The Gryph Programming Language

Vinícius Campos[*] Artur Curinga[†] Vitor Greati[‡] Carlos Vieira[§]

June 18, 2018

Contents

[*]viniciuscampos120@gmail.com
[†]arturmcuringa@gmail.com
[‡]vitorgreati@gmail.com
[§]vieiramecarlos@gmail.com

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 (section 2), in terms of both syntax and semantics; then it expounds on the first, and so far only, implementation of Gryph (section 3); and finally it presents the Gryph syntax in EBNF (section 4). Being an actively developed project, parts of this document may be inaccurate or outdated: in case you detect any inconsistencies, we ask that you contact the authors of this document or raise an issue in the official Gryph repository.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 and design

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 section 2.2.1.2).

```
# print hello world to standard output
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

An 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 section 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.

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

sub my_print(s : string = "") {
    println s;
}
```

In the above code, we define two subprograms. The first, `hello`, receives no argument, and returns a "`string`" (further information on types in section 2.3). Note the `return` keyword in line 2, which signals the function to return the expression following it (see section 2.2.1.4). 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 section 2.2.1.1) 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.

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

The expression used to call a subprogram consists of the subprogram identifier, followed by parentheses containing the arguments being passed to the subprogram. The arguments can be passed sequentially, as comma-separated expressions, or as comma-separated "attributions", of the form parameter=argument. 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:

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

2.1.3 Data type declarations

Another essential part of many Gryph programs, 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 section 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 section 2.3.3.

```
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.

Variables can be declared (see section 2.2.1.1) in any part of a Gryph program, but two variables cannot be declared with the same identifier in the same scope. Likewise, there cannot be two subprograms defined with the same identifier and parameter profile, regardless of scope, since subprograms, as well as user types, can only be defined at the global scope. User-defined data types must also have unique identifiers, and cannot contain two fields with the same name, in the same way that subprograms cannot be defined with two of its parameters with the same name. Furthermore, scope is static, that is, it can be determined prior to execution, and since subprogram definitions cannot be nested, nested scopes are created only by nested blocks (see section 2.2.2).

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 Statements

A statement is the basic building block of a Gryph program. It comes in two forms: simple statements (section 2.2.1); or compound statements (section 2.2.2), which may contain other statements, compound or simple.

2.2.1 Simple statements

A simple statement can be: a variable declaration or assignment (section 2.2.1.1), an I/O statement (section 2.2.1.2), an insertion or removal operation over a composite type (section 2.2.1.3), or a return (section 2.2.1.4) or escape (section 2.2.1.5) statement. The end of a simple statement is always marked by a semicolon.

2.2.1.1 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 reassigns values to two of them. The printed values are shown as comments.

```
a, b, c : int = 0;
a, c = -1, 1;
println a; # -1
println b; # 0
println c; # 1
```

2.2.1.2 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 section 2.4.2.4).

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.

```
s : string;
read s;
println s;
```

2.2.1.3 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.

The `graph` type supports insertion and removal of both vertices and edges. To insert a vertex `v` in a graph `g`: `add v in g`. To remove it: `del v from g`. Removing a vertex from a graph also removes all edges connected to it. Inserting and removing edges works in a similar way, but with the option of using the `where` keyword to specify a value to the edge being inserted, as in graph comprehension (see section 2.4.1).

In the following code we exemplify some of the statements discussed in this section.

```
l : [int] = [-5, 0];
add 5 in l;
del 1 from l;
println l; # [-5, 5]

d : |string, bool| = |"white" ? false|;
d|"black"| = true;
del "white" from d;
println d; # |"black" ? true|

g : <int, char> = <[1]>;
add 2 in g;
add 'x' where 1->2 in g;
println g<1->2>; # 'x'
del 1->2 from g;
```

2.2.1.4 Return statement

The return statement is one that may only be used inside subprograms, it ends the function immediately, 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.

2.2.1.5 Escape statement

An escape statement is used to exit an iteration structure (see section 2.2.2.2) 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.

2.2.2 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 (section 2.2.2.1) and iteration structures (section 2.2.2.2).

2.2.2.1 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`).

```
if (a > 0) {
    println "greater";
} else if (a == 0) {
    println "equal";
} else {
    println "lesser";
}
```

2.2.2.2 Iteration

Another important part of control flow statements are the iteration constructs. Gryph has types of these: a 'while' loop, which repeats code while a certain condition is true; a 'for' loop, which iterate over a composite type, such as a list; and loops for specific traversal of a graph, which can be a BFS loop or a DFS loop.

A while loop is constructed with the keyword `while`, followed by an expression enclosed in parentheses which must evaluate to a boolean value, and then followed by a block of statements enclosed in braces: the code that will be repeated in each iteration of the loop. The block is repeated every time it reaches its end and the condition after `while` is still true. If the condition is false when execution first reaches the `while` statement, the block is never executed.

In the code below, we use a while loop to print multiple values of `n`, shown as comments below the loop.

