Naylang

A REPL interpreter and debugger for the Grace programming language.

Borja Lorente

Director: José Luis Sierra Rodríguez

Abstract

A REPL interpreter and debugger for the Grace programming language.

Keywords: Intepreters,Programming Languages,Debuggers,Grace .

Contents

List of Figures

1. Introduction

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

Motivation

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

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

Objectives and Methodology

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

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

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

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

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

Tradeoffs

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

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

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

2. The Grace Programming Language

Introduction

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

Key Features

Support for multiple teaching paradigms

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

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

Safety and flexibility

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

Gradual typing

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

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

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

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

Multi-part method signatures

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

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

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

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

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

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

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

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

Lexically scoped, single namespace

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

Lineups

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

Object-based inheritance

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

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

Subset of Grace in a Page

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

// TODO: Write subsection

3. State of the art

Minigrace

Kernan

TreeGraph

JavaScript interpreters

JavaScript is similar to a fully dynamically typed version of Grace (our subset).

4. Implementation

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

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

//TODO Class diagram []

Project Structure

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

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

Sources

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

development of other frontends, such as graphical interfaces. The `core` folder is structured as follows:

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

Tests

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

Grammars and examples

There are two Grace-specific folders in the project:

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

Build tools

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

- `cmake` contains the CMake file bundled with the C++ target, which drives the compilation and linking of the ANTLR runtime. It has been slightly modified to compile a local copy instead of a remote one (Lorente, 2017).
- `thirdparty/antlr` contains two major components:

- A frozen copy of the ANTLRv4 runtime in the 4.7 version , `antlr-4.7-complete.jar` (Parr, 2017), to be compiled and linked against.
- The ANTLRv4 tool, `antlr-4.7-complete.jar`, which is executed by a macro in the CMake file described earlier to generate the parser and lexer classes. Obviously, this is also in the 4.7 version of ANTLR.

Lexing and Parsing

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

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

The Naylang Parser Visitor

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

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

The Naylang Parser Stack

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

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

- It must hold references to Statement nodes.

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

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

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

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

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

Left-Recursion and Operator Precedence

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

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

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

As can be seen, the precedence is clearly defined and expressed where it matters the most (the first two lines). Grace's specification does not define a precedence

for any other type of expression, so the rest is left to the implementer.

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

Abstract Syntax Tree

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

// TODO: add class diagram []

The design of the AST is subject to change as new features are implemented in the interpreter.

Statement Nodes

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

Declaration Nodes

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

- Constant and Variable Declarations represent the desire to create fields inside an object, and hold an expression with their initial value. They are also *breakable statements* (see Debugging).
- Method declarations represent a subroutine inside Grace, which contain an arbitrary-length list of executable Statements, which will be executed every time the method is called.

Expression Nodes and Requests

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

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

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.
- 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).

Following are some examples of different code snippets, and how they will be translated into nested Requests (for brevity, we will use IR and ER 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.

Object and Execution Model

Built-in methods and Prelude

Heap and Garbage Collection

Debugging

Frontend

Bibliography

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

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

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

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

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

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

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

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

A. Appendix A: Grammars

ANTLR 4 grammars used for parsing Grace in Naylang.

Lexer Grammar

```
lexer grammar GraceLexer;
tokens {
    DUMMY
}

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

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

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

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

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

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

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

Parser Grammar

```
  parser grammar GraceParser;

  options {
      tokenVocab = GraceLexer;
  }

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

  declaration : variableDeclaration
              | constantDeclaration
              | methodDeclaration
              ;

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

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

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

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

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

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

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

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

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

  primitive   : number
              | boolean
              | string
              ;

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

B. Appendix B: How was this document made?

Author

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

Process

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

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

Diagrams

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

References

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