the storage available. When the number of objects contained in the Heap reaches this value, any subsequent attempts to create objects will result in an error.
- A trigger threshold, which indicates the Heap when it needs to start triggering the garbage collection algorithm. When this number of stored objects is surpassed, the Heap will start triggering the garbage collection algorithm with every interval.
- The object creation interval. This value indicates how often garbage collection has to trigger once the threshold has been hit. For instance, if this value is ten the garbage collection will trigger every tenth object inserted, if the threshold has been hit.

Therefore, this would be the code relevant to triggering the garbage collection:

```
void Heap::triggerIfNeeded() {
    if (_totalObjects >= _maxCapacity) {
        throw std::string{"Out of Memory"};
    }

    if (_nthObject == _interval) {
        if (_totalObjects >= _threshold) {
            markAndSweep();
        }
    }
}
```

Note that, even though objects may vary in size slightly, there are never degenerate differences in size, since even a big object with many fields has every one of the fields stored as a separate objects in the Heap.

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

During the development of the Naylang debugger, the need arised to integrate it with the existing architecture. Specifically, it was important to take advantage of the existing evaluation behavior and build the debugging mechanism *on top of it*, thus avoiding the need to reimplement the evaluation of particular AST nodes just so that the debugging behavior could be embedded mid-processing. This left two possibilities: Either the evaluator was modified to include the debugging behavior, or the debugging behavior was specified elsewhere, and then somehow tied with the evaluator.

Even though the first possibility is much easier to implement, it had serious drawbacks affecting the maintainability and extensibility of the evaluation engine. Since the debugging and evaluation behavior would be intermixed, any time a change had to be made to either part, extensive testing would be required to ensure that the other engine did not suffer a regression. Even with these drawbacks this was the first approach taken when implementing Naylang, with the intention of factoring out the debugger behavior later on. When the core debugger behavior was implemented, the factoring process started.

During the factoring process, a new programming pattern arised. This new pattern allowed for the development of completely separate processing engines, each with it's own set of behaviors, that could be composed to create more powerful engines. After some experimentation, this pattern yielded great results for implementing the Naylang debugger, and showed promising potential for implementing further features of the language.

Description

This pattern takes advantage of the very structure of Visitor-based interpreters. In this model of computation, every node in the AST has a certain function associated with it, which provides implicit entry and exit points to the processing of every node. This gives the class that calls these methods total control over the execution of the tree traversal. Up to this point, this caller class was the evaluator itself.

However, the key to this technique is to take advantage of those intervention points and the extra control over the execution flow and insert arbitrary code at those points. This code pieces could potentially do anything, from pausing the normal evaluation flow (e.g. in a debugger) to modifying the AST itself, allowing for any new feature to be developed.

This pattern is most comfortably used with classes that implement the same methods as the original class, since that will provide with a common and seamless interface with the rest of the system.

The following sections explain different variations in the pattern, and provide examples based on how Naylang would implement the debugging mechanism with each of the variations.

Direct Subclass Modularity

The most straightforward way to implement a Modular Visitor is to directly subclass the class that needs to be extended. This way, the old class can be replaced with the new subclass in the parts of the system that need that functionality with minimal influence in the rest of the codebase.

// TODO: Add class diagram

By directly subclassing the desired visitor, the implementer only needs to override the parts of the superclass that need code injected, and it can embed the normal execution flow of the application by calling the superclass methods.

Example

In Naylang, this would translate to creating a direct subclass of `ExecutionEvaluator`, called `DebugEvaluator`. As is described in Debugging, the aim of this class is to maintain and handle the current debug state of the evaluation (`STOP`, `RUN`…), and to maintain breakpoints.

Assuming the previous mechanisms are in place to handle state, the only capability required from the debugger is to be able to **block the evaluation** of the AST at the points where it is required (e.g. by a breakpoint). As previously described this can only happen in *stoppable* nodes, and therefore only the processing of those nodes need to be modified. For this example, assume that only `VariableDeclaration` and `ConstantDeclaration` nodes are stoppable, and that we need to add processing **both at the beginning and at the end** of the node evaluation to handle changing debug states.

