Trabajo de Fin de Grado del Grado en Ingeniería Informática

Facultad de Informática,
Universidad Complutense de Madrid

Intérprete y Depurador de Grace

Borja Lorente

Director: José Luis Sierra Rodríguez

Abstract

Naylang is an Open Source REPL interpreter and debugger for a subset of the Grace programming language, written entirely in modern C++. The focus of Naylang is on education for both the user and the future contributors, and thus offers extensive test coverage and simple implementations of the most common language components. The front-end features the ANTLRv4 C++ target for parsing direct left-recursive grammars. The core is structured as a Visitor-based interpreter, and introduces the Modular Visitor Pattern to the realm of programming languages.

Keywords: Intepreters, Programming Languages, Debuggers, Grace .

Abstract

Naylang es un intérprete REPL (Read-Eval-Print-Loop), depurador y entorno de ejecución Open Source para un subconjunto del lenguaje de programación Grace, implementado enteramente en C++14. Se enfoca en la educación tanto para como los usuarios finales como para futuros implementadores, y por lo tanto ofrece una extensa cobertura de tests e implementaciones simples para los componentes más comunes de un lenguaje. El front-end hace uso del target C++ de ANTLRv4 para reconocer gramáticas recursivas a izquierdas. El núcleo de interpretación está estructurado como un intérprete basado en visitantes e introduce el Patrón de Visitante Modular a la comunidad de la implementación de lenguajes.

Keywords: Intérpretes, Lenguajes de Programación, Depuradores, Grace .

Contents

List of Figures

1. Introduction

Naylang is an open source REPL interpreter (Abelson et al., 1996), runtime and debugger for the Grace programming language implemented in C++14.

It currently implements a subset of Grace described later, but as both the language and the interpreter evolves the project will strive for near feature-completeness.

1.1. Motivation

Grace is a language aimed to help novice programmers get acquainted with the process of programming (Noble et al., 2013) (Black et al., 2013). As such, it provides safety and flexibility in its design.

However, that flexibility comes at a cost, and most of the current implementations of Grace are opaque and obscure. Since the language is Open Source, most of its implementations are also Open Source, but this lack of clarity in the implementation makes them hard to extend and modify by third parties and contributors, severely damaging the growth opportunities of the language.

1.2. Objectives

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.

1.3. Methodology

The project follows a Test Driven Development (Beck, 2003) agile methodology, in which unit tests are written in parallel or even before the source code in very short iterations. This is the best approach for two reasons:

- 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.
- 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.

The development of Naylang will be carried out in short iterations, usually less than a week long. This has the aim of exploring different architectural approaches to the problems that building an intepreter presents. This way, the benefit of complete test coverage is maximized by being able to completely redesign a subsystem without fear of a regression.

1.4. 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. For instance, it will give them the ability to add a substitute any inefficient part of the system with a more efficient implementation.

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

1.5. Structure of This Document

The rest of this document describes the implementation and results of the project. It first introduces the reader to the Grace programming language and shows the research done about the state of the art as it pertains Naylang. Following that, the document presents a section about the implementation phase of the project, which details the structure and inner workings of the relevant components of the system. After that, two short chapters describe the modular visitor pattern and the testing methodology used in development. Lastly, the results of the research are presented, with an assesment of the state of the project.

The appendices cover a wide range of non-vital material, such as the Spanish translations of the introduction and conclusion and the grammars used for parsing.

2. The Grace Programming Language

2.1. 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 (Noble et al., 2013). To that aim, Grace is designed to provide an intuitive and extremely flexible syntax while maintaining the standards of commercial-grade programming languages (Black et al., 2012).

2.2. Key Features

Grace is aimed towards providing a solid introduction to the basic concepts of programming. Therefore, the following features are all designed to facilitate the use of Grace in an academic setting.

2.2.1. 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 system.

Despite being imperative at its 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 having the possiblity to accept arguments (Black et al., 2013).

2.2.2. Safety

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.

2.2.3. Gradual typing

Grace is gradually typed, which means that the programmer may choose the degree of type checking that is to be performed. This flexibility is atomic at the statement level, which means that any object or method declaration may or may not be typed. For instance, we might have all of the following in the same file:

```
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 instructors the tools necessary 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, as is explained later in this document.

2.2.4. Object Model

Simirarly to other interpreted languages such as JavaScript or Ruby, everything is an object in Grace. A generic object can have constant or variable *fields* that point to other objects, and methods that store user-defined or native subroutines. An object's fields are accessible to any subscope inside that object. Particularly they can be used and assigned to in methods.

```
object {
    def base = "Hi";
    var times := 4;
    def objField = object {
        def innerField = true;
    };
    method repeatBase {
        var i := 0;
        ver res := "";
        while {i < times} do {
            res := res ++ base;
            i := i + 1;
        }
        return res;
    }
}
```

Native types are implemented as objects with no fields and a series of predefined methods (such as the boolean "*or*", `||(_)`).

2.2.5. 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 formal parameter list. That way, methods with much more intuitive names can be formed:

```
// Declaration
method substringOf(str)from(first)to(last) {
    // Method body
}

// Request (call)
substringOf("Hello")from(2)to(5); // 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 nothing more than the concatenation of each of the parts, substituting the parameter names with underscores. That way, the canonical name of the method above would be `substringOf(_)From(_)to(_)`.

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

```
method substringOf(str)from(first : Rational)to(last : Rational) {
    // Code
}

method substringOf(str)from(first : Integer)to(last : Integer) {
    // Code
}
```

In this case, the second method's signature is considered to be **the same** as the first method's, and it will cause a *shadowing error*[1] 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, as is explained in Methods and Dispatch

2.2.6. 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,

[1]http://gracelang.org/documents/grace-spec-0.7.0.html#declarations

so the declarations in a block are accessible to that scope and every scope inside it, but not to any outer scopes.

2.2.7. Lineups

Collections in Grace are represented as Lineups, which are completely polymorphic lists of objects that implement the Iterable interface. As the spec says, the common trait of Lineups is that they implement the `Iterable` interface. In the case of Naylang, since no inheritance or type system is needed yet, no such interface has been implemented. Rather, the `GraceIterable` native type has been created.

2.2.8. 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.

2.3. 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 implemented in Naylang:

```
// Literals
4;
4 + 5; // Number literals and operators
true && false; // Boolean literals and operators
"Hello" ++ " World"; // String literals and operators
["a", 6, true]; // Lineups

// Declarations
var empty; // Uninitialized variable declaration
var x := true; // Initialized variable declaration
def y = 6; // Constant declaration
method add(a)to(b) { // Method declaration
    return a + b;
}

// Object constructor
```

```
def obj = object {
    var size := 3;
    def arity = 1;
    method sizeTimesArity {
        return size * arity;
    }
};

// Lambda blocks
def str = "Block";
{ j ->
    j.substringFrom(2)to(5);
}.apply(str);

// Control structures
var i := 0;
while {i < 20} { // While
    if (i % 2 == 0) then { // If-then-else structure
        print "even";
    } else {
        print "odd";
    }
    i := i + 1; // Assignment
}
```

3. State of the art

Grace is a relatively new language, and thus it does not feature most of the vast tools and libraries other languages have. However, the open-source spirit of the language makes it so that it is possible to access the information available without restriction.

3.1. Kernan

Kernan is currently the most feature-complete implementation of Grace. It is an interpreter written entirely in C# (Hejlsberg et al., 2003), and it features some execution and AST models similar to those implemented in Naylang. Specifically, the method dispatch and execution flow takes heavy inspiration from Kernan. However, Kernan is not visitor-based, and therefore it and Naylang diverge in that regard, as Naylang features a flexible and extensible evaluator structure.

Kernan is publicly available from the Grace website[1].

3.2. Minigrace

Minigrace is the original Grace compiler (Homer, 2014), which is written in Grace itself via bootstrapping with C.[2] 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 in GitHub[3].

3.3. GDB

The GNU Project Debugger has for many years been the de facto debugger for C and C++, and thus it merits some time to study it. The main influence of GDB in Naylang is the design of its command set, that is, the commands it offers to the user. In particular, Naylang will focus on reproducing the functionality of the following commands: `run`, `continue`, `next`, `step`, `break` and `print`.[4] Naylang will add another command, `env`, that allows the user to print the current

[1] http://gracelang.org/applications/grace-versions/kernan/
[2] http://gracelang.org/applications/grace-versions/minigrace/
[3] https://github.com/gracelang/minigrace
[4] http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

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. Since Naylang is an intepreter and thus doesn't generate an executable, this information gathering technique is of course unusable by the project. Instead, Naylang gathers information from the AST (Abstract Syntax Tree) directly to control the debugging flow.

3.4. Evaluation modularity

The means by which a language's evaluation can be modularized have been discussed at length in the field of programming language implementation, specially pertaining to Domain Specific Languages (Sierra, 2004). For Naylang, this topic is specially interesting since the traditionally monolithic approaches to language interpreters (Aho et al., 1986) imposed a particularly hard barrier on the scope of the project.

Amongst these techniques, the ones that stood out the most are the monad-based approaches (such as the one formulated in (Espinosa, 1995)), and the mixin-based approaches (abstract subclassing, as presented in (Duggan, 2000)). These techniques however differ fundamentally from the Visitor-based interpreter pattern that was the aim of Naylang, and thus were discarded in favor of a new approach detailed in the Modular Visitor Pattern section.

4. Implementation

The implementation of Naylang follows that of a visitor-based interpreter (Gamma et al., 1995). First, the source code is tokenized and parsed with a parser generated by ANTLRv4. Then, a custom parser extension traverses the parse tree and generates and Abstract Syntax Tree from the nodes, annotating each one with useful information such as line numbers. Lastly, an evaluator visitor traverses the AST and interprets each node.

In addition to the REPL commands, Naylang includes a debug mode, which allows to debug a file with the usual commands detailed in Debugging and Frontends. The mechanisms necessary for controlling the execution flow are embedded in the evaluator, as is explained later.

