Aquila Documentation

Nicolas Reyland

February 2021

Contents

This paper is dedicated to the Aquila language. To learn how to use official the Aquila Interpreter, please refer to https://github.com/Nicolas-Reyland/Aquila. The project's code is well documented. If you are interested, please take a look! If you found any issues, please add one on github!

1 Introduction

Please note that some of the features described in this document are speculative, Aquila being in a very early development stage.

It is highly recommended that you have some basic programming and Computer Science knowledge before reading this document. You should be familiar with the following concepts:

- variables
- functions
- expressions ("words" or "values")
- sentences (or "instructions")
- variable declaration
- basic arithmetic and logical operations
- nested/conditional instructions
- exceptions/errors
- external code imports/inclusions

1.1 Aquila

The Aquila Programming Language is a Turing-Complete interpreted language made for Automatic Algorithm Visualization. It's use case is very specific. You can download an Interpreter here (in development). It is being developed in C#.

1.2 Purpose

Aquila has been made for the Code Vultus project. In fact, the Code Vultus program needs a complete access and control of it's input algorithm in order to generate a valuable graphical representation of it. To have this control, it has an internal Aquila Interpreter, which it uses to render it's final product (an algorithm visualization).

Aquila does not have a real purpose if used outside of Code Vultus. Except you think it has educative values.

2 Capability

Aquila supports all the basic needs of a programming language (variables, functions, loops, recursive functions, errors and much more):

2.1 Variables

Aquila cannot be declared as a strongly typed programming language, due to it's few variables types and it's flexibility in the multiple-variable structures (list). Here are the basic variable types:

- int
- list
- bool
- float

Any type of variable can be stored into a list, even other lists. Lists are not type-specific (you can have multiple variable-types in a single list).

2.2 Speed

Speed was not a main focus while creating Aquila. This doesn't mean that it will be abnormally slow. But please do understand that Aquila is meant to be paused, resumed, rewinded and fast-forwarded very often, and at the user's will! In this context, execution time becomes a very subjective notion. Calculations will be as fast as they are C# (Aquila is coded in C#), although nested loops will increase the execution time.

Generally, every small action is done quickly, but needs time to be initiated, evaluated and executed. So the less instructions you have to do a task, the better.

2.3 PreProcessor Macros

Since Aquila is meant for Code Vultus, it has some unusual PreProcessor Macros (they start with a "#" in the code):

- #name this is the name of the program
- #use path-to-an-external-file-to-include.aq
- #debug

The "#name" macro sets the name of the main program (useful in Code Vultus).
The "#use" macro includes external files to your code.
The "#debug" macro enables debugging for the interpreter.

3 Compatibility

You can translate the following programming languages into Aquila:

- C#
- C
- C++

These translations might not support every operation you use in these programming languages, so please avoid the usage of the following when giving your algorithm to an Aquila translator:

- comments
- any type that is not: int, bool, float (you can use arrays)
- any coding "shortcuts" (e.g. while (v++ < 5) => while (v < 5); v++)
- things that are done in one line instead multiple lines
- bit-by-bit operations

Most importantly, know that Aquila is really easy to learn, due to the few and basic keywords that exist. It is globally a very "small" programming language, mainly because of it's very specific use case. Very often, the best option is to directly write your algorithms in Aquila. To learn more about Aquila syntax, see the Syntax rules.

You can also translate your code into Aquila and manually correct the translation errors that have occured.

4 Programming

This section will give you examples and properties of the main aspects of the Aquila programming language. Keep in mind that some of these programming-related aspects are customizable through Code Vultus.

4.1 Syntax

Note that single line comment are done like this:

```
// this is a single line comment
```

And multiple line comments like this:

```
/** This is a multiple
line comment **/
```

4.2 Reserved Keywords

Here is the exhaustive list of all the reserved keywords (you cannot declare variables using their name, same goes for function definitions):
"declare", "int", "float", "bool", "list", "auto", "for", "end-for", "while", "end-while", "if", "else", "end-if", "var", "return", "void", "function", all arithmetic and logical operator characters

4.3 Variables

Variables are global in Aquila; there is no notion of variable scope of existence, except for functions. A better approach to variables is in development, but is not the main focus, because the input algorithms are supposed to be rather simple, so that you should not run out of variable names.

