The Gryph Programming Language

Vinícius Campos Artur Curinga Vitor Greati Carlos Vieira

June 17, 2018

Contents

1 Introduction

Gryph is a programming language designed and developed for graph-oriented programming. It is easy to learn and to program with, being a relatively abstract scripting language, with many built-in high-level data structures. Gryph's static typing, along with its interpreted nature, makes it ideal for prototyping fast and securely. But most of all, it is the ease of constructing and manipulating graphs it offers that sets it apart from other programming languages.

One domain in which Gryph may perform best is in the teaching and learning of common graph algorithms, where constructing the necessary data structures may get in the way of implementing such algorithms, and of observing exactly how they work. And yet, it is not limited to a learning context, being powerful enough to be used for implementing solutions to most problems of non-critical performance demands.

This document covers, firstly, the basic functionalities of the language (2), in terms of both syntax and semantics; then it expounds on the first, and so far only, implementation of Gryph (3); and finally it presents the Gryph syntax in EBNF (**??**). Being an actively developed project, parts of this document may be inaccurate or outdated: in case you detect any inconsistencies, we ask that you contact the authors of this document or raise an issue in the official Gryph repository.

2 Usage

2.1 Structure

Being a scripting language, a Gryph program is nothing but a collection of statements, which are executed sequentially. The following is a valid Gryph program (the `println` statement is defined in 2.4.1.2).

```
1 # print hello world to standard output
2 println "Hello World";
```

We delineate in this section particular constructs which form the general structure of a Gryph program, besides regular statements such as `println`.

2.1.1 Comments

The previously listed code already demonstrates the first of such constructs in its first line: comments, text in the source code ignored by the interpreter. The beginning of a comment is marked by the # character, and it ends on a line break. A comment does not have to begin at the beginning of a line; it can follow a valid statement, for instance. Also, since the # character inside a comment is not even parsed by the interpreter, multiple # characters written sequentially have no effect different from any other comment.

2.1.2 Subprogram declarations

Another important part of a Gryph program are subprograms, essential tools for process abstraction. To declare a subprogram, the `sub` keyword is used, followed by a valid identifier (see 2.1.4) for that subprogram. That is followed, in turn, by a list of parameters

enclosed in parentheses, the return type (if there is one), and then finally the body of the subprogram, enclosed in braces.

```
1 sub hello() : string {
2     return "hello";
3 }
4
5 sub my_print(s : string = "") {
6     println s;
7 }
```

In the above code, we define two subprograms. The first, `hello`, receives no argument, and returns a "`string`" (further information on types in 2.2). Note the `return` keyword in line 2, which signals the function to return the expression following it (see 2.4.1.4). The second, `my_print`, receives '`s`' as an argument, a `string`, and returns nothing. The parameter list is nothing more than a sequence of variable declarations (see 2.4.1.1) separated by semicolons, the initialization syntax being used to define a default value, making the parameter optional: in `my_print`, the parameter `s` has the empty string as its default value.

Arguments are generally passed to subprograms by copy, except if the parameter in question is marked by the ampersand character, in which case it is passed by reference. This is the only thing that differs a parameter declaration from a variable declaration, as variables cannot be marked in such way. In the following code, the function `add` adds to the argument passed to `lhs` the value of the `rhs` argument.

```
1 sub add(lhs : int&; rhs: int) {
2     lhs = lhs + rhs;
3 }
```

The expression used to call a subprogram consists of the subprogram identifier, followed by parentheses containing the arguments being passed to the subprogram. Such an expression can be used as a statement in itself or as part of another statement. In the following code, we call the `my_print` procedure, passing the return value of the `hello` function as an argument.

```
1 my_print(hello());
```

2.1.3 Data type declarations

Another part of a Gryph program, is the declaration of user-defined data types: records, more specifically. The declaration of a record starts with a type identifier, which follows rules similar to other identifiers (see 2.1.4), except that it must begin with an *uppercase* alphabetic character, for code clarity.

Following the identifier, we have, enclosed in braces, a sequence of field declarations, defining the contents of a variable of that type. A field declaration is much like a variable declaration, and the initialization syntax is used in much the same way as in a subprogram parameter, as it defines a default value for a new variable of the type being defined.

In the code below, we define the type "`Person`", which contains an age (of type `int`), and a name (of type `string`). We give 0 as a default age for a new variable of the type `Person`. The declaration and use of variables of user-defined types is explained in 2.2.3.

```
Person {
    name : string;
    age : int = 0;
}
```

2.1.4 Identifiers and scope

We have mentioned, in the previous section, the concept of an identifier. An identifier is, simply, a name given to a subprogram or a variable. An identifier can be any combination of alphanumeric characters and the underscore character, as long as it starts with an alphabetic character.