Figure 4.1 shows the general structure of Naylang, and how the components fit with each other.

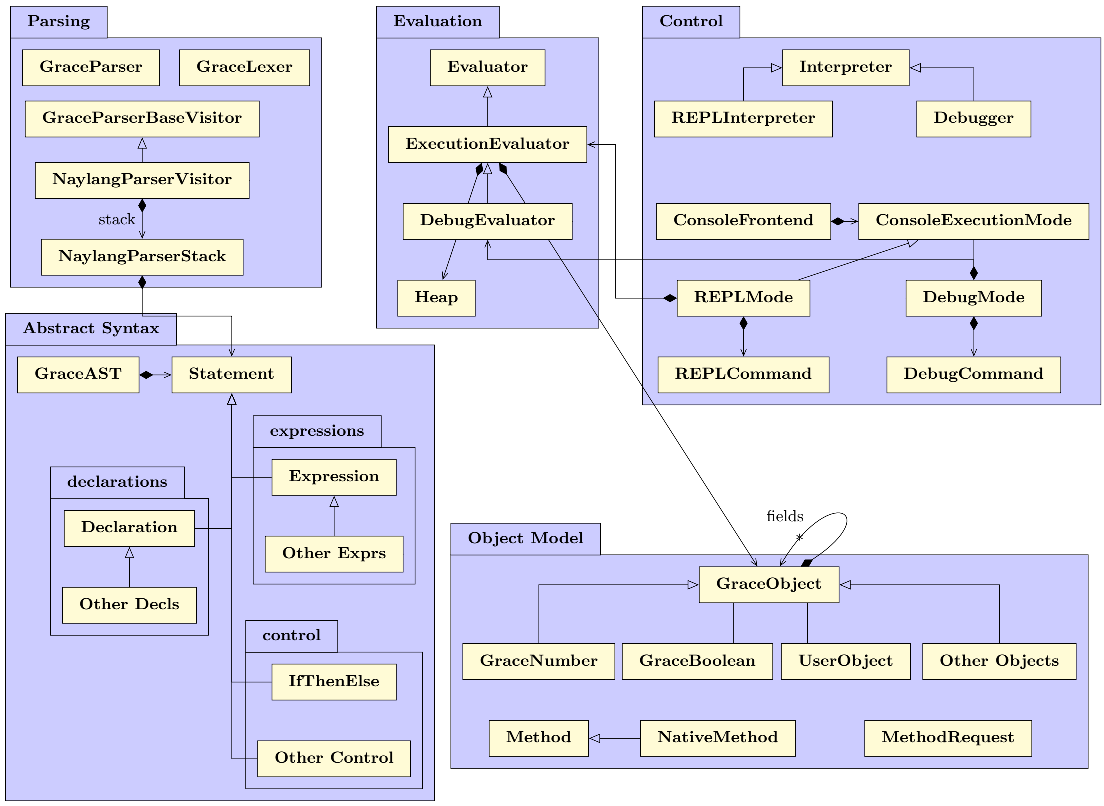

Figure 4.1.: Main Components of Naylang

4.1. Project Structure

The project is structured as a standard CMake multitarget project. The root folder contains a `CMakeLists.txt` file detailing the two compilation 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
```

4.1.1. 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 its 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
```

4.1.2. Tests

For automated testing, the Catch header-only library was used (Nash, 2014). The inner 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 `NaylangParserVisitor` will be found in `./tests/src/core/parser/NaylangParserVisitor_test.cpp`. Each file has one or more `TEST_CASE()`s, each with a number of `SECTION()`s. Sections allow for local and shared initialization of objects between tests.

4.1.3. 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`, which is an visitor of the implicit syntactic analysis tree generated by parsing the code. 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.

4.1.4. 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`[1], 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.

4.2. 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 its core, Naylang is designed to be an visitor-based interpreter (Parr, 2009). 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

[1] https://github.com/antlr/antlr4/tree/c8d9749be101aa24947aebc706ba8ee8300e84ae

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 Modular Visitor Pattern.

Figure 4.2 presents an example AST and its evaluation stack trace is presented below:

```
ExecutionEvaluator.evaluate(const GraceAST &ast)
|– VariableDeclaration.accept(Evaluator &eval)
    |– ExecutionEvaluator.evaluate(VariableDeclaration &node)
        |– ExecutionEvaluator.setIdentifier(node.id())
        |– Number.accept(this)
        |     |– ExecutionEvaluator.evaluate(NumberLiteral &node)
        |         |– ExecutionEvaluator.setPartial(new GraceNumber(node.val()))
        |– ExecutionEvaluator.declareVariable(id, _partial)
```

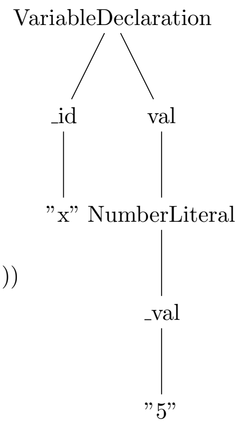

Figure 4.2.: Example AST for execution flow

4.3. 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 the rule methods and execute arbitrary code, as will be shown later.

This method allows instantiation of the AST independently from the grammar specification.

4.3.1. The Naylang Parser Visitor

For this particular program, the visitor versions of the lexer and parser were chosen from amongst the diferent parsing options provided, since their default implementation allowed for a preorder traversal of the parse tree, but offered enough flexibility to manually modify the traversal if needed. Note that the choice of the visitor pattern for static analysis is completely independent from that chosen for the runtime intepretation of the code. One might, for example, prefer to visit the *right* side of an assignment before moving onto the *left* side to instantiate particular types of assignment, depending on the assigned value. To that end, the `NaylangParserVisitor` class was created, which extends `GraceParserBaseVisitor`, a class designed to provide the default preorder implementation of the parse tree traversal.

The class definition along with the overriden method list can be found in `NaylangParserVisitor.h`. Note that ANTLRv4 names the visitor methods `visit<RuleName>` by convention. For example, `visitBlock()` makes it possible to visit the parse tree structure recognized by the `block` rule.

To pass data between methods, the Naylang Parser Visitor utilizes *two stacks*. The **first stack** 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 parameter expressions in a method call). 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, respectively.

4.3.1.1. Lexical Tree Visiting Strategy

The strategy followed 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 to 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 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 (Martin, 2009).

4.3.1.2. 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. The Grace specification states that infix and prefix operators must be converted to explicit requests to an object[2].

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)`, an explicit request for the `+(_)` method of the object `4` with `5` as a parameter.

4.3.2. The Naylang Parser Stack

During the AST construction process, information must be passed between parser function calls. A method call must, for instance, retrieve information about each of its effective parameter expressions. 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 impractical and error-prone.

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

- 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.