To implement this, it is sufficient to override the methods that process those nodes, and to insert the calls to the debug state handlers before and after the call to the parent class. Every other processing would follow it's flow as normal.

```
class DebugEvaluator : public ExecutionEvaluator {
    DebugState _state;

public:
    // Override the desired function
    virtual evaluate(VariableDeclaration &expression) override;
}

void DebugEvaluator::evaluate(VariableDeclaration &expression) {
```

```
    // Call to the debug mechanism
    beginDebug(expression);
    // Call superclass to handle regular evaluation
    ExecutionEvaluator::evaluate(expression);
    // Call to the debug mechanism
    endDebug(expression);
}
```

Discussion

This version of the pattern is the most straightforward to implement, and has minimal impact in how the visitors are used and instantiated. However, it is the version that most limits the modularity of the evaluation system since as more visitors get added to the class hierarchy the inheritance tree deepens considerably. This oftern will result in an unmaintainable class hierarchy with very little flexibility.

Composite Modularity

As a way of solving the rigidity issues posed by the previous version of the pattern, this second version transforms the pattern to use *composition instead of inheritance*, as it is usually preferred by the industry (**???**).

In this technique, what previously was a subclass of the extended class is now at the same level in the class hierarchy. Instead of calling the superclass to access the implementation of the main visitor, the extender class *holds a reference* to the main class and uses it to call the desired evaluation methods.

// TODO: Add class diagram

Obviously, since the main visitor is not being extended anymore, **all of the methods** it implements will have to be overriden from the extender class to include *at least* calls to the main evaluator.

Example

There is little to be changed from the previous example in terms of code. The only necessary changes are to adapt the class declaration of `DebugEvaluator` to hold an instance of `ExecutionEvaluator` instead of inheriting from it, and to change the call to the superclass inside the evaluation methods. All of the methods implemented by `ExecutionEvaluator` must be overriden by `DebugEvaluator`, to include at least calls to `ExecutionEvaluator`.

Lastly, `DebugEvaluator` needs to have some way of obtaining a reference to a valid `ExecutionEvaluator` instance, be it by receiving it in the constructor or by creating an instance itself at startup.

```
class DebugEvaluator : public Evaluator {
    DebugState _state;
    // Note that it will accept any subclass of Evaluator
    Evaluator *_super;

public:
    // Obtain a reference to the desired evaluator
    DebugEvaluator(Evaluator *super);
    // Override from Evaluator this time.
    virtual evaluate(VariableDeclaration &expression) override;
    virtual evaluate(NumberLiteral &expression) override;
    // ...
}

void DebugEvaluator::evaluate(VariableDeclaration &expression) {
    // Call to the debug mechanism
    beginDebug(expression);
    // Call ExecutionEvaluator to handle regular evaluation
    _super->evaluate(expression);
    // Call to the debug mechanism
    endDebug(expression);
}

void DebugEvaluator::evaluate(NumberLiteral &expression) {
    // Only need to call the normal evaluation
    _super->evaluate(expression);
}
```

Discussion

This method simplifies greatly the class hierarchy by moving the composition of visitors from the subclassing mechanism to runtime instantiation. However, this also means that the desired composition of visitors must be explicitly instantiated and passed to their respective constructors (e.g. via *factory methods*).

This problem can be circunvented by having the extender class explicitly create the instances of the visitors it nedds directly into it's constructor. This can be a solution in some cases, but implementors must be aware of the tradeoff in flexibility that it poses, since then the extender is bound to have only one possible class to call.

Lastly, another great drawback of this technique is that it forces the extender class to implement at least the same methods as the main visitor implemented, to include calls to that. This might not be desirable in extensions that only require one or two methods to be modified from the main class.

Wrapper Superclass Modularity

This final version of the Modular Visitor Pattern tries to solve some of the issues with the previous two implementations, while having minimal tradeoffs. Specifically, it aims to provide a system that:

- Is flexible enough to allow for a shallow inheritance tree and composability, and
- Only requires a visitor extension to override the methods that it needs to override, and not be conditioned by the class it is extending.