You have to declare variables with the "declare" keyword before you use or assign them. Although, you can assign their values at the same time as you declare them. If you declare a variable without giving it a specific value, you have to declare the type of the variable. If you assign a value to the variable, you don't have to describe the type. You can also use the "auto" type, which is equivalent to giving no additional information about the used type.

```
declare int a // only declared
declare b true // declared and assigned to "true" -> variable type is bool
declare float PI 3.1415f // declared and assigned float PI
declare hello 4f // declared a float (value is 4.0)
declare list l // declared a list
declare l2 [1, 2, 3, 4] // values separated by commas
declare list l3 [true, 1, [3, 4, 5], 4.20, [[[69]]], false]
// you can do this: lists are not "internally typed"

declare auto l4 ([4, 5, 6]) // declared and assigned a list
declare bool_example ([1, true, 1.4f[1])

// This doesn't work:
declare auto var_name // auto is not a valid var type
```

1. To assign integers, you just have to write the numerical value.
2. There are two ways to assign floats: either you write the float using the dot "." char in the assignment, or use the "f" suffix (e.g. -6f).
3. To assign basic boolean values to variables, you can either write "true" or "false".
4. For lists, you have to use brackets ("[" and "]") as prefix and suffix. Then, you need to separate your values by commas ",".

Of course, you can declare and assign variables to calculated values. Be sure to

write these values between parentheses or brackets (for lists) to separate them from the rest of your sentence:

```
declare a (1 + 2) // int: 3
declare int b (4 * var a) // int: 12
declare auto c ((var b - 1) / var a) // int: 3
declare d (3 } 4) // bool: false (3 >= 4)
declare e (4f / 2.5) // float: 1.6
declare f [var a, var b, var c, var d, var e]
// list: [3, 12, 3, false, 1.6]
```

Variable names can only contain ASCII characters except '#', ' ' (space), '(', ')', '[' or ']', as well as all the operators. They cannot be any of the reserved keywords. This means that, yes, technically you can do this:

```
declare int 0_/\$1"'9 /** You can, but this leads to an
                immediate public execution, in my opinion **/
```

Variable names cannot be any of the reserved keywords or arithmetic and logical operators. Accessing variables works through the "var" keyword:

```
var aquila_var // access the variable "aquila_var"
var l[var i] // access the variable "l" (a list) at index "i"
```

Assigning new values to variables works in the "classic" way:

```
var i = var x + 9
var j = 5
var l[0] = var l[var i % 5] - 6 // '%' => modulo operator
```

You can delete variables through the "delete_var" function:

```
delete_var(var l) // this works
delete_var(var l[0]) // this does not work
```

For casts, see "int2float" & "float2int".
All the variable types except the "list" support basic arithmetic and logic operations. List behaviour works as follows:

```
declare l [0, 1, 2, 4] // declare & assigned list named l
add_value(var l, 5)         // var l = [0, 1, 2, 4, 5]
insert_value(var l, 0, -1)  // var l = [-1, 0, 1, 2, 4, 5]
delete_value(var l, 1)      // var l = [-1, 1, 2, 4, 5]
declare list l2             // declare new list named l2
var l2 = copy_list(var l)   // var l = var l2 = [-1, 1, 2, 4, 5]
```

4.4 Instructions

4.4.1 Arithmetic & Logical Operations

Arithmetic operations, as well as logic operations have to be surrounded by spaces (left and right) to be recognized by the interpreter.