[2] http://gracelang.org/documents/grace-spec-0.7.0.html#method-requests

The resulting structure declaration can be found in `NaylangParserStack.h`. It uses template metaprogramming (Abrahams and Gurtovoy, 2004) 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 guarranteed 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;
}
```

4.3.3. 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 and native methods.

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, which is the case for example for LL(k)-grammar parser generators. ANTLRv4, however, can handle left-recursive rules as long as they are not indirect (Parr, 2013), which allows for the simplification of the grammars by introducing some ambiguity, which is resolved 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.

4.4. 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[3] and the close ties this language has with its execution model. This created a loop where design decisions in the execution model required changes in the AST representation, and vice versa. Figure 4.3 represents the current class hierarchy.

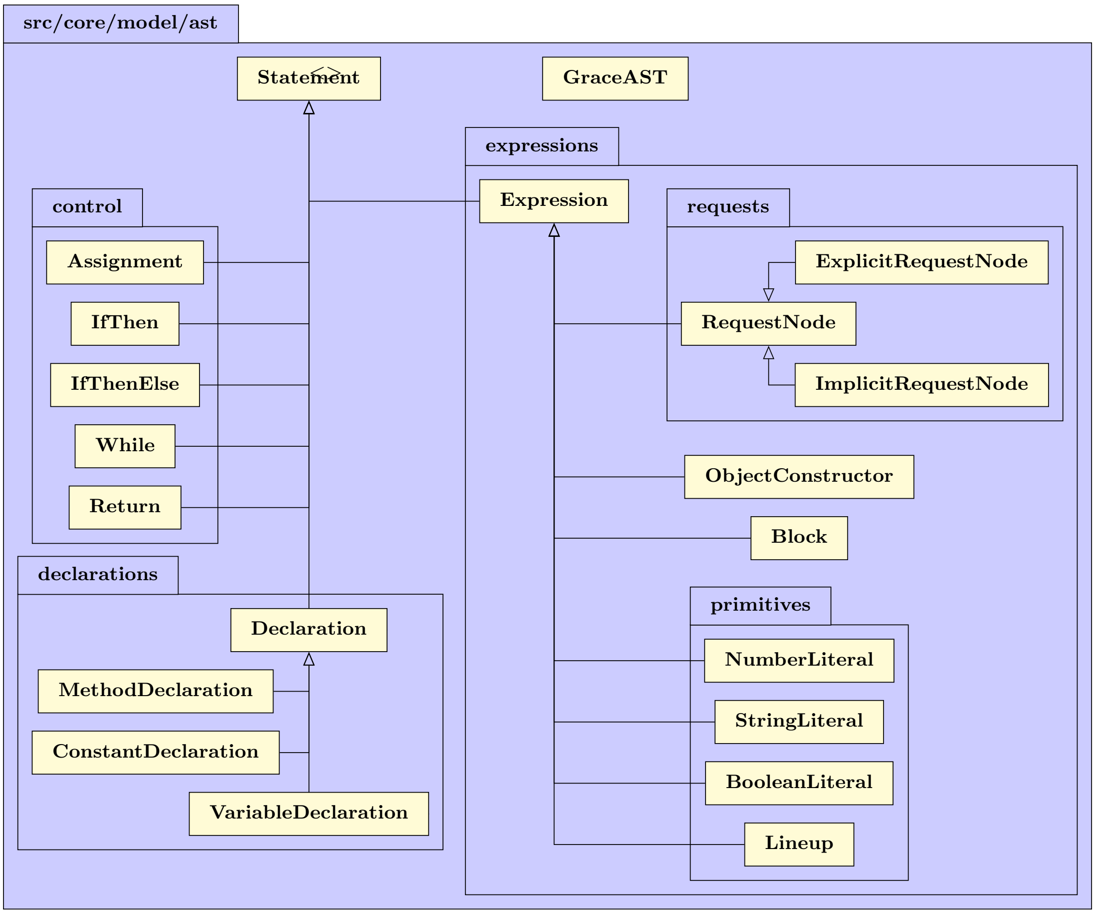

Figure 4.3.: Abstract Syntax Tree class hierarchy

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

The rest of this section covers the implementation of the memory management of the AST, as well as a description of the major nodes in the tree.

[3]http://gracelang.org/documents/grace-spec-0.7.0.html

4.4.1. 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 its value field, while a method declaration might want `DeclarationPtr`s for its formal parameters and high-level `StatementPtr`s for its body.

Currently, the aliases are implemented as reference-counted pointers (`std::shared_ptr<>`[4]). However, as the project has moved towards a centralized tree manager (`GraceAST`), the possibility of making that class responsible for the memory of the nodes has arised. This would permit the aliases to switch to weak pointers[5] or even raw pointers in their representation, probably reducing memory management overhead.

4.4.2. 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.

4.4.2.1. 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 support the language, and every other type of control structure (for loops, for instance) can be implemented in a prelude, in a manner transparent to the user.[6]

Figure 4.4 shows the class definitions of the existing control nodes

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

[4] http://en.cppreference.com/w/cpp/memory/shared_ptr
[5] http://en.cppreference.com/w/cpp/memory/weak_ptr
[6] http://gracelang.org/documents/grace-prelude-0.7.0.html#control-structures

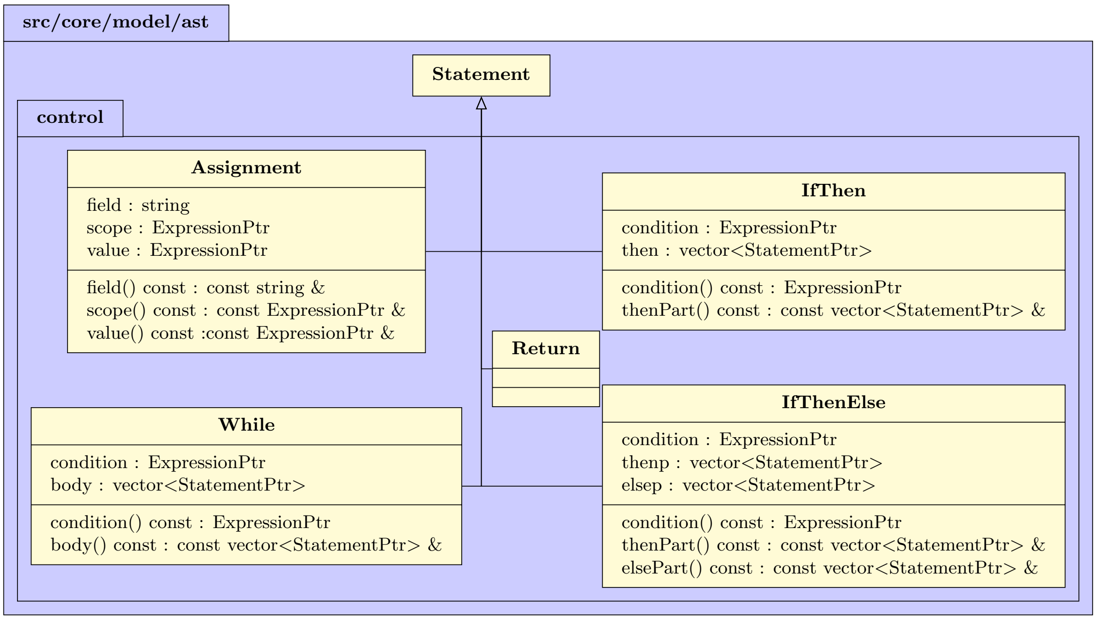

Figure 4.4.: Control nodes in Naylang

```
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.

4.4.2.1.2. Loop Nodes Loop nodes are the nodes used to execute an action repeatedly. In this case, only one node type is necessary, the While node. Every other type of loop can be composed in the Grace prelude using the While loop.

```
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.

4.4.2.1.3. 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()
};
```

4.4.2.2. 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 the 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 situation 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()
};
```

4.4.3. Declaration Nodes

The declaration nodes are nodes that do not return a value, and bind a specific construct 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**.

Figure 4.5 shows the class structure for declarations in Naylang:

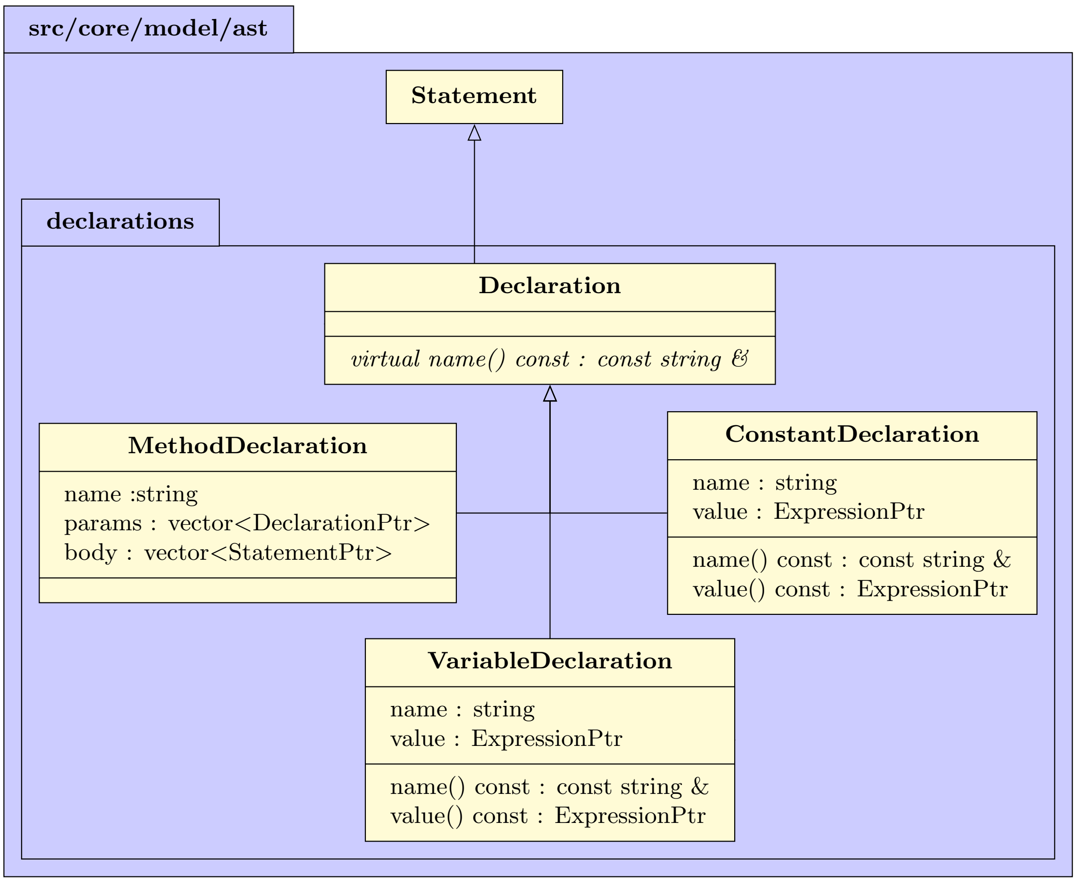

Figure 4.5.: Declarations in Naylang

4.4.3.1. Field Declarations

Field declarations represent the intent of mapping an identifier to a value in the current scope. Depending on the desired mutability of the expression, these declarations will be represented with either `ConstantDeclarations` or `VariableDeclarations`. These two nodes only differ in their evaluation, and their internal representations are identical. They both need an identifier to create the desired field, and optionally an initial value to give to that field. In the case of `ConstantDeclarations`, the initial value is not optional.

