Aquila Documentation

Nicolas Reyland

February 2021

Aquila

Code Vultus

Contents

This paper is dedicated to the Aquila language. To learn how to use official the Aquila Interpreter with your project, 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 logic
- nested/conditional instructions
- exceptions & errors

1.1 Aquila

The Aquila Programming Language is an imperative and procedural interpreted language made for Automatic Algorithm Visualization. It's use case is **very** specific. You can download the 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 or find it interesting.

2 Capability

Aquila supports all the basic needs of a programming language (multiple variable types, loops, recursive functions, errors and much more).

2.1 Possibilities

Aquila is Turing Complete. It supports basic list manipulation, as well as number variables. It is strongly typed, although it has some features from dynamically typed programming languages[1]. Its only multiple-value structure (list) supports multiple variable types as content. Here are the basic variable types:

- int
- float
- bool
- list

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). You may note the absence of the "string" type, which is implemented in most of the known programming languages. The reason for this is that Aquila is made for Code Vultus, where strings don't make sense.

2.2 Speed

Speed was not a main focus while creating Aquila. This doesn't mean that it will be abnormally slow. 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 in C# (Aquila is coded in C#), although tracing variables and enabling the "debug" or "trace_debug" options will increase the execution time drastically!

Generally, every small action is done quickly, but needs time to be initiated, evaluated and executed. Tests (which showed very variable results) approximated Aquila's execution speed to be around 100 times slower than python (execution time only)[2]. But again, Aquila is not designed for speed. Except you force it to, it will never actually run at its "top" speed (in the Code Vultus context).[3]

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
- #trace_debug

[1] You can, in a specific context, leave out the type of a variable in its declaration

[2] It takes less than 100 ms to sort a list of 100 big integers using the bubble sort algorithm.

[3] To minimize execution time, you should disable tracing, debugging, trace_debugging and maximize the use of predefined functions

- #trace_all
- #force_static_list
- #disable_variable_declaration

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#
- OCaml
- 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:

- any type that is not: int, float, bool (you can use arrays of those types -> list type)
- any non-universal "coding shortcuts" [4]
- things that are done in one line instead multiple lines[5]

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 and the PreProcessor Macros.

[4] e.g. "int[] array = new(){1}" instead of "int[] array = new int[] {1}" in C#
[5] e.g. "i++ < 4" instead of "i < 4", then "i++"

4.1 Syntax

The syntax is a mix between C and PHP (a weird mix, indeed!). Note that single line comment are done like this (they are being introduced straight away because they will be used a lot, startign now):

```
// 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 names):

- if
- else
- end-if
- for
- end-for
- while
- end-while
- function
- end-function
- recursive
- decl
- safe
- overwrite
- trace
- null
- int
- float
- bool
- list
- auto

4.3 Variables

Every variable must have a specific type, which cannot change (hence the "strong typing"). There are 4 variable types: "int" for integers, "float" for floating point numbers, "bool" for Boolean values and "list". There is another type: the "null" type, but its use-case is very strict and there is only one variable which has the null value: "$null"[6].

You have to declare variables with the "decl" (short for "declare") keyword before you use or assign them values. 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.

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

decl auto l4 ([4, 5, 6]) // declared and assigned a list
decl _e6xa2mple ([1, true, 1.4f[1])

// This doesn't work:
decl auto var_name // auto is not a valid var type
decl int safe [56] // "safe" is in the reserved keywords.
// You can use "_safe" tho
```

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, which is the same as -6.0).
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:

```
decl a (1 + 2) // int: 3
decl int b (4 * $a) // int: 12
decl auto c (($b - 1) / $a) // int: 3
decl d (3 } 4) // bool: false (3 >= 4)
```

[6] see the function section for more information about the "null" type

```
decl e (4f / 2.5) // float: 1.6
decl f [$a, $b, $c, $d, $e]
// list: [3, 12, 3, false, 1.6]
```

Variable names follow the 'classic' variables naming rules: they should only include alphabetical and numerical characters, as well as the underscore. The first character of the variable name should not be a numerical symbol.

```
/** Those are correct **/
decl _ 5
decl list t5 []
/** Those are incorrect **/
decl 6var_name 14f // cannot start with numerical character
decl bool zerzµ? false // cannot include 'µ' or '?'
```