List of the arithmetic operators:

- '+' => classic addition
- '-' => classic subtraction
- '/' => classic division
- '*' => classic multiplication
- '%' => classic modulo

All theses operations work on integers ("int"). The modulo does not work on floating point numbers ("float"). The modulo has a greater "priority" than the multiplication, which has a higher priority than the division, which has a higher priority than the addition and subtraction (both are commutative operators so the priority between them isn't relevant).

```
declare int a
var a = 5 + 2 % 2 // var a = 7 % 2 = 1
```

The division on integers is an integer division ("7 / 2" is "3"). You cannot divide integers and floating point numbers, but you can cast "float2int" or "int2float" like this:

```
// declare some variables
declare i 5
declare f 1.5
declare int x
declare y -5.0

// the type of the variable doesn't change !
var x = float2int(var i) // var x = 1 (takes away the precision)
var y = int2float(var f) // var y = 5.0
float2int(var y) = 5 // does NOT work !
```

Here are examples of arithmetic operator applications:

```
declare a 5
var a += 8 // var a = 13
declare b [3.1415, 4, 5]
var b[0] = var a // var b[0] was a float, but is an int (13) now
var b[0] ++ // 14
var a = var b[2] % 2 // var a = 5 % 2 = 1
```

Due to the fact that all the operators are coded on one char, some logic operators are unusual. Here is the list of the logic operators (comparison operators are counted as logical operators):

- '^' => xor
- '&' => and
- '|' => or
- ':' => not equal
- '~' => equal
- '}' => greater or equal than
- '{' => less or equal than
- '>' => greater than
- '<' => less than

You may have noticed the absence of the '!' operator (logical "not"). It does exist, but it's usage is unique, because it is not an operator between two numbers (int/float) or booleans, but is applied to a single boolean value. The logical operator priorities are as foolow: "xor" over "and" over "or" (for boolean values). The other logical operators have less priority than those three, but there is no sense priority between them, because they compare numbers and result in a boolean value.
Here are examples of logical operator applications:

```
declare bool x
var x = 6 : 5 // true bc "6 not equal 5"
declare bool y
var y = var x ^ 4 > 5 // var y = true xor false = true
var y = !(var y) /** parentheses are mandatory -> "Syntax" subsection
                    var y = not true = false **/
```

4.4.2 Function calls

There are two types of function: "Void Function" and "Value Function". Each type has a different call method, called "void function calls" and "value function calls". Void functions don't have return values. They are called as instructions and should alter variables. Value functions return values which can be assigned to variables.
Functions are called in the "classical" manner:

```
// void function calls
void_function_name(arg_1, arg_2, ..., arg_n)
// value function call
var l = value_function_name(arg_1, arg_2, ..., arg_n) // var l is a list
```

For more info about existing (predefined) functions, function definition and function overwriting, see the Functions section.

4.4.3 Conditional Instructions

Conditional instructions access parts of code that you wrote only if certain conditions are met. They can also be called "nested instructions" There are three conditional instructions in Aquila: "if", "for" and "while".

They have a beginning and an end. They are used like this:

```
instr_name (arguments)
    instruction 1
    instruction 2
    instruction 3
end-instr_name
```

The "if" instructions only access the code you give them if the condition you have given them (included in the arguments, or basically the argument itself) is true. You can also give the "if" an "else" tag. If the "if" condition is false, then all the instructions following the "else" tag will be executed.

```
if (condition) // the tabs are not mandatory, they are used
    instruction 1 // to increase readability
    instruction 2
else
    instruction 3
    instruction 4
end-if
```

If the condition, which should be a boolean value, is set to true, the instructions 1 and 2 will be executed. If this is not the case, the instructions 3 and 4 are executed. Of course, you can also have an "if" instruction that doesn't have an "else" to it.