```
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()
};
```

4.4.3.2. 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. Syntactically, a `MethodDeclaration` is comprised of a canonical identifier[7], a list of formal parameter definitions (to be later instantiated and bound to 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()
};
```

[7] http://gracelang.org/documents/grace-spec-0.7.0.html#method-names

4.4.4. Expressions

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

4.4.4.1. 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 its type, and it corresponds directly with a native type constructor. For instance, a `NumberLiteral` node will only need to hold its numeric value, which is all that's necessary to create a `GraceNumber` object. Of course, this makes the evaluation of these nodes straightforward, as 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()
};
```

Figure 4.6 shows a diagram of the current primitive expressions in Naylang

4.4.4.2. Requests

Everything is an object in Grace, 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 expressions 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);
```

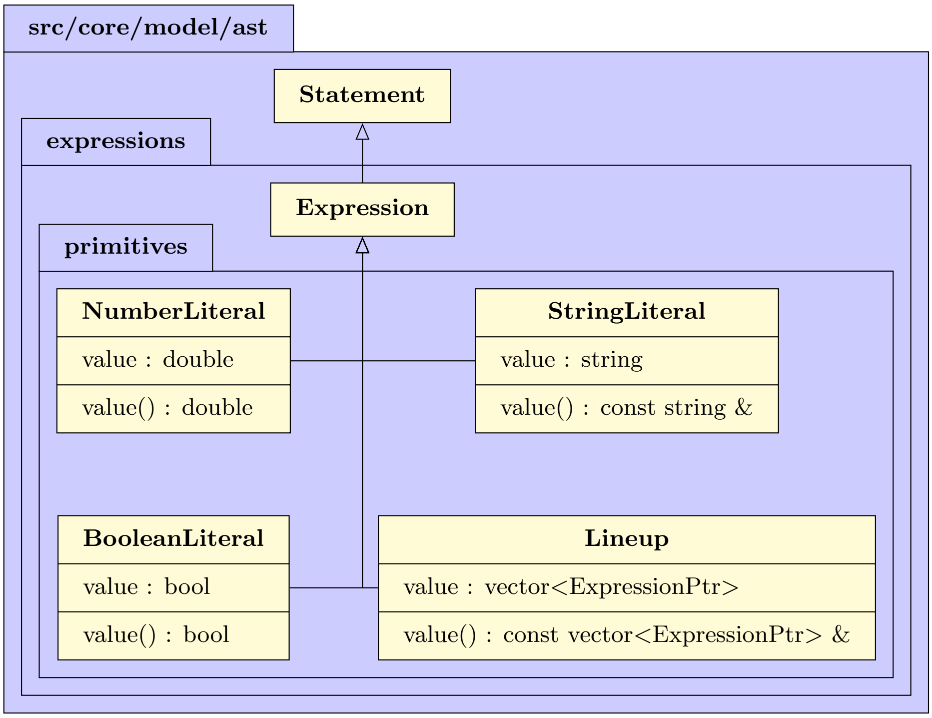

Figure 4.6.: Primitive expressions in Naylang

```
    // 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.

Figure 4.7 shows a diagram of the current requests in Naylang

4.4.4.3. 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 Grace object.

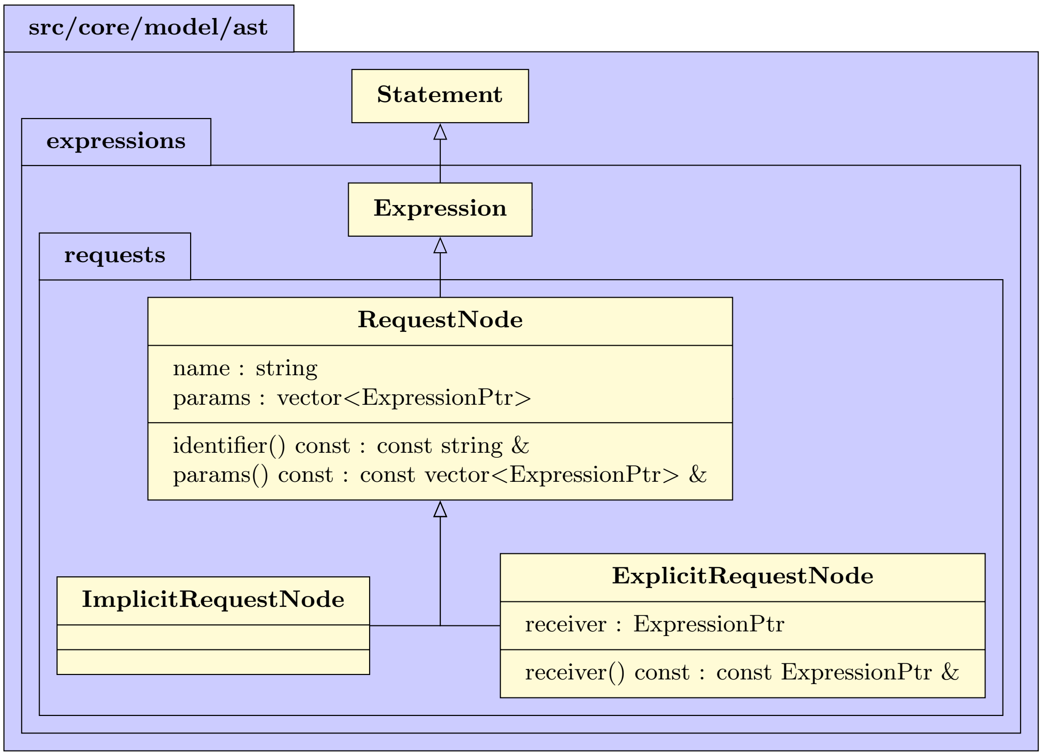

Figure 4.7.: Requests in Naylang

Since an object can contain virtually any Grace construct, an `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()
};
```

4.4.4.4. Block Nodes

`Block`s 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()
};
```

4.5. Execution Evaluator

The `ExecutionEvaluator` (or EE) is one of the most crucial components of Naylang. It is its 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 `ExecutionEvaluator` is the engine of the interpreter.

As previously described, the `ExecutionEvaluator` (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.

4.5.1. Structure

An important part of the EE is the mechanism used to share information between node evaluations. For instance, there has to be a way for the evaluator to access the number object 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** result object is the means of communicating between the evaluation of different nodes. 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
};
```

4.5.2. 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.

4.5.3. 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).

4.5.3.1. 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 = create_obj<GraceNumber>(expression.value());
}
```

4.5.3.2. 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 `ObjectConstructor` 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. The following algorithm has been used to evaluate the `ObjectConstructor` 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 = create_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;
}
```

4.5.3.3. Implicit Requests

These 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 its behavior.

However, Grace provides a useful invariant to design the evaluation of requests: All identifiers are unique within a scope or its 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**, its evaluation becomes simple. `Requests` are expressions, and thus must place a value in the partial. `ImplicitRequests` 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, paramValues);
}
```

4.5.3.4. Explicit Requests

They are similar to `ImplicitRequests`, the only difference being that `ExplicitRequests` 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 `ImplicitRequests`). 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 evaluation functions that are completely independent from each other.

4.5.3.5. Block Nodes

`Block` nodes are similar to `ObjectConstructor` nodes in that they place a new object with effectively arbitrary content in the partial. The only difference is that while `ObjectConstructor` 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 its `apply()` method, to be evaluated whenever it is requested.

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

4.5.4. 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 they do not modify the partial directly.

4.5.4.1. Field Declarations

Field Declarations are the nodes that, when processed, **insert a new 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(), create_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.

4.5.4.2. Method Declarations

The evaluation of a `MethodDeclaration` 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 `MethodDeclaration` will not be evaluated until a `Request` for it is encountered and effective parameters are provided.

To evaluate a `MethodDeclaration`, 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);
}
```

4.5.5. 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 means 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).

```
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;
}
```

4.5.6. 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 there, we can retrieve it and assign it 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 `ObjectConstructors`. 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;
}
```

4.6. Methods and Dispatch

One of the advantages of Grace is that it integrates native methods and user-defined methods seamlessly in its 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<StatementPtr> &body);
  // ...
};
```

4.6.1. 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 a context (`ExecutionEvaluator`) in which 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 objects, 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.

4.6.2. 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.

```cpp
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:

```cpp
GraceObjectPtr Method::respond(
  ExecutionEvaluator &context,
  GraceObject &self,
  MethodRequest &request)
  {
    // Create the scope where the parameters will be instantiated
    GraceObjectPtr closure = make_obj<GraceClosure>();
    // Instantiate every parameter in the closure
    for (int i = 0; i < request.params().size(); i++) {
        closure->setField(
            request.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;
}
```

4.6.3. 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). Some 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:
  virtual GraceObjectPtr respond(
      GraceObject &self, MethodRequest &req)
      {
      throw std::string {"Called an unimplemented native method"};
  }

  virtual GraceObjectPtr respond(
      ExecutionEvaluator &ctx, GraceObject &self, MethodRequest &req)
      {
      return respond(self, req);
  }
};
```

Each native method is a subclass of `NativeMethod`, and implements its functionality in the body of the overriden `respond()` method. For convenience, each subclass of `GraceObject` that implements native types defines them inside its 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:

```
// GraceNumber.h
class Equals : public NativeMethod {
public:
    virtual GraceObjectPtr respond(
      GraceObject &self,
      MethodRequest &request);
};

// GraceNumber.cpp
GraceObjectPtr GraceNumber::Equals::respond(
  GraceObject &self,
  MethodRequest &request)
  {
  if (self.asNumber().value()
      == request.params()[0]->asNumber().value()) {
      return GraceTrue;
  }
  return GraceFalse;
}
```

4.7. Object Model

Everything is an object in Grace, 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.

To represent this, a shallow but wide class hierarchy was used, with an abstract `GraceObject` class at the top and every other type of object implemented as a direct subclass of it.

4.7.1. 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,[8] 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);

    // ...
};
```

[8] http://gracelang.org/documents/grace-spec-0.7.0.html#method-names

4.7.2. 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 few methods and only one element in their type class. One such type is `Undefined`, which throws an error whenever the user tries to interact with it.

4.7.2.1. Blocks

Blocks are a particular case of native types in Naylang. They represent lambda functions that respond to an `apply()` method with a correct number of parameters. Therefore, a block will be represented as a `GraceBlock` with one user-defined method (`apply`) which will nave a variable number of parameters, and will simpy consume all the parameters available. The implementation of `apply` will represent the desired behavior of the lambda function.

4.7.3. Casting

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

```
// GraceObject.h

// Each of these methods will throw an exception when 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 invariants 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 to the AST as the possibility of proper static typing is implemented.

4.8. Memory Management

Grace is a garbage-collected language,[9] and therefore there must be some mechanism to automatically control memory consumption during the evaluation.

This section details such mechanisms, and their implementation and evolution throughout the development of Naylang.

4.8.1. Reference-counting

The first solution to this problem was to have reference-counted objects, so that when an object would be referenced by one of the objects in the subscopes of the evaluator they would remain in memory. 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 was be defined to create objects. With the help of C++ template metaprogramming, 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 memory management strategy has a number of fatal flaws, the worse of them being the *circular reference problem* (Jones et al., 2016). With 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.