One way to accomplish these goals is to define an intermediate layer of inheritance in the class hierarchy such that all the default calls to the main visitor are implemented in a superclass, and only the relevant functionality is implemented in a subclass. Roughly speaking, it consists on **grouping together** extensions that need to interject the execution at similar times, and **moving all the non-specific code to a superclass**. This way, it is the superclass that has the responsibility of handling the main evaluator instance.

// TODO: Add class diagram

Example

Following the previous example, it is possible to define a superclass that agrupates the behavior of "executing code before and after evaluating a node". Let us call that class `BeforeAfterEvaluator`. This class has the responsibility of implementing calls to the regular evaluation and providing interfaces for the `before()` and `after()` operations.

```
class BeforeAfterEvaluator : public Evaluator {
protected:
    Evaluator *_super;

public:
    BeforeAfterEvaluator(Evaluator *super);

    virtual evaluate(VariableDeclaration &expression) override {
        _super->evaluate(expression);
    }
    // ...
    virtual void before(Statement *stat) = 0;
    virtual void after(Statement *stat) = 0;
}
```

Having done that, we can transform `DebugEvaluator` to be a subclass of `BeforeAfterEvaluator`, and thus inherit the regular calls to the main evaluator. We can then override the processing of `VariableDeclarations` to include calls to `before()` and `after()`, and implement those methods to include the debugging behavior:

```
class DebugEvaluator : public BeforeAfterEvaluator {
    DebugState _state;

public:
    // Override the desired function
    virtual void before();
    virtual void after();

    virtual evaluate(VariableDeclaration &expression) override;
}

void DebugEvaluator::evaluate(VariableDeclaration &expression) {
    before(&expression);
    _super->evaluate(expression);
    after(&expression);
}
```

// TODO: Add class diagram

Discussion

This is by far the most flexible method, and the one that offers the best tradeoff in terms of ease-of-use and flexibility. However, it requires a great amount of setup effort in order to make it easy to add new subclasses, and therefore it is only worth it for projects that plan to use visitor composition extensively.

Applications

This visitor design pattern has a myriad of applications. The main benefit is that it allows to practically extend the functionality of an intepreting engine without needing to change the previous processings. It permits to add both semantic power to the language (e.g. by creating a type checking extension, or an importing system) and extralinguistic tools (such as the debugging mechanism) with minimal risk to the existing processing core of the language.

Further investigation is necessary, but this technique could lead to a way of **incrementally designing a language**, wherein a language implementation could grow incrementally and iteratively in parallel to it's design and specification, safely. It is not hard to imagine the benefits of having the most atomic parts of a language implemented first, and more visitor extensions are added as more complex features are introduced to the language.

This idea of a fully modular language has been developed in several academic works where the use of monads was suggested (**???**). This approach, when applied specifically to Visitor-based interpreters, allows similar levels of flexibility while maintaining the approachability that a design pattern requires.

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.

Note that, from this point forward, `TEST_CASE()` refers to a construct in the framework, while "test case" refers to a logical set of one or more assertions about the code, which will usually be included inside a `SECTION()`.

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 va
```

```
        REQUIRE_NOTHROW(ImplicitRequestNode variableReq("x"););
        REQUIRE_NOTHROW(ImplicitRequestNode methodReq("myMethod"););
    }

    // Accessor sections
    SECTION("A ImplicitRequestNode can return the identifier name and parameter
        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

The evaluator was one of the more complicated parts of the system to test, since it's closely tied to both the object model and the abstract representation of the language. In addition to that, it is very useful to be able to make assertions about the internal state of the evaluator after evaluating a node, which goes against the standard practice of testing an object's interface, and not it's internal state. This problem required an extension of the Evaluator to be able to make queries about and modify it's internal state (namely, the current scope and the partial result), which later proved useful when implementing user-defined method evaluation.

The test structure of the evaluator is probably one of the lengthiest ones in the project, since the evaluation of every node has to be tested, and some nodes need more than one test case such as Requests, which can be either field or method requests. Therefore, the evaluator test file contains several test cases:

- **Particular Nodes** tests the evaluation of every node in the AST, with at least a section for each node class.
- **Environment** tests the scope changes and object creation of the evaluator.
- **Native Methods** and **Non-Native Methods** test the evaluation of methods by creating a placeholder object and requesting a method from it.