The while instruction is used like this:

```
while (condition)
    instruction 1
    instruction 2
end-while
```

While the "condition" is true, the instructions 1 and 2 are executed (in a loop). Note that if none of your instructions in the while loop alter the condition, this loop will run forever (instr 1, instr 2 -> instr 1, instr 2 -> ....)
A for loop is basically a while loop, but you give it certain instructions, as arguments, to ease the writing of certain tasks. It has a start-instruction, a stop-condition and a step-instruction.
The for instruction is used like this:

```
for (start-instruction, stop-condition, step-instruction)
    instruction 1
    instruction 2
    instruction 2
end-for
```

It is "translated" to this:

```
start-instruction
while (stop-condition)
    instruction 1
    instruction 2
    instruction 3
    step-instruction
end-while
```

This has many use-cases, like counting from 0 to the length of a list to access all the values of a list. Here is an example:

```
// This code prints every element of the list "l" on a new line
declare len length(var l)
for (declare index 0, var i < var len, var index ++)
    print_str(value at index )
    print(var index)
    print_str( is )
    print(var l[var index])
    print_endl()
end-for
/**
If the list "l" is [5, 1, 6, -6, 3],
the code prints this to the stdout stream:

value at index 0 is 5
value at index 1 is 1
value at index 2 is 6
value at index 3 is -6
value at index 4 is 3
**/
```

4.5 Functions

Functions are key elements to a programming language. You can define, use and overwrite functions yourself! Be sure that you have read the Function calls section before reading this one!

4.5.1 Default functions

Predefined void functions:

- return(expression)
- delete_var(variable)
- print(expression)
- print_str(string-like) [1]
- print_endl()

Predefined value functions:

- length(list variable)
- swap(list variable, integer index1, integer index2)
- copy_list(list variable)
- insert_list(list variable, integer index, variable)
- delete_list_at(list variable)
- float2int(float variable)
- int2float(int variable)

4.5.2 Function creation

4.5.3 Function overwriting

4.6 Exceptions & Errors

4.7 Includes (#use)

5 Interpreter

This section explains the inner workings of the Aquila interpreter. You do not need to read this in order to code better Aquila. This section is useful for debugging and curiosity purposes. The interpreter executes your code in 6 execution steps:

[1] There is no string variable type. That is why the argument is simply all the raw text that you give

5.1 Reading & Parsing

First, your source code is read (obviously). During this process, comments are filtered out from your source code. Every line is "purged"; unnecessary spaces are removed, as well as empty lines.

The following Exceptions may be raised during this execution step:

- InvalidCommentException

5.2 Macro Handling

During this execution step, all your macros (lines starting with a '#' character) are read and executed. Note that all your code still hasn't be touched: no syntax errors are raised during this step. To see how macros are understood, please read the PreProcessor Macros section.

The following Exceptions may be raised during this execution step:

- NoNameMacroException
- UseMacroFileNotFoundError

5.3 Function Definition

5.4 Instruction Initialization

Now is the time to look at your code: All the instructions (each line of your code, basically) will now be initialized. All the instruction-related syntax checks will be done, as well as variable and function creation: empty variables and functions are created, so that later function calls or variable accesses can be checked for undefined variable or functions names.

The following Exceptions may be raised during this execution step:

- SyntaxError
- TypeError
- UnknownTypeError
- DeclarationException
- DeclaredExistingVarException

5.5 Execution

The Execution step is the most crucial step in the whole program execution process. It works as follows:

It will execute every instruction in the code, one by one. Each instruction, when being executed uses some text-stored values (as is from the source code) which are named "expressions". Those expressions are then "parsed" (or "evaluated") into their respective values.

So, every instruction leads to one or more "Expression Parsings" (it is very rare that no parsing takes place in an instruction execution. one example is the "print_endl" void function: since it doesn't take any arguments whatsoever, there is no expression parsing). We can split the execution step into two separate sub-steps: the **instruction execution** and the **expression parsing**.

5.5.1 Instruction Execution

The instruction execution step executes the initialized instance of some instruction

The following Exceptions may be raised during this execution step:

- SyntaxError
- TypeException
- AssignmentError
- RuntimeError
- ReturnValueException[2]
- UnknownFunctionNameException (for void functions only)

5.5.2 Expression Evaluation

The following Exceptions may be raised during this execution step (specifically the evaluation step):

- SyntaxError
- TypeException
- RuntimeError
- InvalidIndexException
- UnkownVariableNameException
- UnknownFunctionNameException (for value functions only)

[2]This Exception should be catched internally

5.6 Code Return Value

6 Conclusion