PRTZL REFERENCE MANUAL

Mathew Mallett
Rusty Nelson
Guanqi Luo

1.Introduction

We propose a language where vertices (aka nodes) and links are a first class data types. These types are used to construct, manipulate and test an implicit graph structure which is the main store in the language. A user of our language would be able to create and manipulate graphs as easily as they could work with an array in most other languages. The language will provide a way to construct the nodes and links, as well as provide efficient implementations of common operations, like searching and path finding through a standard library. The data structures will be open enough to enable developers to build other graph algorithms in the language as well as support a variety of different graph types from directed/undirected, weighted and restricted depending on the programmers adherence to their own rules.

2.Lexical Conventions

There are six types of tokens: identifiers, keywords, number literals, string literals, operators, and other separators. Whitespace is ignored, unless it is included within a string literal.

Reserved Keywords

The following keywords are reserved by the language and not available for variable or function names

```
Number      String          Vertex          Edge
if          elseif          else            endif
while       do              endwhile
return      endfunc         null
```

Comments

```
/*  this is a comment */
/* also
a
comment */
```

Identifiers

An identifier is an alphanumeric string. The first character must be alphabetic. We also allow _ within identifiers. Identifiers are case sensitive.

Separators

We separate statements with ‘;’. Anything after a ‘;’ is a new statement.

3.Types

We have 5 types

Number

Can represent both integer and floating point numbers.
The number 0 is considered “falsey” as a boolean value, all other Number values are “truthy”.
Numbers take a default value of 0.
Numbers are mutable. You are able to reuse variables and rewrite the value stored within them multiple times.

String

Represents a string of characters.
Strings take a default value of “” (empty string)
Strings are immutable. Rewriting a String results in the creation of a new String buffer.

Vertex

Represents a vertex within the graph.
Vertex variables take a value of **null** until they are bound to a Vertex within the graph.
Vertexes are mutable. You are able to write changes to Vertex variables.
Vertexes have a key/value attribute system attached to them. You are able to attach additional information about a specific vertex through a key/value pair.

Edge

Represents an edge between two Vertexes.
Edge variables take a value of **null** until they are bound to an Edge within the graph.
Edges are mutable. You are able to write changes to Edge variables.
Edges have a key/value attribute system attached to them. You are able to attach additional information about a specific vertex through a key/value pair.

List

A list containing any of the other data types. Each list is able to hold one data type. Lists are mutable, you can add and remove items from a list.
Lists variables are initialized to **null**

null
null is a keyword that indicates that a variable has not had a valid pointer assigned to it, or otherwise does not point to a valid object.

Literals
We have number and string literals, as well as an empty list literal.
Number literal
```
      1
      42
      3.141592
```
String literal
```
      “I am literally a literal”
      “Hello, World!”
      “”
```
Empty List literal
```
      []
```

4.Expressions
Below we outline the expressions within our language. We also define order of operations and left or right associativity for each operation. An expression is made up of one or more of the operators listed below. You are able to naturally combine operators to form sequences of complex operations, as long as the types align correctly.

Variable declaration
A declaration is made of two tokens: a type identifier, and a variable identifier. Once a variable identifier has been used within the current scope, you aren’t able to use that identifier again. If you reuse an identifier within a more specific scope, the identifier of the more specific scope will shadow the less specific identifier, just as in C. Variable declarations must be the leftmost section of an expression.

Here are declarations of our 5 types:
```
Number my_num;
String my_string;
Vertex my_vert;
Edge my_edge;
List my_list;
```

Assignment

Assign a literal or another expression to a variable

```
my_num = 3;
my_string = “I am literally a literal”;
```

You can declare and assign in the same expression

```
Number my_other_num = 3.141592;
my_num = my_other num + 32;

Vertex my_other_vert = my_vert;
```

Assignment is right associate. Everything to the right of = is evaluated first. The left side of = (lvalue) must be either a variable declaration or a variable identifier. Other expressions are not valid lvalues.