Testing the Objects

Test files for object classes have two `TEST_CASE()`s defined in them, since objects have two responsibilities: To hold the relevant data of that type (e.g. a boolean value for GraceBoolean) and to implement the required native methods. Thus, a `TEST_CASE()` was defined for each of these responsibilites. Since native methods are defined as internal classes to the objects, it is natural to test them in the same file as the object.

```
TEST_CASE("Grace Boolean", "[GraceObjects]") {
    GraceBoolean bul(true);

    SECTION("A GraceBoolean can return it's raw boolean value") {
        REQUIRE(bul.value());
    }

    // ...
}

TEST_CASE("Predefined methods in GraceBoolean", "[GraceBoolean]") {

    SECTION("Not") {
        MethodRequest req("not");
        GraceBoolean::Not method;

        SECTION("Calling Not with self == GraceTrue returns GraceFalse") {
            GraceObjectPtr val = method.respond(*GraceTrue, req);
            REQUIRE(*GraceFalse == *val);
        }

        SECTION("Calling Not with self == GraceFalse returns GraceTrue") {
            GraceObjectPtr val = method.respond(*GraceFalse, req);
            REQUIRE(*GraceTrue == *val);
        }
    }

    // ...
}
```

Testing the Naylang Parser Visitor

The testing methodology for the parser was standardized rather quickly, with the aim of making writing additional tests as quick as possible. The job of the parser is to translate strings into AST nodes, so every test has a similar structure:

1. Form the input string as the valid Grace statement under test (e.g. `"var x := 3;"`).
2. Perform all the steps necessary to feed the input string into the parser. Since this process in ANTLRv4 is rather verbose and repetitive, it has been factored out into a function:

```
GraceAST translate(std::string line) {
    ANTLRInputStream stream(line);
    GraceLexer lexer(&stream);
    CommonTokenStream tokens(&lexer);
    GraceParser parser(&tokens);
```

```
        NaylangParserVisitor parserVisitor;

        auto program = parser.program();
        parserVisitor.visit(program);

        auto AST = parserVisitor.AST();
        return AST;
    }
```

3. Retrieve the AST resulting from the parsing process (e.g. `auto AST = translate("var x := 3;");`).
4. Use static casts (**???**) and assertions to validate the structure of the tree.

```
// Full example test case

SECTION("Assignments can have multiple requests and an identifier") {
    // Translation
    auto AST = translate("obj.val.x := 4;\n");

    // Conversion
    auto assign = static_cast<Assignment &>(*(AST[0]));
    auto scope = static_cast<ExplicitRequestNode &>(*assign.scope());
    auto obj = static_cast<ImplicitRequestNode &>(*scope.receiver());
    auto val = static_cast<NumberLiteral &>(*assign.value());

    // Validation
    REQUIRE(assign.field() == "x");
    REQUIRE(scope.identifier() == "val");
    REQUIRE(obj.identifier() == "obj");
    REQUIRE(val.value() == 4.0);
}
```

All of the test cases follow a similar structure, and are grouped in logical `TEST_CASE()`s, such as "Control Structures" or "Assignment".

Integration testing

To test whether particular features of the language fit inside the grand scope of the project, a series of integration tests were developed. These tests are comprised of Grace source files, which for the moment have to be run by hand from the interpreter. The files are located in the `/examples` folder, and each of them is designed to test the full pipeline of a particular feature of the language, from parsing to AST construction and evaluation. For example, `Conditionals.grace` tests the `if () then {}` and `if () then {} else {}` constructs, while `Debugger.grace` is aimed to provide a good test case for the debugging mechanism.

Testing Frontends

Naylang does not feature any unit tests for the frontends, for several reasons. On the one hand, the frontends are not part of the core evaluation and debugging system, and thus are not as important for the prospect student to learn from. On the other hand, the frontends feature some of the shortest and most industry-standard code of the project, and thus it's design is deemed straightforward enough to not grant their inclusion in the test suite.

However, as more and more complicated frontends are added to the project, the possibility of including them in the test suite will be reconsidered.

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=methodBodyI
  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. How was this document made?

Author

The process described in this Appendix was devised by Álvaro Bermejo, who published it under the MIT license in 2017 [@persimmon]. What follows is a verbatim copy of the original.

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