```
n : int = 3;
while (n > 0) {
    println n;
    n = n - 1;
}
# 3
# 2
# 1
println n; # 0
```

A for loop, commonly called in this case a foreach loop, since it iterates over elements in a data structure, is composed of the keyword `for`, followed by an identifier or list of identifiers, the `over` keyword, and the expression or list of expressions that will evaluate to values of composite type over which you will iterate. Then comes the block that will be executed in each iteration.

In the case of lists, you might have one identifier in the left of `over` and one list in the right: in this case, a variable with the identifier given may be used inside the `for`-block, with its value assuming at each iteration the value of the next element in the list on the right side of `over`, starting with the first. In the case of multiple identifiers and lists on either side of `over`, the iteration works much like a nested loop: for each value of the first list assumed by the first identifier's variable, all values of the second list are assumed in order by the second identifier's variable, and so on.

We exemplify both of these cases in the code below. The first loop should have the same output as the while loop shown above, and the output of the second is shown as comments below it.

```
for i over [3, 2, 1] {
    println i;
}

for i, j over [0, 1], [2, 4] {
    println i + j;
}
# 2
# 4
# 3
# 5
```

Iteration over a dictionary works in a similar way to a list, but the type of the iteration variable is that of a (key, value) tuple, for the key and value types of the dictionary being iterated over, to which the values for each of the dictionary's entry is assigned in turn. In the code below, we iterate over a dictionary `d`, printing "true" for each of its entries.

```
d : |int, char| = |2 ? 'x', 1 ? 'y'|;
for entry over d {
    println d|entry\0\| == entry\1\;
}
```

As for graph iteration, the `for` syntax works in the same way as a list, iterating over the graph's vertices in no particular order. To specify an order, a graph-traversal keyword such as `bfs` or `dfs` may be used. When iterating through a graph using `bfs` or `dfs`, up to three identifiers may be specified, which will assume at each iteration the values of, respectively: the current vertex, its 'parent' vertex, and the current distance or depth from the starting vertex.

As with conditional structures, the blocks contained in iteration structures may actually be a single statement, alternatively to one or many statements enclosed by braces.

2.3 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 (section 2.3.1), and composite (section 2.3.2); as well as user-defined types (section 2.3.3).

2.3.1 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;
```

2.3.2 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 section 2.2.1.3). A list can also be constructed by list comprehension constructs, described in section 2.4.1.

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.

Graphs are essentially a set of vertices, and a set of edges connecting pairs of vertices. This same concept applies to the `graph` type in Gryph. The `graph` type itself is written as the type of its vertices, an optionally, followed by a comma, the type of its edges, all enclosed by the '`<`' and '`>`' characters. When no type is given to edges of a graph, it means its edges hold no data.

Graphs themselves are delimited by these same characters, inside which there is a vertex set and an edge set, separated by commas, each one being optional to the construction of graph. A vertex set is an expression which must evaluate to a list whose elements are of the same type as that graph's vertices' type, while an edge set must be a graph comprehension expression, defined in section 2.4.1. An edge itself is generally represented by two expressions (representing vertices) separated by '`--`', for an undirected edge, or by '`->`' or '`<-`', for a directed edge.

Access to a certain vertex on a graph, through its characteristic delimiters, gives that vertex's adjacency list, while access to an edge gives that edge's data, or "weight", which can be modified through attribution. Adding and removing edges from a graph can be done using the regular appropriate statements, defined in section 2.2.1.3.

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 section 2.2.1.3 and operations (besides access) with them in section 2.4.2.

```
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

g : <int, float> = <[1, 2, 4]>;
```

```
add 0.5 where 1->2 in g;
add 0.25 where 1->4 in g;
println g<1>; # [4, 2]
println g<1->4>; # 0.25
```

2.3.3 User-defined types

A user-defined type, or `record`, must firstly be declared, as described in section 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
```

2.4 Expressions

Gryph expressions can take the form of literals of primitive type (section 2.3.1); composite type structures (section 2.3.2), of which comprehension expressions are a part of (section 2.4.1); subprogram calls (section 2.1.2); variables (section 2.2.1.1); and any combination of these with valid operations (section 2.4.2).

2.4.1 List and graph comprehension

Lists and graphs can be expressed in Gryph through special comprehension constructs. For lists, this is an expression, followed by a `for` loop (see section 2.2.2.2), and then an optional predicate, all enclosed in square brackets. This iterates through the for loop as normal, and appends sequentially to the new list the values for the contained expression at each iteration. The predicate is composed of the `when` keyword, followed by an expression enclosed in parentheses which must evaluate to a boolean value: it means that, at each iteration, the value that would be appended to the list is only inserted if the expressed condition is satisfied. In the following code we create two lists using list comprehension:

```
l1 : [int] = [i+1 for i over [1, 2, 3]];
println l1; # [2, 3, 4]

l2 : [int] = [i*2 for i over l1 when (i % 2 == 0)];
println l2; # [4, 8]
```

Graph comprehension is used to generate a graph's edges and vertices simultaneously. This is is done with a syntax similar to list comprehension, only the first expression is not a value to be inserted in a list, but and edge to be inserted in a graph, along with the vertices it connects. This edge expression can be preceded by a 'weight' specifier: an expression representing the value to be stored in the edge being inserted at each iteration, followed by the keyword `where`.

In the code below, we construct a fully connected directed graph over vertex set `v`, giving each edge a weight equal to the product of its vertices. Printing the graph shows the adjacency list of each vertex, as well as the weight of each edge, in parentheses. This output is shown here as comments below the `println` statement.