Binary operators

We support standard operators on Number type, with PEMDAS order of operations

```
      +
      -
      *
      /
my_num = 1 + 2 / 3;
Number my_other_num = my_num + 3.141592;
```

You can concatenate String types with the ^

```
String my_string = “I am literally “ ^ “a literal”;
String my_longer_string = my_string ^ “ Hello, World!”;
```

Binary operators are left associative. The left side of the expression is evaluated first, then the right side, then the operator is applied.

Parenthesis

Use parenthesis to increase the precedence of a sub-expression

```
Number my_number = (1 + 2) * 3;  /* 9 */
```

Binary comparators

We can compare two numbers with the C style comparators

```
    ==
    !=
    >
    >=
    <
    <=
```

Comparators return 1 for true, 0 false

```
Number my_num = 3.14 > 3.13; /* 1 */
my_num = 24 <= -24; /* 0 */
```

Comparators are left associative.

Unary operators

Unary minus (-) inverts the sign on a Number

```
Number my_number = 42;
my_number = -my_number; /* -42 */
```

Not operator inverts the logical truth value of the Number. 0 is false, anything else is true

```
Number my_number = !0; /* 1 */
my_number = !(1 + 1); /* 0 */
```

Precedence of operators

In order of most to least precedence

! - (Unary minus)

/ *

+ -

== != < <= > >=

=

Lists

Allow you to store lists of elements, where all contained elements are the same type. You can create a list with the empty list literal []. You can then add elements onto the end with add(). You can get a specific index with the [] syntax similar to C, you can get the length with .length(), and you can empty it with .clear().

```
List my_list = [];
add(my_list, 3.14);
add(my_list, 42);

my_list[0]; /* 3.14 */

length(my_list); /* 2 */
```

4.1.Graph Operators

There are three primary operations that interface with the graph. The graph is implicitly declared in any PRTZL program, and we provide operators to insert, retrieve, and remove vertices from it.

Insert key operator <+ String string +>

Inserts the string key into the graph. String can be a variable or a literal.

```
<+ “Detroit” +>; /* insert detroit vertex into the graph */

String my_string = “Houston”
<+ my_string +>;
```

Query key operator <? String string ?>

Check the existence of a vertex within a graph, returns the Vertex object, or null object

```
Vertex my_vertex = <? “Detroit” ?>; /* gives the Vertex for Detroit */
Vertex my_other_vertex = <? “Miami” ?>; /* gives null */

my_other_vertex == null; /* gives 1 */
```

Delete vertex from the graph <- String string ->

Attempts to delete the vertex from the graph. If it does not exist, returns 0, if it does exist, it removes all edges to and from the vertex, then removes it from the graph structure, returning 1

```
Number success = <- “Detroit” ->; /* Detroit is deleted, and returns true
*/
success = <- “Detroit” ->; /* Detroit is not in the graph this time, now
returns false; */
```

Vertex data fields

You can store key/value pairs in a Vertex object to associated data with that Vertex with the . operator. Gives null on properties that have not been set yet. You can put Number, String, Vertex, Edge, and Lists into a value. Keys are alphanumeric and allow _ .

```
Vertex houston = <? “Houston” ?>;
houston.visited = 0;
houston.visited; /* 0 */

houston.other; /* null */
```

All Vertices have the following properties built into them

`out`	List of outbound Edges
`in`	List of inbound Edges
`in_degree`	Number of inbound edges
`out_degree`	Number of outbound edges

Edges

Edges are similar to Vertices. They are created when you link two Vertices together. You are able to attach key/value pairs to Edges in the same manner as Vertices.

All Edges have the following properties in their key/value map.