2.1.5 Importing files

A Gryph program can be separated into multiple files through the use of the keyword `use`. It essentially indicates the insertion of source code from another file into that part of the code. For that, `use` must be followed by the path (relative to the interpreter) to the file one wishes to include, enclosed in quotes. Say we define the following function in a file named "`file1.gph`":

```
sub f() : string {
    return "hello from file1";
}
```

We can use this function in another file as follows, considering for instance that `file1.gph` is in the same directory as the interpreter used to run the following code:

```
use "file1.gph"

println f();
```

2.2 Types

The Gryph language is statically typed, meaning the type of all variables is know at "compile time", before beginning the execution. Also, most variables, all subprogram parameters, and all fields in user-defined data types are explicitly typed. In this section, we discuss all possible types in the Gryph language, their use, and their form: built-in types, both primitive (2.2.1), and composite (2.2.2); as well as user-defined types (2.2.3).

2.2.1 Primitive types

Primitive types in Gryph are much the same as in most other programming languages. There are two numeric types: `int`, which stores an integer, and `float`, which stores a floating-point real number. Bounds for the possible values for variables of these types are implementation-dependent, as well as the precision for `float` variables. Literals of these types are recognized as being a sequence of digits, and a sequence of digits containing a period, respectively.

A variable of the `char` type contains up to a single character, whereas one of the `string` type contains any number of characters, of an encoding defined by the implementation. A `char` literal is delimited by single quotes, and a `string` by double quotes.

Last but not least, is the type `bool`, which represents a boolean value: `true` or `false`. In the following code, we declare and initialize a variable of each of the primitive types.

```
i : int = -9;
f : float = 0.1;
c : char = '$';
s : string = "hwyl";
b : bool = true;
```

2.2.2 Composite types

Composite types are built-in types of the Gryph language which are composed of other types. As we will see, each composite type has its own characterizing delimiter, used to write variables of that type and to access them.

The first and most simple of these is a `tuple`, delimited by parentheses. The tuple type is written as a sequence of comma-separated types, indicating the type of each position in the tuple sequentially, enclosed in parentheses. The tuple itself consists of a fixed sequence of immutable values of different types, separated by commas, enclosed in parentheses. The syntax for accessing a particular (0-indexed) position in a tuple is an exception in that it does not use the type delimiter (parentheses, in this case), but the backslash.

We also have the `list`, a list of mutable values, all of the same type, delimited by square brackets, which are also used to access a particular (again, 0-indexed) position in the list. The list type is written as a single type, representing the type of its elements, enclosed in square brackets. Lists can grow and shrink in size in execution time, as elements are appended to or removed from them (see 2.4.1.3).

The `dictionary` or `map` is used for associating "keys" of a certain type to "values" of another (or the same) type. It can also grow and shrink dynamically. It consists of comma-separated key/value pairs separated by a question mark, and is delimited by the pipe (or vertical bar), which is also used to access the value associated with a particular key. The dictionary type's syntax is two types, separated by a comma, representing the type of the keys and of the values in that dictionary, respectively, both delimited by a pair of pipes.

In the code below, we declare and initialize variables of each of these types, and access values contained in them, the printed values being indicated by an accompanying comment. Statements for insertion and removal of elements from certain composite types is covered in **??** and operations (besides access) with them in **??**.

```
t : (int, char, int) = (-1, 'x', 1);
println t\1\; # 'x'

l : [int] = [-2, 0, 1];
l[1] = l[0] + 1;
println l[1]; # -1

```

```
d : |char, int| = |'a' ? 1, 'c' ? 3|;
d|'a'| = 0;
println d|'a'|; # 0
```

2.2.3 User-defined types

A user-defined type, or `record`, must firstly be declared, as described in 2.1.3, before any variable of that type can be declared. A variable of a user-defined type is declared as one of any other type, and an expression of a user-defined type consists of that type's identifier, followed by a list of comma-separated assignments for any subset of that type's fields, delimited by braces. Braces are also used to access a particular field in a `record` variable, as demonstrated in the following code:

```
Person {
    name : string;
    age : int = 0;
}

p : Person = Person {name = "Leonhard"};
p{age} = 311;
println p{name}; # "Leonhard"
println p{age}; # 311
```

2.3 Expressions

pass

2.3.1 Operations

pass

2.3.1.1 Logical and relational operations

pass

2.3.1.2 Arithmetic operations

pass

2.3.1.3 Structure-oriented operations

pass

2.3.1.4 Casting

pass

2.4 Statements

A statement is the basic building block of a Gryph program. It comes in two forms: simple statements (2.4.1); or compound statements (2.4.2), which may contain other statements, compound or simple.