4.8.2. Heap and ObjectFactory classes

The next step was to use one of the well-researched memory management algorithms (Jones et al., 2016). 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.

[9] http://gracelang.org/documents/grace-spec-0.7.0.html#garbage-collection

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.

4.8.3. 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 ExecutionEvaluator per programming session, and therefore it is reasonable that every instance of the evaluator will have an instance of the Heap.

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

```
class GraceObject {
protected:
    std::map<std::string, GraceObjectPtr> _fields;
    // ...

public:
    bool _accessible;
    // ...
```

```
    const std::map<GraceObjectPtr> &fields();
};
```

4.8.4. 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 (Jones et al., 2016), 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 depth-first marked 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 its memory is deallocated.

Since this implementation of the `Heap` only *simulates* the storage of the objects, and does not make claims about its continuity, heap fragmentation is handled by the underlying C++ implementations. Therefore, no strategy is needed at this level 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.

4.8.5. 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.

4.8.5.1. 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`[10]) is not necessary. Furthermore, it can be said that

[10] http://en.cppreference.com/w/cpp/container/map

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,[11] 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 ownsership* 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 (Kieras, 2016). The object that holds that reference 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.[12] 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`.

4.8.5.2. 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->_accessible = false;
    }
    auto scope = _eval->currentScope();
    scope->_accessible = true;
    visitMark(scope);
```

[11] http://en.cppreference.com/w/cpp/container/list
[12] http://en.cppreference.com/w/cpp/memory/unique_ptr

```
    for (auto&& obj = _storage.begin();
            obj != _storage.end();) {
        if (!(*obj)->_accessible) {
            obj = _storage.erase(obj);
        } else {
            ++obj;
        }
    }
}

void Heap::visitMark(GraceObject* scope) {
    for (auto field : obj->fields()) {
        if (field.first != "self" &&
                !field.second->_accessible) {
            field.second->_accessible = true;
            visitMark(field.second);
        }
    }
}
```

4.8.5.3. 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 `10` 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::triggerGCIfNeeded() {
    if (_storage.size() >= _capacity) {
        markAndSweep();
        if (_storage.size() >= _capacity) {
            throw std::string{"Heap: Out of Memory"};
        }
    }
```

```
    if (_storage.size() >= _threshold) {
        if (_nthObject == _triggerInterval) {
            _nthObject = 0;
            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, as is explained in Figure 4.8

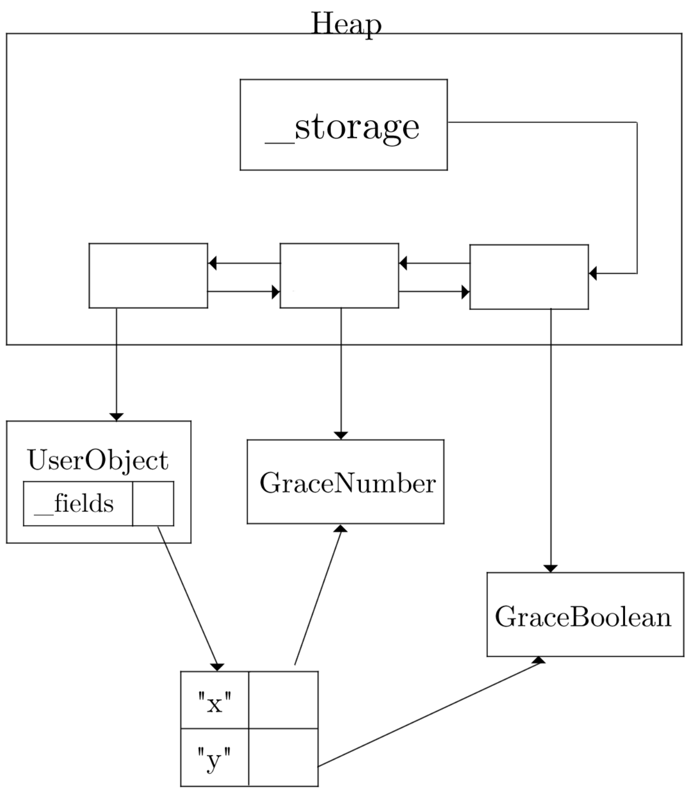

Figure 4.8.: Heap Storage Model

4.9. Debugging

As previously mentioned, Naylang implements a set of debug commands similar to that of GDB. More precisely, the set of commands whose functionality is replicated is `run`, `continue`, `next` (step over), `step` (step into), `break` and `print`. The list of commands and an explaination of their uses is listed in the Frontends section.

The debugging mechanisms described are implemented using the Modular Visitor Pattern. Specifically, since the debugger needs only to interject in the `ExecutionEvaluation` function calls, the Direct Subclass Pattern was used.

In addition to that, a controler was created (`Debugger`) to act as an adaption layer between the extended evaluatio and the frontend.

4.9.1. Before-After Stateful Debugging

The debugger uses a *before-after* stateful execution pattern. In general, the debugger behaves exactly the same as the `ExecutionEvaluator`, **except** for when a pause in the execution is required, in which case the execution must block and request commands until a command is provided that resumes execution (e.g. `continue` or `next`). A pause can happen either because a breakpoint is reached, or the execution was paused in the instruction before and a step instruction was executed (e.g. `step` will execute an instruction and block again).

The extension of the evaluation must only handle the cases where a pause is necessary. In these cases two calls are added before and after the call to the regular evaluation. Either function can block if the conditions demand so. When they do, they request commands from the frontend until the conditions are met to resume exection.

```
void DebugEvaluator::evaluate(VariableDeclaration &expression) {
    // Call to the debug mechanism
    beginDebug(expression);
    DebugState prevState = _state;
    // Call superclass to handle regular evaluation
    ExecutionEvaluator::evaluate(expression);
    // Call to the debug mechanism
    endDebug(expression, prevState);
}
```

To handle all the possible cases and commands, the debugger holds a `state` field, which determines the behavior of a certain `<begin/end>debug()` call. Therefore, the `<begin/end>debug()` functions are also resposible for handling automatic state transitions in the debugger, that is, transitions that do not require user interaction. The possible debug states are the following:

```
enum DebugState {
    CONTINUE,
    STOP,
```

```
    STEP_IN,
    STEP_OVER,
    STEP_OVER_SKIP
};
```

And the debug functions handle a relatively small set of cases:

```
void DebugEvaluator::beginDebug(Statement *node) {
    if (_state == STEP_OVER)
        _state = CONTINUE;
    _debugger->debug(node);
}

void DebugEvaluator::endDebug(Statement *node, DebugState prevState) {
    if (!node->stoppable())
        return;
    if (prevState == STEP_OVER)
        _state = STOP;
    if (_state == STEP_IN)
        _state = STOP;
}
```

The state can also be changed with external commands such as `continue`, which changes the state unconditionally to `CONTINUE`, or by the controller for diverse causes, such as a breakpoint being reached.

4.9.2. Debugger Class

The `Debugger` class can be thought of as the controller for the `DebugEvaluator`. It is responsible for:

- Handling user-defined breakpoints. In this case, the breakpoints are only a set of lines in which a breakpoint is set.
- Implementing the `debug()` function which the `DebugEvaluator` calls to update its state.
- Implementing auxuliary public functions that correspond with the different debug commands (e.g. `run()`, `continue()`).
- Interfacing with the execution mode (and therefore the frontend) to output information and request additional commands when necessary.

```
class Debugger : public Interpreter {
    GraceAST _AST;
    std::set<int> _breakpoints;
    DebugMode *_frontend;
public:
    // Functions to be used by DebugCommands
    void run();
    void setBreakpoint(int line);
    void printEnvironment();
    void resume();
```

```
    void stepIn();
    void stepOver();

    // Called from the Debugger
    void debug(Statement *node);
};
```

4.10. Frontend

One of the design goals of Naylang is to serve as a teaching example in interpreter construction. This requires that the execution core (parsing, AST and evaluation) be as isolated as possible from the interaction with the user, with aims to help the student in discerning the fundamental 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 its `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 interpreter. Each mode has its own commands, which are implemented using the Command pattern (Gamma et al., 1995). 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
```

Figure 4.9 displays the main class structure that allows for such a command flexibility. Since the frontends are not the main focus of the projects not many more explainations are given, but more than usual information is provided such that it should be apparent how such structure could work.