```
v : [int] = [-1, 1, 3];
g : <int, int> = <i*j where i->j for i, j over v, v>;

println g;
# <
#     -1 -> 3 (-3), 1 (-1), -1 (1)
#     1 -> 3 (3), 1 (1), -1 (-1)
#     3 -> 3 (9), 1 (3), -1 (-3)
# >
```

2.4.2 Operations

In this section we describe all operations which can be used to modify and combine expressions, except for structure access operations, described in section 2.3.2.

2.4.2.1 Logical and relational operations

Gryph has logical operators `not`, `and`, `or`, and `xor`, which behave in the usual way. Expression containing logical operators are always evaluated using short-circuit evaluation. The equality operation `==` is defined between two expressions if they are of the same type, or if one can be coerced into the other's type, and evaluates to true if and only if both expressions represent the same value. The relational operators `<`, `>`, `<=`, and `>=` follow the same type rule, but are defined only for built-in types, signifying numerical or lexicographical order depending on its operands.

2.4.2.2 Arithmetic operations

Arithmetic operators include unary `-` and `+`, and binary `+`, `-`, `*`, and `/`, with ordinary meanings for numeric types. Additionally, there are the `^` operator for exponentiation, and the modulo operator `%`.

2.4.2.3 Structure-oriented operations

As for structure-specific operations, in the case of strings, `+` can be used with another string for concatenation, and `*` can be used with an integer for duplication. Similarly for lists, there is `++` and `**` for list concatenation and duplication, respectively. Not to mention access operations described in section 2.3.2.

2.4.2.4 Casting

Expression can be explicitly converted from one type to another with the use of the operator `@`. For example, in the following code we cast an integer and a float to its string representations before concatenating and printing them:

```
i : int = 3;
f : float = 0.3;

print i@string + " + " + f@string;
println " = " + (i+f)@string;
# 3 + 0.3 = 3.3
```

3 Implementation

The interpreter for the Gryph Programming Language was implemented in Haskell during the first semester of 2018, for an undergraduate course on programming languages. It is comprised of three main steps: **lexical analysis**, for tokens discovery; **syntactic analysis**, for language constructs identification; and **program execution**, for performing machine state changes according to the meaning of the parsed entities, given by the semantic rules of the language (see section 2). This section presents in detail how these three steps were implemented.

3.1 Lexical analysis

Lexical analysis is the part of the syntax analysis task which deals with small-scale constructs. Dealing with these separately is justified by gains in terms of simplicity, since it can be done with simple techniques (pattern matching, essentially); efficiency, since it is more suitable for optimization than the general syntactic parser; and portability, since lexical analysers are platform dependent, in contrast with syntax analysers, which can be made independent.

In order to expedite the development process, **Alex**[1], a lexical analyzer generator for Haskell, was used. It is basically built upon a sequence of regular expressions that describe the language tokens. The implemented lexical analyzer takes a program as input and produces a list of the identified tokens, which is then transmitted to the syntactic parser, described in the next subsection. Alex also furnishes the position (line and column) of each token, for later error reporting, but this version of the interpreter does not yet take this information into account.

3.2 Syntactic analysis

The syntactic analyzer deals with the large-scale constructs, such as expressions, statements and subprogram definitions. It takes the list of tokens produced in the lexical analysis and the language grammar (generally in BNF, Backus-Naur Form) and performs the syntactic parsing according to its productions. Some adjustments may be necessary in

[1]https://www.haskell.org/alex/

order to adequate the grammar rules to some restrictions imposed by the adopted parsing algorithm.

Syntactic analysis, in the implemented interpreter, is performed with the aid of the **Parsec**[2] parser library for Haskell. Parsec is based on *monadic parser combinators*, and is fast, safe, well-documented, and highly customizable. Basically, developing a parser with this tool is a matter of writing and combining small parsers for each grammar construct, bewaring of left-recursive productions, since the underlying algorithm doesn't accept them.

The output of the syntactic parser in this implementation is a syntactic tree represented by Haskell user-defined types. The most general constructs are called **program units**, which are abstractions for import commands, subprogram definitions, user type definitions and statements. The Haskell type defined to represent such concept was:

```
data ProgramUnit =  Stmt Stmt |
                    SubprogramDecl Subprogram |
                    StructDecl StructDecl |
                    Use String
deriving (Show, Eq)
```

Therefore, a program is just a sequence of program units. Statements, in turn, are the basic execution units in a program, and can assume various forms, like control structures, input/output statements, variable declaration and assignments (see section 2.2). The type defined for representing statements was:

```
data Stmt = ReadStmt Identifier |
            PrintStmt ArithExpr |
            PrintLnStmt ArithExpr |
            DeclStmt VarDeclaration |
            AttrStmt [ArithExpr] [ArithExpr] |
            SubCallStmt SubprogCall |
            IfStmt ArithExpr IfBody ElseBody |
            ReturnStmt (Maybe ArithExpr) |
            ForStmt [Identifier] [ArithExpr] Block |
            WhileStmt ArithExpr Block |
            BfsStmt [Identifier] ArithExpr (Maybe ArithExpr) Block |
            DfsStmt [Identifier] ArithExpr (Maybe ArithExpr) Block |
            AddStmt ArithExpr ArithExpr |
            AddEdgeStmt (Maybe ArithExpr) Edge ArithExpr |
            DelStmt ArithExpr ArithExpr |
            DelEdgeStmt Edge ArithExpr |
            BreakStmt
deriving (Show, Eq)
```

The listing above shows the general approach of representing each language construct with values of custom Haskell types. It's worth noting that the type ArithExpr appears frequently, since it represents any valid expression, as shown by its type definition:

```
data ArithExpr =    ArithUnExpr ArithUnOp ArithExpr |
                    ArithBinExpr ArithBinOp ArithExpr ArithExpr |
                    ArithTerm Term |
                    ExprLiteral ExprLiteral |
                    GraphAccess ArithExpr ArithExpr |
                    GraphEdgeAccess ArithExpr Edge |
                    DictAccess ArithExpr ArithExpr |
                    ListAccess ArithExpr ArithExpr |
```

[2] http://hackage.haskell.org/package/parsec

```
                    StructAccess ArithExpr Identifier |
                    TupleAccess ArithExpr ArithExpr |
                    CastExpr ArithExpr GType |
                    ArithRelExpr RelOp ArithExpr ArithExpr |
                    ArithEqExpr EqOp ArithExpr ArithExpr |
                    LogicalBinExpr BoolBinOp ArithExpr ArithExpr |
                    StructInitExpr StructInit
deriving (Show, Eq)
```

In this way, every syntactically correct program becomes a list of program units, which, in turn, are values defined over other Haskell types. For example, consider the following Gryph valid program, which prints the sum of two declared and initialized variables:

```
a : int = 10;
b : int = 30;
println a+b;
```

The corresponding output of the syntactic analysis is:

```
[Stmt (DeclStmt (VarDeclaration [Ident "a"] GInteger [ArithTerm
(LitTerm (Lit 10))])), Stmt (DeclStmt (VarDeclaration [Ident "b"]
GInteger [ArithTerm (LitTerm (Lit 30))])), Stmt (PrintLnStmt
(ArithBinExpr PlusBinOp (ArithTerm (IdTerm (Ident "a")))
(ArithTerm (IdTerm (Ident "b")))))]
```

The complete list of types defined for representing syntactic entities can be found in the interpreter module Syntactic.Syntax. Finally, after the syntactic analysis, the list of program units is processed, in order to execute the implemented semantic rules of the language and change the machine state for performing computation.

3.3 Program execution

3.3.1 Auxiliary structures

Some data structures were used to simulate the data and program memory, as well as the current scope stack during a program execution.

3.3.1.1 Memory

The memory structure's purpose is to store the variables' attributes during program execution. It is represented by a Data.Map in Haskell, which is an efficient implementation of a dictionary, having the following definition:

```
type Memory = M.Map CellIdentifier Cell
```

A CellIdentifier is a pair containing the name and scope of a variable, and the Cell is another pair, storing a type and a memory value. There is a difference between memory values and common values of the language, since the memory can also contain a reference (a CellIdentifier) to other variables in memory.

3.3.1.2 Program Memory

The program memory stores subprogram and user-type definitions. Thus, it is defined to store parsed Gryph code. The following type definitions clarify how this was implemented:

```
type ProgramMemory = M.Map UnitIdentifier UnitContent
data UnitIdentifier =   SubIdentifier SubIdentifier |
                        StructIdentifier StructIdentifier deriving (Eq, Show, Ord)
data UnitContent =  SubContent SubContent |
                    StructContent StructContent deriving (Eq, Show)
```

3.3.1.3 Scope stack

The scope stack is intented to point to the current context in a program's execution. It is quite important, since it defines the referencing environments during execution. Values inside this stack are of type Scope, whose definition is:

```
data Scope =    GlobalScope |
                SubScope Integer |
                IterationScope Integer |
                BlockScope Integer deriving (Eq, Show, Ord)
```

Differentiating the types of scopes is important to allow the correct execution of return and break statements. For example, a return statement searches for the last SubScope, to move the execution flow outside of the last called subprogram.

3.3.2 Execution

The input of the execution phase is a list of program units. The exec function is responsible for processing it, as indicated by its signature:

```
exec :: Memory -> ProgramMemory -> Scopes -> [ProgramUnit] ->
                                    IO(Memory, ProgramMemory, Scopes)
```

Each unit is processed by the function execUnit, whose signature is:

```
execUnit :: ProgramUnit -> Memory -> ProgramMemory ->
                    Scopes -> IO (Memory, ProgramMemory, Scopes)
```

The processing is based in pattern matching based on the value constructors of the types presented in the previous subsection. The definition of the execUnit function demonstrates how this works:

```
execUnit (SubprogramDecl sub) m pm ss =
                do
                    pm' <- execSubDecl sub m pm ss
                    return $ (m, pm', ss)
execUnit (StructDecl struct) m pm ss =
                do
                    pm' <- execStructDecl m pm ss struct
                    return $ (m, pm', ss)

execUnit (Stmt stmt) m pm ss =
                do
                    (m', ss', v) <- execStmt stmt m pm ss
                    return $ (m', pm, ss')
execUnit (Use path) m pm ss =
                do
                    f <- parseFile path
                    let decls = filter (\x -> case x of
                                                (Use _) -> False
                                                (Stmt _) -> False
                                                _ -> True) f
                    exec m pm ss decls
```

Similar definitions occur throughout the Execution.Semantic module, which is responsible for the execution step. It is important to notice that the (data) memory, the program memory and the stack of scopes are always transmitted function to function, allowing the effects of each statement over the machine state and the program execution flow to be relayed to the next statement. The following sections details some other important aspects of the implementation.

3.3.2.1 Scope condition for block execution

Before the execution of each statement inside a block, the interpreter checks if the scope introduced in the block entrance is the same as the current scope (the top of the scope stack). This is done to allow returns and breaks to work properly: such statements cause the scope stack to change, which, by this condition, makes it so the subsequent statements don't execute.

3.3.2.2 Graph representation

A custom type was defined for the representation of graphs in Gryph. This was done to allow graphs storing data of any type in its vertices and edges. Below is the Haskell definition:

```haskell
data Vertex a = Vertex Int a
data Edge a b = Edge (Vertex a) (Vertex a) b
data Graph a b = Graph (S.Set (Vertex a)) (M.Map Int [Edge a b])
                       deriving (Eq, Ord)
```

A graph is then a set of vertices internally identified by integers holding some type a, together with a map of vertex identifiers to edges connecting vertices of type a and holding data ("weight") of type b. This map is actually a representation of the graph adjacency list. In this way, vertices and edges can be of any Gryph type, increasing the range of possible applications of the language.

3.4 Verifications

The interpreter performs mainly the following verifications, exhibiting an error message when one of them is not satisfied by the program:

Duplicity of variable declaration There cannot exist two or more variables with the same name in the same scope.

Duplicity of subprogram definition There cannot exist two or more subprograms with the same name and the same parameter profile.

Duplicity of subprogram formal parameters A subprogram cannot have two or more formal parameters with the same name.

Duplicity of structs There cannot exist two or more structs with the same name.

Duplicity of struct fields A struct cannot have two fields with the same name.

Positional actual parameters after named parameters A subprogram call cannot have positional actual parameters after named parameters.

Type compatibility Types are verified for every operation. Widening coercion occurs only for Integer and Float types.

Range of indexes for list access Always varies between 0 and $n-1$, where n is the length of the list.

Dictionary access A dictionary can only be accessed in previously defined keys.

Fuction call inside expression A function call cannot be used in an expression if its execution ends with no return value.

4 Syntax in EBNF

In this section we present Gryph's syntax in Extended Backus-Naur Form (EBNF). This particular version of EBNF uses three common extensions to BNF: (i) optional parts in the right-hand side (RHS) of a production are enclosed in square brackets; (ii) parts of the RHS of a production that can be repeated indefinitly, or left out altogether, are enclosed in braces; and (iii) a multiple-choice option may appear in the RHS of a production, enclosed in parentheses, where exactly one of the pipe-separated options must be chosen.

Note that all terminal symbols appear **bolded**.

4.1 Program

$$\begin{aligned}
\langle\text{program}\rangle &\models \langle\text{program-unit}\rangle\{\langle\text{program-unit}\rangle\}\\
\langle\text{program-unit}\rangle &\models \langle\text{stmt}\rangle \mid \langle\text{subprog-def}\rangle \mid \langle\text{type-def}\rangle \mid \langle\text{include}\rangle\\
\langle\text{include}\rangle &\models \textbf{use}\ \langle\text{string-lit}\rangle
\end{aligned}$$

4.2 Identifiers

$$\begin{aligned}
\langle\text{id-list}\rangle &\models \langle\text{identifier}\rangle\{\textbf{,}\langle\text{identifier}\rangle\}\\
\langle\text{identifier}\rangle &\models \langle\text{alpha}\rangle\langle\text{id-tail}\rangle\\
\langle\text{user-type-id}\rangle &\models \langle\text{upper-alpha}\rangle\langle\text{id-tail}\rangle\\
\langle\text{id-tail}\rangle &\models \{\langle\text{alpha-num}\rangle\}\{\textbf{'}\}\\
\langle\text{alpha-num}\rangle &\models \langle\text{alpha}\rangle \mid \langle\text{digit}\rangle \mid \textbf{\_}\\
\langle\text{digit}\rangle &\models \textbf{0} \mid \ldots \mid \textbf{9}\\
\langle\text{alpha}\rangle &\models \langle\text{upper-alpha}\rangle \mid (\textbf{a} \mid \ldots \mid \textbf{z})\\
\langle\text{upper-alpha}\rangle &\models \textbf{A} \mid \ldots \mid \textbf{Z}
\end{aligned}$$

4.3 Statements

$$\begin{aligned}
\langle\text{stmt-list}\rangle &\models \langle\text{stmt}\rangle\{\langle\text{stmt}\rangle\} \\
\langle\text{stmt-block}\rangle &\models \textbf{\{}\ \langle\text{stmt-list}\rangle\ \textbf{\}} \\
\langle\text{stmt}\rangle &\models \langle\text{matched-stmt}\rangle \mid \langle\text{unmatched-stmt}\rangle \\
\langle\text{block-or-matched}\rangle &\models \langle\text{stmt-block}\rangle \mid \langle\text{matched-stmt}\rangle \\
\langle\text{matched-stmt}\rangle &\models \langle\text{matched-if-else}\rangle \mid \langle\text{iteration-stmt}\rangle \mid \langle\text{simple-stmt}\rangle \\
\langle\text{unmatched-stmt}\rangle &\models \langle\text{if-stmt}\rangle \mid \langle\text{unmatched-if-else}\rangle \\
\langle\text{simple-stmt}\rangle &\models (\langle\text{io-stmt}\rangle \mid \langle\text{var-stmt}\rangle \mid \\