2.4.1 Simple statements

A simple statement can be: a variable declaration or assignment (2.4.1.1), an I/O statement (2.4.1.2), an insertion or removal operation over a composite type (2.4.1.3), or a return (2.4.1.4) or escape (2.4.1.5) statement. The end of a simple statement is always marked by a semicolon.

2.4.1.1 Variable declaration and assignment

A variable declaration statement consists of an identifier, or a list of comma-separated identifiers, followed by a colon and a valid type. This declares a variable of the given type for each of the identifiers used. Optionally, the declared variables can be initialized, with an equals sign followed by an expression or a list of comma-separated expressions, after the declared type.

If a list of identifiers is initialized with a single expression, that expression is assigned to all the declared variables, but if they are initialized with a list of expressions, each expression is assigned respectively to each variable corresponding to that identifier. A declaration with a list of identifiers and a list of initializing expressions of different sizes is not meaningful, and is in fact invalid.

A variable assignment consists of an expression, or list of comma-separated expressions, where each expression must evaluate to a variable, followed by an equals sign and a single expression or a list of expressions. Assignment to multiple variables works in the same way as initialization of multiple variables.

The following code declares and initializes three integers, and reasigns values to two of them. The printed values are shown as comments.

```
1 a, b, c : int = 0;
2 a, c = -1, 1;
3 println a; # -1
4 println b; # 0
5 println c; # 1
```

2.4.1.2 I/O statements

As for I/O output statements, there is `read` for input and `print` or `println` for output. For input, the `read` keyword is used, followed by a variable of the type `string`, where `read` will store the one line it reads from standard input. Numeric values can be "read" by attempting to cast the read string to the desired numeric type (see 2.3.1.4).

And for output, the `print` statement, followed by any expression, will print a string representing the value of that expression to standard output. The `println` alternative works in the same way as `print`, put also prints a line break after the printed value.

The following code simply reads a line from standard input an prints it back.

```
1 s : string;
2 read s;
3 println s;
```

2.4.1.3 Insertion and removal statements

Now, in regards to insertion and removal, the two relevant keywords to be used are `add` and `del`, respectively. Lists support both insertion and removal: to insert a new element `x` to a list `y`, the correct syntax is `add x in y`, which appends `x` to the end of `y`, increasing the list's size by one. To remove an element from a list, one might write `del p from y`, where `p` is an integer indicating the position in the list of the element to be removed.

The `tuple`, by contrast, supports neither insertion nor removal, being immutable; while the `dictionary` type supports removal, but insertion only through assignment to a key not yet in it. For removal of an entry with key `k` from a dictionary `d`, the expected syntax is `del k from d`. Note that just as a list will not allow an element to be removed from an invalid position, a dictionary will not allow removal of an entry with a certain key if it contains no such entry.

2.4.1.4 Return statement

The return statement is one that may only be used inside subprograms, it ends the function immeadiatly, executing none of the statements that would follow it, and returns a value if necessary. For functions, in particular, the `return` keyword is always followed by an expression, which must be evaluated to the same type as that function's return type. For procedures (subprograms with no return value), the return statement consists of only the `return` keyword.

2.4.1.5 Escape statement

An escape statement is used to exit an iteration structure (see 2.4.2.2) independently to its regular exit conditions, continuing execution from the first statement following the end of the iteration structure wherein it was called. As such, it may only be used inside an iteration structure, be it a `for` or a `while`. Note that this escape statement is not "multi-level", that is, when called from within a nested loop, it exits only the innermost loop.

2.4.2 Compound statements

A compound statement can also be defined as a control structure, in that they alter the sequential flow of control based on certain conditions. They are divided in conditional structures (2.4.2.1) and iteration structures (2.4.2.2).

2.4.2.1 Conditional structures

Conditional structures are constructed with the keywords `if` and `else`. These work in much the same way as in other programming languages, such as Java. The former, `if`, is always followed by an expression enclosed in parentheses, which must evaluate to a boolean value. If it is true, the code in the braces-enclosed block that follows it is

executed. The latter, `else`, is always preceded by an `if`-block. The block that follows `else` is executed if and only if the preceding block wasn't. The blocks that follow `if` or `else` may actually be a single statement, alternatively to one or many statements enclosed by braces.

The following code will print a different message depending on the value of variable `a` (which must be of type `int` or `float`).

```
1 if (a > 0) {
2     println "greater";
3 } else if (a == 0) {
4     println "equal";
5 } else {
6     println "lesser";
7 }
```

2.4.2.2 Iteration

pass

3 Implementation

Done