`to`	Destination Vertex
`from`	Source Vertex
`weight`	Number weight of the edge, if not specified, is 1

5.Control Flow

Conditional branching

Use if elseif else to group conditional branches together. Each conditional flow has an **if**, and optionally has more conditionally executed statements as **elseif**. **elseif** is executed if the preceding **if** and **elseif**’s did not evaluate to true. If none of the conditions were true, **else** clause is invoked. The end of a string of **if elseif else** is marked with an **endif**. Every conditional flow is marked by one **if**/**endif** pair, optionally many **elseif**, and optionally one **else** as the final conditional branch.

```
if(1 == 0)
      /* do this */
elseif(1 == 2)
      /* do this */
else
      /* do this */
endif

/* another example */
if(1)
      /* conditional body */
endif
```

Loops

We support the **while** loop. The for loop has 2 pieces. The first is a statement that determines the number of iterations of the loop, the second is the body of the loop. Anything declared in the for statement has loop scope, not global scope. The variables exist only within the loop and are reclaimed after the loop finishes.

Broken down:

```
while (/* continue looping condition */) do
      /* loop body */
end
```

And an example iterating a List.

```
List my_list = [];
/* list is filled here */

Number i = 0
while( i < list_length(my_list)) do
      my_list[i]; /* do something with this element */

      i = i + 1;
endwhile
```

6.Functions

Function declaration is similar to C, with a type, label, and argument signature. The end of the function is denoted with the **end** keyword.

A breakdown of the function declaration with optional pieces marked with [ ]

```
return_type function_name([arg_type arg_name, arg_type arg_name])
      [ function body, has function scope ]
      return a_variable_of_return_type;
end
```

/* here is a demonstration */

```
Number my_function(Number arg1, Number arg2)
      /*  variables declared here have function, not global, scope */
      Number local_var = 2;
      return arg1 + arg2 + arg3;
endfunc
```

Functions can be recursive. Functions must be declared within global scope.

Calling a function
`my_function(arg1, arg2);`

7.Scope Rules

Scoping is simple. We have global scope, if/else scope, loop scope, and function scope.

Anything not declared within an if/else statement, loop, or function is global scope. Globally scoped identifiers are visible anywhere within the program.

If/else, loop, and function scope all work the same. Anything declared within the bodies of these structures have scope limited to the body of the structure. You are able to reuse identifiers that have been used at global scope, the more specific scope will shadow the global identifier. Anything declared within the more specific scope will go away after the body finished (if completes, loop finishes, function returns).

A nested if/else or loop will have its own scope, the same rules apply. You are able to see everything from the higher level scope, but the higher level scope doesn't have access to the more limited scope.

8.Standard Library

The following function are built into the language and available to use at any time.

Number `link(`**Vertex** `a,` **Vertex** `b)`

Creates a unidirectional Edge link from Vertex a to Vertex b
Returns 1 on success

Number `bi_link(`**Vertex** `a,` **Vertex** `b)`

Create a bidirectional Edge link from Vertex b to Vertex b and back.
Internally, creates 2 Edges
Returns 1 on success

Number `weighted_link(`**Vertex** `a,` **Vertex** `b,` **Number** `weight)`

Create a weighted Edge from Vertex a to Vertex b
Returns 1 on success

Number `weighted_bi_link(`**Vertex** `a,` **Vertex** `b,` **Number** `weight)`

Create a weighted bidirectional Edge between Vertex a and Vertex b
Returns 1 on success

Number `add(`**List** `l, <`**Type**`> item)`

Add a new entry to the end of the list. Type must match the type of the list.
Returns 1 on success

Number `remove(`**List** `l,` **Number** `index)`

Removes the index from the list
Returns 1 on success

Number `list_length(`**List** `l)`

Returns the length of the list

Number `string_length(`**String** `s)`

Returns the length of the String

Number `print_number(`**Number** `n)`

Prints the number to STDOUT

Number `print_string(`**String** `s)`

Prints the string to STDOUT

Number `print_vertex(`**Vertex** `v)`

Prints the Vertex to STDOUT

Number `print_edge(`**Edge** `e)`

Prints the Edge to STDOUT

Number `print_list(`**List** `l)`

Prints the entries of the list to STDOUT

References

Dennis M. Ritchie, C Reference Manual
http://www.cs.columbia.edu/~sedwards/papers/cman.pdf