&\qquad \langle\text{add-del-stmt}\rangle \mid \langle\text{subprog-call}\rangle \mid \\
&\qquad \langle\text{return-stmt}\rangle \mid \langle\text{break-stmt}\rangle)\textbf{;}
\end{aligned}$$

4.3.1 IO

$$\begin{aligned}
\langle\text{io-stmt}\rangle &\models \langle\text{read-stmt}\rangle \mid \langle\text{write-stmt}\rangle \\
\langle\text{read-stmt}\rangle &\models \textbf{read}\ \langle\text{identifier}\rangle \\
\langle\text{write-stmt}\rangle &\models \textbf{print}\ \langle\text{expression}\rangle
\end{aligned}$$

4.3.2 Variables

$$\begin{aligned}
\langle\text{var-stmt}\rangle &\models \langle\text{var-decl-stmt}\rangle \mid \langle\text{var-attr-stmt}\rangle \\
\langle\text{var-decl-list}\rangle &\models \langle\text{var-decl-stmt}\rangle\{\textbf{;}\langle\text{var-decl-stmt}\rangle\}\textbf{;} \\
\langle\text{var-decl-stmt}\rangle &\models \langle\text{id-list}\rangle\textbf{:}\langle\text{type}\rangle[\langle\text{var-attr}\rangle] \\
\langle\text{var-attr-stmt}\rangle &\models \langle\text{lhs-expr-list}\rangle\langle\text{var-attr}\rangle \\
\langle\text{id-attr}\rangle &\models \langle\text{identifier}\rangle\textbf{=}\langle\text{expression}\rangle \\
\langle\text{id-attr-list}\rangle &\models \langle\text{id-attr}\rangle\{,\langle\text{id-attr}\rangle\} \\
\langle\text{var-attr}\rangle &\models \textbf{=}\langle\text{expr-list}\rangle
\end{aligned}$$

4.3.3 Insertion and removal

$$\begin{aligned}
\langle\text{add-del-stmt}\rangle &\models \langle\text{add-stmt}\rangle \mid \langle\text{del-stmt}\rangle \\
\langle\text{add-stmt}\rangle &\models \textbf{add}\ \langle\text{expression}\rangle\ \textbf{in}\ \langle\text{lhs-expr}\rangle \\
\langle\text{del-stmt}\rangle &\models \textbf{del}\ \langle\text{expression}\rangle\ \textbf{from}\ \langle\text{lhs-expr}\rangle
\end{aligned}$$

4.4 Control Structures

4.4.1 Conditionals

$$\begin{aligned}
\langle\text{if-expr}\rangle &\models \textbf{if (}\langle\text{expression}\rangle\textbf{)}\\
\langle\text{if-stmt}\rangle &\models \langle\text{if-expr}\rangle\ \langle\text{stmt}\rangle\\
\langle\text{unmatched-if-else}\rangle &\models \langle\text{if-expr}\rangle\ \langle\text{matched-stmt}\rangle\ \textbf{else}\ \langle\text{unmatched-stmt}\rangle\\
\langle\text{matched-if-else}\rangle &\models \langle\text{if-expr}\rangle\ \langle\text{block-or-matched}\rangle\ \textbf{else}\ \langle\text{block-or-matched}\rangle\ \mid\\
&\quad\ \ \langle\text{if-expr}\rangle\ \langle\text{stmt-block}\rangle
\end{aligned}$$

4.4.2 Iteration

$$\begin{aligned}
\langle\text{iteration-stmt}\rangle &\models \langle\text{for-stmt}\rangle\ \mid\ \langle\text{while-stmt}\rangle\ \mid\ \langle\text{bfs-dfs-stmt}\rangle\\
\langle\text{while-stmt}\rangle &\models \textbf{while}\ \langle\text{expression}\rangle\ \langle\text{block-or-matched}\rangle\\
\langle\text{for-loop}\rangle &\models \textbf{for}\ \langle\text{id-list}\rangle\ \textbf{over}\ \langle\text{expr-list}\rangle\\
\langle\text{for-stmt}\rangle &\models \langle\text{for-loop}\rangle\langle\text{block-or-matched}\rangle\\
\langle\text{bfs-dfs-loop}\rangle &\models (\textbf{bfs}\ \mid\ \textbf{dfs})\ \langle\text{id-list}\rangle\ \textbf{over}\ \langle\text{expr-list}\rangle\\
\langle\text{bfs-dfs-stmt}\rangle &\models \langle\text{dfs-dfs-loop}\rangle\langle\text{block-or-matched}\rangle\\
\langle\text{break-stmt}\rangle &\models \textbf{break}
\end{aligned}$$

4.5 Subprograms

$$\begin{aligned}
\langle\text{subprog-def}\rangle &\models \textbf{sub}\langle\text{identifier}\rangle\textbf{(}\langle\text{parameters}\rangle\textbf{)}[\textbf{:}\langle\text{type}\rangle]\langle\text{stmt-block}\rangle\\
\langle\text{parameters}\rangle &\models \langle\text{var-stmt}\rangle\{\textbf{;}\langle\text{var-stmt}\rangle\}\\
\langle\text{subprog-call}\rangle &\models \langle\text{identifier}\rangle\textbf{(}[\langle\text{arguments}\rangle]\textbf{)}\\
\langle\text{arguments}\rangle &\models \langle\text{id-attr-list}\rangle\ \mid\ \langle\text{expr-list}\rangle\\
\langle\text{return-stmt}\rangle &\models \textbf{return}
\end{aligned}$$

4.6 Types

$$\begin{aligned}
\langle\text{type-list}\rangle &\models \langle\text{type}\rangle\{\textbf{,}\langle\text{type}\rangle\}\\
\langle\text{type}\rangle &\models \langle\text{native-type}\rangle\ \mid\ \langle\text{user-type-id}\rangle\\
\langle\text{native-type}\rangle &\models \langle\text{primitive-type}\rangle\ \mid\ \langle\text{composite-type}\rangle\\