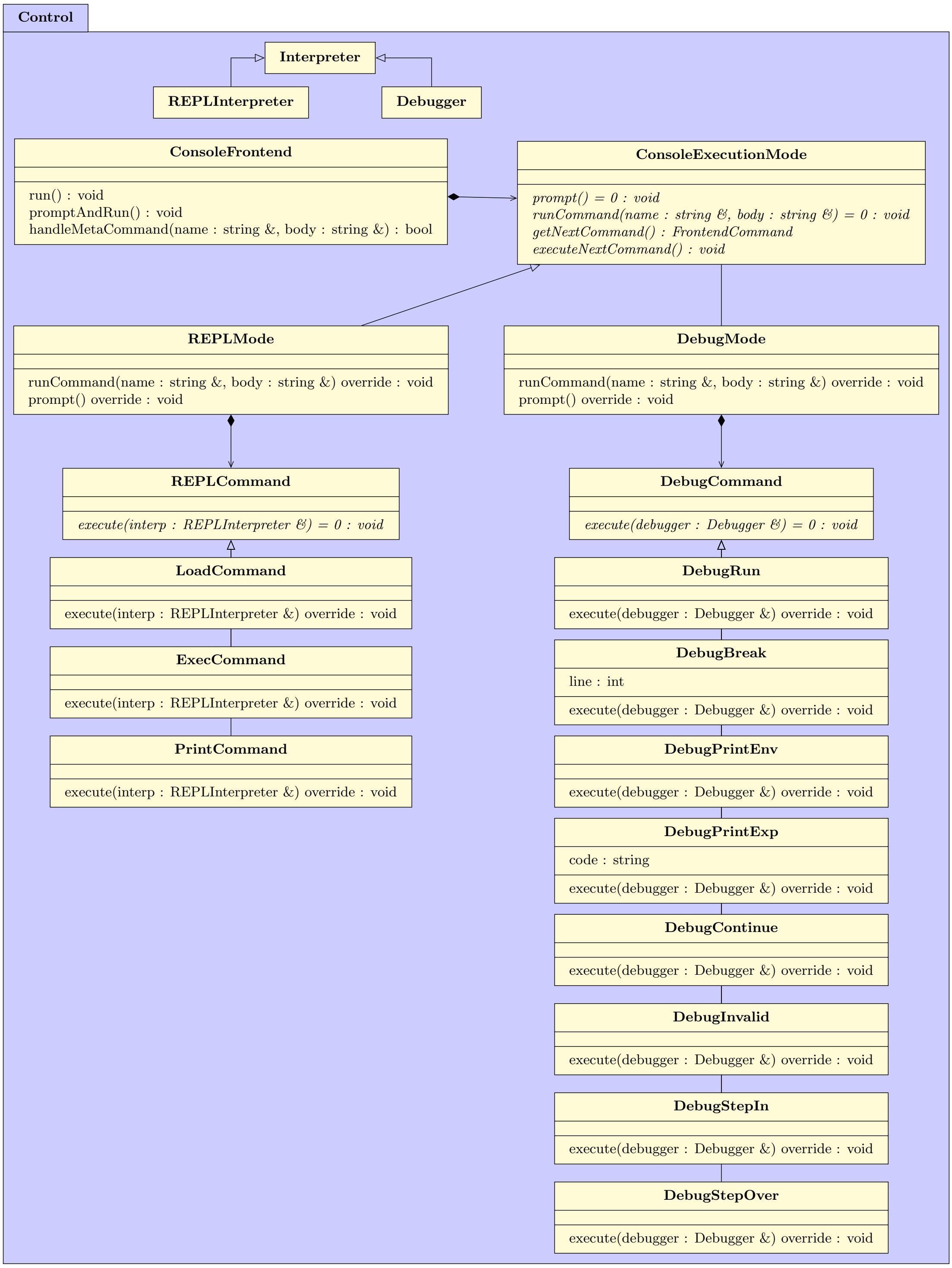

Figure 4.9.: Frontends And Controllers

5. Modular Visitor Pattern

During the development of the Naylang debugger, the need arose 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 intertwined, 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 refactoring process started.

During the refactoring process, a new programming pattern arose. This new pattern allowed for the development of completely separate processing engines, each with its 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.

5.1. Description

This pattern takes advantage of the very structure of Visitor-based interpreters. In this model of computation, every node in the AST has an `Evaluator` method 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 the intervention points and the extra control over the execution flow and insert arbitrary code in those locations. This code pieces could potentially do anything, from pausing the normal evaluation flow (e.g. in a debugger) to modifying the AST itself, potentially 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.

5.1.1. 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 (Liskov and Wing, 1994).

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.

Figure 5.1 demonstrates the use of this specific technique. In this case, the instantiation of the visitors would be as follows:

```
proc createExtensionVisitor() {
    return new ExtensionVisitor();
}
```

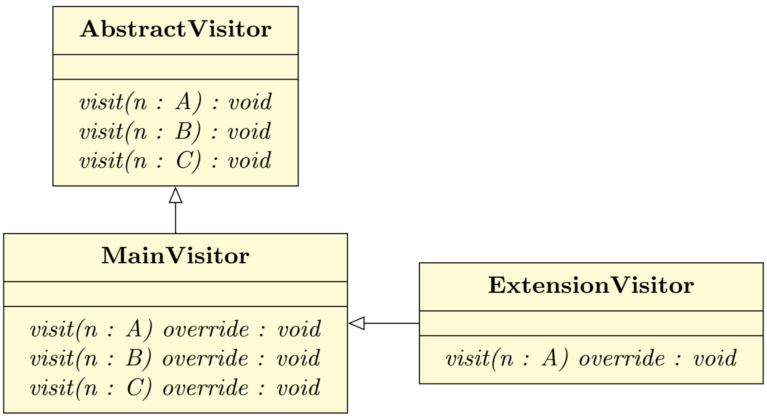

Figure 5.1.: Direct Subclass Modular Visitor Pattern

5.1.1.1. 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 its 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);
}
```

5.1.1.2. 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.

5.1.2. 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 (Gamma et al., 1995).

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.

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.

Figure 5.2 demonstrates an implementation of this pattern. In this case, the instantiation of the extension is as follows:

```
proc createExtensionVisitor() {
    super := new MainVisitor();
    return new ExtensionVisitor(super);
}
```

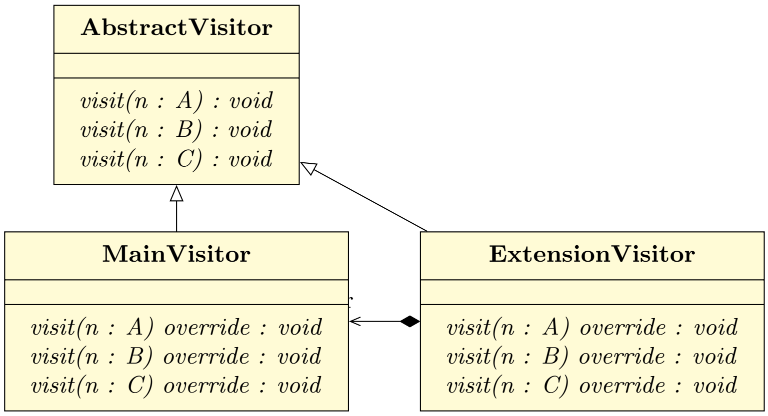

Figure 5.2.: Composite Modular Visitor Patern

5.1.2.1. 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);
}
```

5.1.2.2. Discussion

The Composite Modularity method simplifies greatly the class hierarchy by moving the composition of visitors from the subclassing mechanism to runtime instantiation, creating wider, more shallow class hierarchies. 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* (Gamma et al., 1995)).

This problem can be circunvented by having the extender class explicitly create the instances of the visitors it nedds directly into its 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.

5.1.3. 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 intercept 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.

Figure 5.3 demonstrates an implementation of this pattern. In this case, the instantiation of the extension is as follows:

```
proc createExtensionVisitor() {
    super := new MainVisitor();
    return new ExtensionVisitorA(super);
}
```

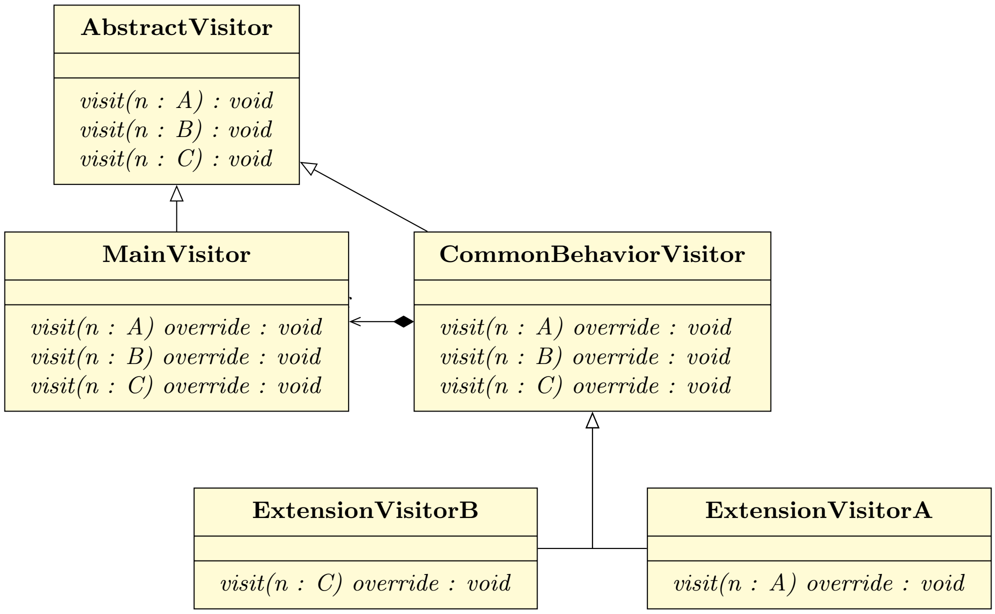

Figure 5.3.: Wrapper Superclass Modular Pattern

5.1.3.1. Example

Following the previous example, it is possible to define a superclass that bundles 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 {
```

```
        before(&expression);
        _super->evaluate(expression);
        after(&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() override;
    virtual void after() override;

    virtual evaluate(VariableDeclaration &expression) override;
}

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

5.1.3.2. 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.

5.2. Applications

This visitor design pattern has a myriad of applications. The main benefit is that it allows to extend the functionality of an intepreting engine without needing to change the previous processings. It permits the addition of 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 research 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 its 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.

As mentioned previously, this idea of a fully modular language has been developed in several academic works where the use of monads was suggested (Sierra, 2004). 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 made a thousand-line project manageable.

6.1. Tests as an educational resource

Naylang aims to be more than just a Grace interpreter, but to also be an approachable Free Software[1] project for both potential collaborators and programming language students. 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 how it 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 its 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 (Beck, 2003).
- 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

[1] https://www.fsf.org/about/what-is-free-software

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.

6.2. 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 crucial part of the codebase will by default have an extensive test coverage.

TDD may feel slow at first, but as the project grew 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 (Beck, 2003).

6.3. The Framework

Naylang is a relatively small (less than 10.000 lines of code), single-threaded 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()s, 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, such as 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().

