

Manual for AGOS (AmaGle Operating System)
and AGRT (Amagle Runtime)

Ash Hodlan

Date: 13.08.1999

Contents

1	AGOS	3
1.1	Introduction	3
1.2	Solving Tasks	3
2	AGR	5
2.1	Introduction	5
2.1.1	Data types	5
2.1.2	Dictionary	6
2.2	Basic Operators	7
2.2.1	Assignment	7
2.2.2	Arithmetic Operators	7
2.2.3	Mod Assign	8
2.2.4	Comparison Operators	8
2.2.5	Boolean Operators	9
2.3	Basic Functions	11
2.3.1	<code>print</code>	11
2.3.2	<code>input</code>	11
2.3.3	<code>hasinput</code>	11
2.3.4	<code>output</code>	12
2.3.5	<code>not</code>	12
2.3.6	<code>string</code>	12
2.3.7	<code>int</code>	13
2.3.8	<code>.length</code>	13
2.3.9	<code>eval</code>	13
2.3.10	<code>sleep</code>	14
3	Flow Control	15
3.1	Introduction	15
3.2	Methods of flow control	15
3.2.1	<code>if, endif</code>	15
3.2.2	<code>else</code>	16
3.2.3	<code>while, endwhile</code>	17
3.2.4	<code>for, endfor</code>	18

4	Advanced Data Structures	20
4.1	Lists	20
4.1.1	Instantiation	20
4.1.2	Accessing elements	20
4.1.3	Modifying elements	21
4.1.4	<code>.length</code>	21
4.1.5	<code>.add</code>	22
4.1.6	<code>.remove</code>	22
4.2	Buffers	22
4.2.1	Instantiation	22
4.2.2	Pointer Instantiation	23
4.2.3	<code>.insert</code>	23
4.2.4	<code>.token</code>	24
4.2.5	<code>.base</code>	24
4.2.6	Pointer Arithmetic Operations	25
4.2.7	<code>.clear</code>	25
5	Functions	26
5.1	Function declaration and use	26
5.2	<code>return</code>	27
5.3	Recursion	27
5.4	Limitations	28
6	Network Communication	30
6.1	Description	30
6.2	Functions	30
6.2.1	<code>send</code>	30
6.2.2	<code>receive</code>	31
6.2.3	<code>last</code>	31
6.2.4	<code>broadcast</code>	31

Chapter 1

AGOS

1.1 Introduction

AGOS stands for AmaGle Operating System.

AGOS features all a modern server-cluster for development and production needs.

Once a machine has AGOS installed and the automated clustering tool AGCE (AmaGle Communication Environment) has been configured: clustering, updates, and route-setups happen automatically. This means code which is executed in an automated environment for communication can have generic names automatically set up depending on what the code is set to communicate with.

A tiered security system is also in place. A machine can only send instructions to machines one level above them and below. This means that data breaches are less likely to leak all the data in case they should happen.

1.2 Solving Tasks

As a programmer in AmaGle you must keep the following in mind when solving tasks:

- Unless you are writing a core system daemon (you probably won't), your process must terminate when it has solved its task. Your process will be re-launched when needed automatically.
- Unless the task has a fixed length, always rely on the `hasinput`-function to figure out when your program should ask for more input.

- Keep your skills up to date with our built-in tutorials and programming tasks.

Chapter 2

AGR

2.1 Introduction

AGR stands for AmaGle Runtime.

Programming on AGOS using the AGR features everything a modern programmer requires such as syntax highlighting, flow control, automatic indentation, a modern and fresh text interface, an easy to understand language, and much more! A simple hello world is as follows:

```
1 print(" Hello world")
```

Language keywords in code listings will be highlighted with bold and blue (such as `print` above).

2.1.1 Data types

AGR has the following data types:

Name	Description
<i>string</i>	A string declared using quotes surrounding a string of letters and numbers, such as "This is a string". To put a literal " inside a string, use \".
<i>integer</i>	A whole number declared using unquoted numbers, such as 42.
<i>boolean</i>	A true/false state type. Result of comparisons.
<i>list</i>	A list of any kind of data. More information can be found in section 4.1 - Lists.