\langle\text{primitive-type}\rangle &\models \textbf{int}\ \mid\ \textbf{float}\ \mid\ \textbf{char}\ \mid\ \textbf{string}
\end{aligned}$$

⟨composite-type⟩	⊨	**[**⟨type⟩**]** | |⟨type⟩| | **(**⟨type⟩**,**⟨type-list⟩**)** | ⟨graph-type⟩
⟨graph-type⟩	⊨	<⟨type⟩> | <⟨type⟩**,**⟨type⟩>
⟨type-def⟩	⊨	⟨user-type-id⟩**{**⟨var-decl-list⟩**}**

Observation Although there is no maximum size for tuples in the definition above, there may be one for specific language implementations.

4.7 Expressions

⟨expr-list⟩	⊨	⟨expression⟩{**,**⟨expression⟩}
⟨expression⟩	⊨	⟨logical-xor-expr⟩
⟨logical-xor-expr⟩	⊨	⟨logical-or-expr⟩{**xor** ⟨logical-or-expr⟩}
⟨logical-or-expr⟩	⊨	⟨logical-and-expr⟩{**or** ⟨logical-and-expr⟩}
⟨logical-and-expr⟩	⊨	⟨equality-expr⟩{**and** ⟨equality-expr⟩}
⟨equality-expr⟩	⊨	⟨rel-expr⟩{⟨equality-op⟩ ⟨rel-expr⟩}
⟨rel-expr⟩	⊨	⟨add-expr⟩{⟨rel-op⟩⟨add-expr⟩}
⟨add-expr⟩	⊨	⟨mult-expr⟩{⟨add-op⟩⟨mult-expr⟩}
⟨mult-expr⟩	⊨	⟨exp-expr⟩{⟨mult-op⟩⟨exp-expr⟩}
⟨exp-expr⟩	⊨	⟨cast-expr⟩[⟨exp-op⟩⟨exp-expr⟩]
⟨cast-expr⟩	⊨	⟨unary-expr⟩{**@**⟨type⟩}
⟨unary-expr⟩	⊨	⟨unary-op⟩⟨cast-expr⟩ | ⟨postfix-expr⟩
⟨postfix-expr⟩	⊨	⟨primary-expr⟩{⟨access-expr⟩}
⟨lhs-expr⟩	⊨	⟨identifier⟩{⟨access-expr⟩}
⟨lhs-expr-list⟩	⊨	⟨lhs-expr⟩{**,**⟨lhs-expr⟩}
⟨access-expr⟩	⊨	|⟨expression⟩| | <⟨expression⟩> | **[**⟨expression⟩**]** | **{**⟨identifier⟩**}** | \⟨expression⟩\
⟨primary-expr⟩	⊨	**(**⟨expression⟩**)** | ⟨identifier⟩ | ⟨subprog-call⟩ | ⟨literal⟩ | ⟨structure⟩

4.7.1 Literals

⟨literal⟩	⊨	⟨int-lit⟩ | ⟨float-lit⟩ | ⟨string-lit⟩ | ⟨bool-lit⟩ | ⟨char-lit⟩
⟨bool-lit⟩	⊨	**true** | **false**
⟨string-lit⟩	⊨	`"`{⟨char⟩}`"`
⟨char-lit⟩	⊨	`'`⟨char⟩`'`
⟨char⟩	⊨	`implementation dependent`

$$\begin{aligned}
\langle\text{int-lit}\rangle &\models [\texttt{-}]\langle\text{digit-seq}\rangle \\
\langle\text{float-lit}\rangle &\models [\texttt{-}]\langle\text{digit-seq}\rangle\texttt{.}\langle\text{digit-seq}\rangle \\
\langle\text{digit-seq}\rangle &\models \langle\text{digit}\rangle\{\langle\text{digit}\rangle\}
\end{aligned}$$

Observation A char must be one character, of an enconding defined by the implementation.

4.7.2 Structures

$$\begin{aligned}
\langle\text{structure}\rangle &\models \langle\text{tuple}\rangle \mid \langle\text{list}\rangle \mid \langle\text{dict}\rangle \mid \\
&\ \langle\text{graph}\rangle \mid \langle\text{user-type}\rangle \mid \langle\text{edge}\rangle \\
\langle\text{tuple}\rangle &\models \mathbf{(}\langle\text{expr-list}\rangle\mathbf{)} \\
\langle\text{dict}\rangle &\models |[\langle\text{dict-entry-list}\rangle]| \\
\langle\text{dict-entry}\rangle &\models \langle\text{expression}\rangle\mathbf{?}\langle\text{expression}\rangle \\
\langle\text{dict-entry-list}\rangle &\models \langle\text{dict-entry}\rangle\{\mathbf{,}\langle\text{dict-entry}\rangle\} \\
\langle\text{user-type}\rangle &\models \langle\text{user-type-id}\rangle\mathbf{\{}[\langle\text{id-attr-list}\rangle]\mathbf{\}} \\
\langle\text{list}\rangle &\models \mathbf{[}[\langle\text{list-expr}\rangle]\mathbf{]} \\
\langle\text{list-expr}\rangle &\models \langle\text{expr-list}\rangle \mid \langle\text{list-comprehension}\rangle \\
\langle\text{list-comprehension}\rangle &\models \langle\text{expression}\rangle\langle\text{for-loop}\rangle[\langle\text{comp-condition}\rangle] \\
\langle\text{graph-comprehension}\rangle &\models \langle\text{edge}\rangle\langle\text{for-loop}\rangle[\langle\text{comp-condition}\rangle] \\
\langle\text{comp-condition}\rangle &\models \mathbf{when}\ \mathbf{(}\langle\text{expression}\rangle\mathbf{)} \\
\langle\text{graph}\rangle &\models <(\langle\text{vertex-set}\rangle, \langle\text{edge-set}\rangle \mid \langle\text{vertex-set}\rangle \mid \langle\text{edge-set}\rangle)> \\
\langle\text{vertex-set}\rangle &\models \langle\text{expression}\rangle \\
\langle\text{edge-set}\rangle &\models [\langle\text{edge-weight}\rangle]\langle\text{graph-comprehension}\rangle \\
\langle\text{edge-weight}\rangle &\models \langle\text{expression}\rangle\ \mathbf{where} \\
\langle\text{edge}\rangle &\models \langle\text{expression}\rangle\langle\text{edge-symbol}\rangle\langle\text{expression}\rangle \\
\langle\text{edge-symbol}\rangle &\models -- \mid -> \mid <-
\end{aligned}$$

4.7.3 Operators

$$\begin{aligned}
\langle\text{rel-op}\rangle &\models > \mid < \mid <= \mid >= \\
\langle\text{equality-op}\rangle &\models == \mid \mathbf{!}= \\
\langle\text{unary-op}\rangle &\models + \mid - \\
\langle\text{add-op}\rangle &\models + \mid - \\
\langle\text{mult-op}\rangle &\models * \mid / \mid \% \mid ++ \mid ** \\
\langle\text{exp-op}\rangle &\models \hat{}
\end{aligned}$$