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

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

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

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

You can delete variables through the "delete_var" function:

```
delete_var($l) // this works
delete_var($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:

```
decl l [0, 1, 2, 4] // declare & assigned list named l
add_value($l, 5)          // $l = [0, 1, 2, 4, 5]
insert_value($l, 0, -1)   // $l = [-1, 0, 1, 2, 4, 5]
delete_value($l, 1)       // $l = [-1, 1, 2, 4, 5]
$l[2] = $l[-1]            // $l = [-1, 1, 5, 4, 5]
/** $l[-1] points to the last value
    when the index is negative, the couting starts
    at the end of the list and goes towards the start **/
swap($l, 0, 3)            // $l = [4, 1, 5, -1, 5]
decl list l2              // declare new list named l2
$l2 = copy_list($l)       // $l2 = [4, 1, 5, -1, 5] ($l2 independant from $l)
```

4.4 Instructions & Operations

4.4.1 Declaration & Tracing

The declaration basis has already been covered in the Variables section, but it is good to note that a declaration is an instruction. Although, there is more:

There are two other types of variable declarations: the variable overwriting: and the "safe" declaration

Overwriting

Overwriting variables pretty much does what is says: you can completely re-declare variables. You cannot overwrite non-existing variables (you cannot use the "overwrite" keyword as a declaration).

```
// This works:
decl int i 9
overwrite bool i false
overwrite i [1, 4, -3]
// This doesn't work:
overwrite a 45 // the variable "a" does not exist
```

Safe Declaration

The "safe" prefix can do exactly what the "overwrite" keyword cannot: if the variable does not exist, it gets declared, and if it does exist, it gets overwritten:

```
decl a 5
safe decl a 45f
safe decl bool a false
safe decl list b // this works too, even though "b" is not yet declared
```

You cannot use any of these 2 keywords on traced variables !

Variable tracing lets you, well ... trace your variables! Every change will be saved, so that you can rewind some instructions. You can also trace functions:

```
decl x 0
trace $x // you can trace multiple variables by separating them by spaces
trace_func power // let's say you defined a power function earlier
$x = 4 // 0 -> 4
$x = power($x) // 4 -> 16
rewind 2 $x /** interpreter interactive mode only !!
            16 -> 4 -> 0 **/
trace_info  /** interpreter interactive mode only !!
            prints info about all the traced variables and functions
            **/
```

More info on tracing in the Interactive Mode and Debugging sections.

4.4.2 Arithmetic & Logical Operations

All the basic arithmetical and logic operations are supported in Aquila:

List of the arithmetic operators:

- '+' => addition
- '-' => subtraction
- '/' => division
- '*' => multiplication
- '%' => 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).

```
decl int a
$a = 5 + 4 % 2 // $a = 5 + 0 = 5
```

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
decl i 5
decl f 1.5
decl int x
decl y -5.0

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

Here are examples of arithmetic operator applications:

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

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

- '&' => and
- '^' => xor
- '|' => 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 has been implemented, but it's usage is unique: 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 follow: "and" over "xor" over "or" (for boolean values). The other logical operators have less priority than those three, but there is no order of priority between them, because they compare numbers, which means that they cannot be used one in parallel with another.
Here are examples of logical operator applications:

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

4.4.3 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, although they can be used as instructions too. Theoretically, you could assign the return value of a void function call to a variable, but you would not be able to use that variable.
Functions are called in the "classical" manner:

```
// void function calls
void_function_name(arg_1, arg_2, ..., arg_n)
// value function call
$l = value_function_name(arg_1, arg_2, ..., arg_n) // $l is a list
value_function_name(arg_1, arg_2, ..., arg_n) /** This works too,
    but if your function does not alter its input variables
    or does not print anything to the screen, this
    instructions is useless **/
```

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

4.4.4 Conditional Instructions

Conditional instructions only access parts of code that you wrote if a certain condition is 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 accesses the code you give them if the condition, when evaluated, 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
for (decl index 0, $i < length($l), $index ++)
    print_str(value at index )
    print($index)
    print_str( is )
    print($l[$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! Be sure that you have read the Function calls section before reading this one!

4.5.1 Predefined functions

Predefined void functions:

- return(expression) : return the expression in a function or as the algorithm result

- delete_var(variable) : delete a variable by name
- print(expression) : print the evaluated expression
- print_str(string-like) : print the raw value of the input[7]
- print_endl() : print the EOL char (\n)
- insert_list(list variable, int index, auto var) : insert 'var' at 'index' in 'list'
- delete_list_at(list l, int index) : delete the elemetn at 'index' in the list 'l'
- swap(list l, int index1, int index2) : swap the elements at 'index1' and 'index2' in 'l'

Predefined value functions:

- length(list variable) -> int : length of the list
- copy_list(list variable) -> list : copies the given list
- float2int(float variable) -> int : converts float to int
- int2float(int variable) -> float : converts int to float
- random() -> int : get a random int
- sqrt(int|float variable) -> int|float : square root function

4.5.2 Function definition

To define your own functions, you should use the "function" keyword. You function definition ends with the "end-function" keyword. Inside a function definition, you can use any Aquila instruction, except macro preprocessor instructions. This means that yes, you can define functions inside functions (which would be valid outside the function too)! Variables declared outside the function cannot be accessed from inside it. You have to pass those as arguments. The variables you give your functions are passed through "as-is", which means that any alteration to those variables inside your function will alter them in the global variable scope. If you do not wish to alter the "original" variables, you should copy them like this:

```
decl a_copy $a // declare a new variable with the value of "$a"
```

This would copy the value of "$a" into the new variable "a_copy". If you have not read the variables sub-section: you don't need to specify the variable type if the value is given in the declaration.

When defining a function, you have to specify the return-type, although the "auto" keyword is accepted: you would then be able to return any type. You also have to specify the name of the function, as well as the arguments it takes as parameters, without specifying their types:

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

```
// mathematical square function for integers
function int int_square(n)
    return($n * $n)
end-function
/** This function does not alter the input value,
    but the input value has to be an int, because
    the output value is expected as an int **/
// mathematical square function for
// anything that supports the '*' operator
function auto square(n)
    return($n * $n)
end-function
// This function can also take floats as arguments
decl int x int_power(5) // 25
decl float y
$y = power(4f) // 16.0
// random number between "a" and "b"
function int rand_range(a, b)
    decl int r random() // predefined random function
    decl int range ($a - $b)
    if ($range < 0)
        $range = -$range
    end-if
    if (a < b)
        decl min $a
    else
        decl min $b
    end-if
    $r %= $range
    $r += $min
    return($r)
end-function
// this function generates a random number between "a" and "b"
```

4.5.3 Recursive functions

Of course, you can define recursive functions, using the keyword "recursive":

```
// fibonacci sequence : recursive definition
function recursive int fib(n)
    if ($n < 2)
        return($n)
    end-if
    return(fib($n - 1) + fib($n - 2))
end-function
// use the function as any other function
```

```
decl int v fib(10) // $v = 55
```

Recursive functions work exactly as normal functions, which means that the variables are "reset" at each call.

4.6 Exceptions & Errors

5 Interpreter

This section explains the inner workings of the Aquila interpreter. You do not need to read this in order to code in Aquila, but it could be useful for debugging. The interpreter executes your code in 6 execution steps:

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:

- UnkownMacroError
- InvalidMacroParamterError
- FileNotFoundException

5.3 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 declaration: 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

5.4 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 values (as is, from the source code) which are named "expressions". An expression could be a numerical, boolean or any other form of value, which is expressed using any other expression: for example: "$5 + 6 < 4$" is a boolean value. 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**.

5.4.1 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[8]
- UnknownFunctionNameException (for void functions only)

5.4.2 Expression Evaluation

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

- SyntaxError
- TypeException
- RuntimeError
- InvalidIndexException

[8] This Exception should be catched internally

- UnkownVariableNameException
- UnknownFunctionNameException (for value functions only)

5.5 Code Return Value

6 Interactive Mode

7 Debugging

8 Develop in Aquila

9 Conclusion