6.4. 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 expressions and no reciever") {
```

```
        REQUIRE_NOTHROW(ImplicitRequestNode req("myMethod", {five}););
    }

    SECTION("An ImplicitRequestNode with an empty parameter list"+
            "can request variable values or parametereless methods") {
        REQUIRE_NOTHROW(ImplicitRequestNode variableReq("x"););
        REQUIRE_NOTHROW(ImplicitRequestNode methodReq("myMethod"););
    }

    // Accessor sections
    SECTION("A ImplicitRequestNode can return the identifier name"+
            "and parameter expressions") {
        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 (Martin, 2009). Therefore, these two types of tests are sufficient.

6.5. Testing the Evaluation

The `ExecutionEvaluator` 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 its internal state. This problem required the `ExecutionEvaluator` to be able to make queries about its state, and modify it (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 (e.g. `Requests`), which can be either field or method requests.

Therefore, the `ExecutionEvaluator` test file contains several `TEST_CASE()`s:

- **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.

6.6. 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 its 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);
    }
}

// ...
}
```

6.7. 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[2] and assertions to validate the structure of the tree.

```
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".

6.8. Integration testing

To test whether particular features of the language fit inside the whole of the project, a series of integration tests were developed. These tests are comprised of

[2] http://en.cppreference.com/w/cpp/language/static_cast

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.

6.9. 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 its 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 and Future Work

Having reached the end of the development period for this project, it is necessary to review the results obtained and compare them with the proposed objectives.

This chapter explains the main challenges faced when implementing Naylang, a review of which goals were accomplished, which were not, and a brief summary of future work that would move the project forward.

7.1. Challenges

This section details the main roadblocks for the development of Naylang. Fortunately, many of these roadblocks were overcomed and served as a learning experience.

7.1.1. Modern C++

The language chosen for this project was modern C++ (C++14). Having worked with previous versions of C++ (C++98) extensively before, it seemed that this language choice was the best one. However, the newer versions of C++ proved to be substantially different from the older ones, which introduced a great deal of additional difficulty to the development cycle as the new features had to be learned in parallel to implementing the code. Often, a wrong choice of feature (such as using owning pointers where shared pointers were due) meant that a substantial part of the codebase had to be rewritten or reaconditioned to use the new class.

As a result, more than half of this project's debugging time was spent wrestling with the new features instead of fixing actual bugs.

7.1.2. Abstract Representation

The Grace specification offers very sparce information on the desired behavior of certain operations (such as the assignment operator), specially with regards to their structure and their place in the syntax. That being the case, forming a representation of the Abstract Syntax Tree required several iterations and a great deal of guesswork.

For instance, the first approach was to introduce arithmetic and logic operators explicitly to the abstract syntax, which had to later be discarded in favor of the current *request*-based approach.

Needless to say, these iterations proved to be very costly on development time, since rewriting the entire abstract representation is a simple but long process, specially when the tests had also to be rewritten.

7.1.3. Requests and Method Dispatch Model

This issue ties with the previos one in that it results from the particularities of the Grace language. Since methods are part of objects and can contain either custom Grace code or native code, the closures and structure of method definitions and requests was difficult to implement. Luckily, extensive research of Kernan facilitated a starting point for the architecture, but it was nevertheless a long iterative process until a complete solution was found.

7.1.4. Debugger Decoupling

The problem of integrating debugging mechanisms in Naylang *without* modifying the core evaluation model led to some research on the field and, eventually, the Modular Visitor Pattern described earlier.

7.2. Goal review

Following is a review of the goals described in the introduction, detailing which ones were achieved, and which ones were not.

7.2.1. Implementation Goals

Naylang set out to be an interpreter and debugger for a *subset* of Grace, enough to teach the basic concepts of Computer Science to inexperienced students.

While it is indeed a fully-fledged debugger and it accepts a substantial subset of Grace, many important features of the language were left out (such as the type system), which limits what a novice can achieve with the language.

7.2.2. Education Goals

The other key goal of Naylang was to be **approachable** to any student learning about language implementation or any future contributors to the project. In this objective Naylang has excelled, featuring extensive and descriptive test coverage that acts as documentation for the project, and great modularity in its components.

7.3. Future work

Even though the work done in Naylang was fairly satisfactory, there are still many areas that could be greatly improved with future work. The completion of these tasks would make Naylang a useful tool for Computer Science education.

7.3.1. Modular Visitor

The Modular Visitor Pattern is probably the area that deserves the most attention in further developments, since it shows the potential to introduce **great flexibility** in the development of intepreters, and even new languages. If the potential it shows is fulfilled, even the development of custom 'à la carte' languages would become a much easier task, accomplished by recombining evaluation modules developed by different third parties.

7.3.2. Language features

Many of the features of Grace were left unimplemented in Naylang. While Naylang will not strive for feature-completeness in Grace, it should implement some of its most important features for education, such as the class and type systems.

However, by not embedding these features directly into the core evaluation, the possibility arises to **use Naylang as a research project** for the viability of the Modular Visitor Pattern, as the new language features can be added using it.

7.3.3. Web Frontend

One of the faults in Naylang's use in an educational setting is the **distribution of the executables** to target users. For novice programmers, the source compilation process and the unfriendly interface could result discouraging at first.

A possible solution to that problem would be to get rid of distributing executables altogether, and have a web-based interface to interact with Naylang from any browser. Some early work has been done with promising results, but due to time constraints the development of this interface was left out of the scope of the project.

Bibliography

Abelson, H., Sussman, G.J. and Sussman, J. (1996), *Structure and Interpretation of Computer Programs*, Justin Kelly.

Abrahams, D. and Gurtovoy, A. (2004), *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Portable Documents*, Pearson Education.

Aho, A.V., Sethi, R. and Ullman, J.D. (1986), *Compilers, Principles, Techniques*, Addison wesley Boston.

Beck, K. (2003), *Test-Driven Development: By Example*, Addison-Wesley Professional.

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.

Black, A.P., Bruce, K.B., Homer, M., Noble, J., Ruskin, A. and Yannow, R. (2013), "Seeking grace: A new object-oriented language for novices", in *Proceeding of the 44th Acm Technical Symposium on Computer Science Education*, ACM, pp. 129–134.

Duggan, D. (2000), "A mixin-based, semantics-based approach to reusing domain-specific programming languages", in *European Conference on Object-Oriented Programming*, Springer, pp. 179–200.

Espinosa, D.A. (1995), *Semantic Lego*, PhD thesis, Columbia University New York, NY, USA.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

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).

Hejlsberg, A., Wiltamuth, S. and Golde, P. (2003), *C# Language Specification*, Addison-Wesley Longman Publishing Co., Inc.

Homer, M. (2014), "Graceful language extensions and interfaces", Victoria University of Wellington.

Jones, R., Hosking, A. and Moss, E. (2016), *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press.

Kieras, D. (2016), "Using c++11's smart pointers", available at: http://www.

umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf (accessed 22 May 2017).

Liskov, B.H. and Wing, J.M. (1994), "A behavioral notion of subtyping", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, Vol. 16 No. 6, pp. 1811–1841.

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).

Martin, R.C. (2009), *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson Education.

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).

Noble, J., Homer, M., Bruce, K.B. and Black, A.P. (2013), "Designing grace: Can an introductory programming language support the teaching of software engineering?", in *Software Engineering Education and Training (Csee&T), 2013 Ieee 26th Conference on*, IEEE, pp. 219–228.

Parr, T. (2009), *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Pragmatic Bookshelf.

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

Sierra, J. (2004), *Hacia Un Paradigma Documental de Desarrollo de Aplicaciones*, PhD thesis, Tesis doctoral Universidad Complutense de Madrid.

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. Introducción

Naylang es un intérprete REPL, entorno de ejecución y depurador para el lenguaje de programación Grace, implementado totalmente en C++14.

Actualmente, implementa un subconjunto de Grace (descrito a continuación), pero a medida que el proyecto evolucione tenderá hacia una mayor compatibilidad con el lenguaje.

A.1. Motivación

Grace es un lenguaje diseñado para ayudar a nuevos estudiantes a adquirir los conceptos fundamentales de la programación. Como tal, provee de seguridad y flexibilidad en su diseño.

Sin embargo, el coste de esta flexibilidad es que muchas de las actuales implemetaciones de Grace son opacas y difíciles de abordar. Grace es un lenguaje abierto, y por lo tanto sus implementaciones también son abiertas. Esta falta de claridad en la implementación hace que la apertura de su código se devalúe ya que, aunque las posibles entidades contribuyentes tengan acceso al código fuente, éste es difícil de entender y por supuesto de modificar, dañando severamente las oportunidades de crecimiento y expansión del lenguaje.

A.2. Objetivos

Naylang tiene como primer objectivo servir como ejercicio en la construcción de intérpretes de lenguajes, tanto para los creadores, como para cualquier futuro contribuyente al código. Como consecuencia, el proyecto presenta los siguientes objetivos primordiales:

- Proveer una implementación **sólida** de un **subconjunto relevante** de Grace.
- Ser tan **amigable** como sea posible para los **usuarios finales** (estudiantes de programación) y para posibles futuras **contribuciones**.
- Ser en sí misma una **herramienta** para **aprender sobre la implementación** de lenguajes tan flexibles como Grace.

A.3. Metodología

El proyecto se rige por la disciplina del Desarrollo Basado en Tests (TDD), por la cual tests unitarios se escriben en paralelo al código (muchas veces antes que éste), en iteraciones muy cortas. Se ha elegido este modelo de desarrollo por varias razones:

- En primer lugar, contar con una cobertura extensa de tests provee una forma fácil y automática de verificar qué partes del proyecto están funcionando como deberían. Por lo tanto, nuevos contribuyentes al proyecto sabrán con exactitud qué subsistemas afectan los cambios que hagan y de qué forma, lo que permitirá hacer cambios con mayor rapidez y seguridad.
- En segundo lugar, los tests unitarios en sí mismos sirven también como *documentación* del proyecto, dado que proveen **ejemplos funcionales** del uso de cada parte del código. Esto resulta en una facilidad mucho mayor a la hora de entender las interacciones y el funcionamiento de los diferentes subsistemas. Como beneficio añadido, los tests unitarios se mantienen por defecto siempre actualizados con el código, por lo que no es necesario redactar una documentación por separado.

