

The Gryph Programming Language

Vinícius Campos Artur Curinga Vitor Greati Carlos Vieira

June 16, 2018

Contents

1	Introduction	1
2	Manual	2
2.1	Structure	2
2.1.1	Comments	2
2.1.2	Subprogram declarations	2
2.1.3	Data type declarations	3
2.1.4	Identifiers and scope	3
2.1.5	Importing files	3
2.2	Types	4
2.2.1	Primitive types	4
2.2.2	Composite types	4
2.2.3	User-defined types	5
2.3	Expressions	6
2.4	Statements	6
3	Implementation	6

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developed project, parts of this document may be inaccurate or outdated: in case you detect any inconsistencies, we ask that you contact the authors of this document or raise an issue in the official Gryph repository.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 Manual

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 ??).

```
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. 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 (variable declaration and attribution are described in ??) 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.

```
1 Person {  
2     name : string;  
3     age  : int = 0;  
4 }
```

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`”:

```
1 sub f() : string {
2     return "hello from file1";
3 }
```

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:

```
1 use "file1.gph"
2
3 println f();
```

2.2 Types

The Gryph language is statically typed, meaning the type of all variables is known 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.

```
1 i : int = -9;
2 f : float = 0.1;
3 c : char = '$';
4 s : string = "hwyl";
5 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. It 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. Lists can grow and shrink in size in execution time, as elements are appended to or removed from them.

The **dictionary** or **map** is used for associating “keys” of a certain type to “value” 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.

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 ??.

```

1  t : (int, char, int) = (-1, 'x', 1);
2  println t\1\; # 'x'
3
4  l : [int] = [-2, 0, 1];
5  l[1] = l[0] + 1;
6  println l[1]; # -1
7
8  d : |char, int| = |'a' ? 1, 'c' ? 3|;
9  d|'a'| = 0;
10 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 attributions for any 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:

```

1  Person {
2      name : string;
3      age : int = 0;
4  }
5
6  p : Person = Person {name = "Leonhard"};
7  p{age} = 311;
8  println p{name}; # "Leonhard"
9  println p{age}; # 311

```

2.3 Expressions

pass

2.4 Statements

pass

3 Implementation

Done