El desarrollo de Naylang será llevado a cabo en iteraciones cortas, muchas veces de menos de una semana de duración. El objetivo es explorar las diferentes arquitecturas de las posibles soluciones a los problemas presentados por la construcción de intérpretes. Así, se consigue maximizar el beneficio que brinda la completa cobertura de tests interando sobre los diseños sin miedo a una regresión en la funcionalidad.

A.4. Compromisos

Dado que Naylang está diseñado como un caso de estudio, la claridad en el código y las buenas prácticas toman precedencia sobre la eficiencia a la hora de hacer decisiones de implementación. En concreto, si existe una implementación simple y robusta para algún componente ésta tomará predecencia por encima de otras más eficientes pero más oscuras.

Sin embargo, el diseño modular, desacoplado y robusto resultante de la disciplina TDD hace que sea relativamente sencillo para futuras contribuciones intercambiar uno de los componentes menos eficientes por una implementación más eficiente con funcionalidad similar.

En resumen, este proyecto pretende optimizar sus decisiones para maximizar su **claridad** y **extensibilidad**, en lugar de maximizar parámetros como **tiempo de ejecución** o **uso de memoria**.

B. Conclusión

Habiendo llegado al final del periodo de desarrollo para este projecto, es necesario revisar el resultado y compararlo con los objetivos propuestos.

Este capítulo explica las principales dificultades encontradas a la hora de implementar Naylang, una revisión de qué objetivos fueron cumplidos, cuales no, y un breve sumario de las posibles vías de trabajo futuro.

B.1. Desafíos

Esta sección detalla los principales obstáculos para el desarrollo de Naylang. Afortunadamente, muchos de estos obstáculos fueron superados, y sirvieron como experiencias de aprendizaje.

B.1.1. C++ Moderno

El lenguaje elegido para este proyecto fue la última versión estable de C++ (C++14). Habiendo trabajado extensamente con otras versiones de C++, esta elección de lenguaje parecía ser la mejor. Sin embargo, las nuevas versiones de C++ resultaron ser substancialmente diferentes a las anteriores, con una miríada de funcionalidades vitales para el correcto uso de éstas. Esto introdujo un alto grado de dificultad adicional al desarrollo del proyecto, ya que las nuevas funcionalidades debían ser estudiadas al mismo tiempo que se desarrollaba el proyecto. El resultado fue que grandes partes del código tuvieron que ser reescritas más de una vez, a medida que se descubrían mejores formas de enfocar el problema.

Como resultado, más de la mitad del tiempo de depuración de este proyecto se usó intentando integrar estas nuevas funcionalidades, en lugar de arreglando fallos del Naylang.

B.1.2. Representación Abstracta

La especificación de Grace ofrece información limitada sobre el comportamiento deseado de ciertas operaciones (como la asignación), especialmente en lo que respecta a su estructura y representación abstracta. Siendo éste el caso, el diseño del Arbol de Sintaxis Abstracta requirió muchas iteraciones y un alto grado de interpretación de la especificación.

Por ejemplo, una de las primeras aproximaciones fue introducir operadores lógicos y aritméticos explícitamente en la sintaxis abstracta, lo que se tuvo que descartar más adelante cuando se descubrió el modelo de ejecución basado en *requests*.

Estas iteraciones sobre la representación abstracta probaron ser sencillas pero muy costosas en tiempo de desarrollo, dado que modificar el banco de tests y el código principal son operaciones extremadamente tediosas.

B.1.3. Modelo de Dispatch y Requests

Este problema está asociado al anterior en tanto en cuanto a que resulta de las particularidades de Grace. Dado que los métodos son una parte integral de los objetos en Grace y pueden contener tanto código arbitrario o funcionalidad predefinida, el modelo de ejecución y dispatch presentó un gran desafío. De hecho, la funcionalidad de dispatch y ejecución de métodos está repartida en al menos tres subsistemas.

B.1.4. Depurador Desacoplado

El problema de integrar la depuración en Naylang *sin modificar* el motor básico de evaluación llevó a cierto grado de investigación y, finalmente, al desarrollo del Patrón del Visitante Modular descrito anteriormente.

B.2. Revisión de Objetivos

Esta sección incluye una evaluación de los objetivos impuestos en la introducción, detallando cuáles han sido conseguidos y cuáles no.

B.2.1. Objetivos de Implementación

Naylang tenía la intención de ser un intérprete y depurador para un *subconjunto* de Grace suficientemente extenso como para poder enseñar los conceptos básicos de la Informática a estudiantes totalmente nuevos en la materia.

Mientras que es, de hecho, un depurador muy potente e implementa un subconjunto substancial de Grace, muchas de las características importantes del lenguaje fueron dejadas a un lado en Naylang, limitando lo que un estudiante puede aprender sobre programación.

B.2.2. Objetivos de Educación

El segundo objetivo vital de Naylang era ser **amigable** para cualquier estudiante interesado en aprender sobre implementación de lenguajes o para cualquier futuro contribuyente al proyecto. En este aspecto Naylang ha sido un éxito, ya que cuenta con una extensa cobertura de tests, lo que proporciona cientos de casos de uso y una gran modularidad en sus componentes.

B.3. Trabajo Futuro

Aunque el trabajo realizado en Naylang haya sido razonablemente satisfactorio, aún hay muchas áreas que podrían beneficiarse de trabajo futuro. Completar estas tareas haría de Naylang una herramienta útil para la educación en Informática.

B.3.1. Visitante Modular

Probablemente el área que merece mayor atención en futuros desarrollos, con el potencial de introducir **gran flexibilidad** en el desarrollo de intérpretes, e incluso nuevos lenguajes. El desarrollo de lenguajes con caracterísitcas desacopladas se convertiría en un trabajo mucho más sencillo, por la recombinación de módulos funcionales desarrollados de forma independiente entre sí.

B.3.2. Funciones del Lenguaje

Muchas de las características de Grace fueron dejadas aparte en Naylang. Mientras que ya no pretenderá implementar todas estas características, debería implementar algunas de las áreas más necesarias para la educación, como el sistema de clases y tipos.

Sin embargo, al no incluir estas áreas directamente en el núcleo de evaluación, surge la posibilidad de **usar Naylang como proyecto de investigación** para estudiar la viabilidad del Patrón del Visitante Modular, usándolo para implementar nuevas características.

B.3.3. Frontend Web

Una de las fallas en el uso de Naylang en un entorno educativo es la distribución de binarios ejecutables a los usuarios finales. Para programadores inexpertos, la instalación y la interfaz podrían resultar poco amigables en un principio.

Una solución a este problema sería descartar el modelo de ejecución local y tener una interfaz web para interactuar con Naylang desde cualquier navegador. Aunque cierto trabajo ha sido realizado con resultados prometedores, el desarrollo de esta interfaz fue descartado del proyecto por razones de tiempo.

C. Grace Grammars

ANTLR 4 grammars used for parsing Grace in Naylang.

C.1. Lexer Grammar

The grammar used to generate the string tokenizer:

```
lexer grammar GraceLexer;

tokens {
    DUMMY
}

WS : [ \r\t\n]+ -> channel(HIDDEN);
INT: Digit+;
Digit: [0-9];

METHOD: 'method ';
VAR_ASSIGN: ':=';
VAR: 'var ';
DEF: 'def ';
PREFIX: 'prefix';
OBJECT: 'object';
IF: 'if';
ELSE: 'else';
WHILE: 'while';

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: '=';
LESS: '<';
LESS_EQUAL: '<=';
GREATER: '>';
GREATER_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];
```

C.2. Parser Grammar

The grammar used to generate the AST constructor:

```
parser grammar GraceParser;
options {
    tokenVocab = GraceLexer;
}
program: (statement)*;
statement: expression DELIMITER
         | declaration
         | assignment
         | control;

assignment : field=identifier VAR_ASSIGN val=expression
             DELIMITER #SelfAssign
           | scope=explicitRequest
             DOT field=identifier
             VAR_ASSIGN val=expression DELIMITER #ExplAssign
           | scope=implicitRequest
             DOT field=identifier
             VAR_ASSIGN val=expression DELIMITER #ImplAssign
           ;

control : ifThen
```

```
        | ifThenElse
        | whileNode
        ;

ifThen : IF OPEN_PAREN cond=expression
         CLOSE_PAREN thn=methodBody;
ifThenElse :
        IF OPEN_PAREN cond=expression
        CLOSE_PAREN thn=methodBody ELSE els=methodBody;
whileNode:
    WHILE OPEN_BRACE cond=expression CLOSE_BRACE body=methodBody;

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 DELIMITER
                | control
                | assignment
                ;

// Using left-recursion and implicit operator precendence.
// ANTLR 4 Reference, page 70
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 #InfixExp
          | value #ValueExp
          ;

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

implicitRequest : multipartRequest              #MethImplReq
             | identifier effectiveParameter #OneParamImplReq
             | identifier                  #IdentifierImplReq
                  ;
multipartRequest: methodRequestPart+;
methodRequestPart:
        methodIdentifier OPEN_PAREN effectiveParameterList?
        CLOSE_PAREN;
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=methodBodyLine* CLOSE_BRACE;
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
        | LESS
        | LESS_EQUAL
        | GREATER
        | GREATER_EQUAL
        | EQUAL EQUAL
        | EXCLAMATION EQUAL
        ;
```

D. How was this document made?

D.1. Author

The process described in this Appendix was devised by Álvaro Bermejo, who published it under the MIT license in 2017[1]. What follows is a verbatim copy of the original.

D.2. 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.

D.3. 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`.

D.4. 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

[1] https://github.com/AlvarBer/Persimmon/tree/master/docs