Name	Description
<i>buffer</i>	A fast processing character-buffer for parsing messages and more. Further information about these can be found in section 4.2 - Buffers.
<i>bufferptr</i>	A pointer into a fast processing buffer containing an offset. Further information about these can be found in section 4.2 - Buffers.

Table 2.1: Data types

2.1.2 Dictionary

Reference for meaning of various words

Word	Meaning
<i>lhs</i>	Left hand side of an operation. In $a=3$, a is the lhs
<i>rhs</i>	Right hand side of an operation. In $a=3$, 3 is the rhs
<i>yield(s)</i>	The same as printing out

Table 2.2: Dictionary

2.2 Basic Operators

AGR features all basic operators you may ever need.

2.2.1 Assignment

= is the assignment operator. It takes the value on the right hand side and assigns it to unquoted name on the left.

```
1 a = 4
2 b = "hello"
3 print(b)
```

This example code assigns two different variables and then prints one of them.

2.2.2 Arithmetic Operators

+ - / * % are the arithmetic operators in the language.

```
1 a=3
2 b=7
3 print(a+b)
4 print(a-b)
5 print(b/a)
6 print(a*b)
7 print(b%a)
```

While the first four of these may be known to most people, the last one is a special one invented at AmaGle which may require further explanations. The %-operator is known as modulo. It returns the remainder of a division - in the example above, $7\%3$ is 1 because 6 is the highest whole number that is a multiple of three and smaller than seven, hence it returns $7-6=1$.

The +-operator is also defined for strings as a concatenation operator.

```
1 a="foo"
2 b="bar"
3 print(a+b)
```

This program will print the following:


```
1 foobar
```

All arithmetic operators are defined for `bufferptrs`, but those are defined in subsection 4.2.6

2.2.3 Mod Assign

All the arithmetic operators can be combined with a equality sign, such as `+=`. The `+=` operator sums the lhs and rhs and then assigns the result to the lhs. Returns the old value of lhs.

```
1 a = 13
2 a %= 2
3 a += 3
4 b = "Hello , "
5 b += "World"
```

After this, `a` is 4, while `b` is `Hello, World`.

```
1 a = 3
2 b = a += 3
```

After this, `a` is 6, while `b` is 3.

Additionally, `++` is equivalent to `+= 1`, and `--` is equivalent to `-= 1`

```
1 a = 3
2 a++
3 b = 10
4 c = b—
```

After this, `a` is 3, `b` is 9, and `c` is 10.

2.2.4 Comparison Operators

AGR has the following comparison operators: `==` `!=` `<` `>` `<=` `>=`

Operator	Description
----------	-------------

Operator	Description
<code>==</code>	Returns true if the sides are equal, false if the sides are unequal. Defined for <code>integers</code> and <code>string</code> . Example: <code>3 == 3</code> yields <code>true</code> , <code>"string" == "anotherstring"</code> yields <code>false</code> .
<code>!=</code>	Opposite of <code>==</code> .
<code><</code>	Returns true if the rhs is greater than the lhs, false otherwise. Defined only for <code>integers</code> . Example: <code>3 < 5</code> yields <code>true</code> , <code>5 < 3</code> and <code>3 < 3</code> yield <code>false</code> .
<code>></code>	Opposite of <code><</code> .
<code><=</code>	Same as <code><</code> , but yields true if the integers are equal - such as <code>3<=3</code> .
<code>>=</code>	Same as <code>></code> , but yields true if the integers are equal - such as <code>3>=3</code> .

Table 2.3: Comparison operators

Example code:

```

1 print(3 > 5)
2 print((6-3) <= 3)
3 print("Test" == "Test")

```

This code will yield the following:

```

1 False
2 True
3 True

```

2.2.5 Boolean Operators

AGR has the following boolean operators `||` `&&` `^^`

Operator	Description
<code> </code>	Returns true if either the lhs or rhs are <code>true</code> , else <code>false</code> .
<code>&&</code>	Returns true if both the lhs and rhs are <code>true</code> , else <code>false</code> .
<code>^^</code>	Returns true if the lhs and rhs are unequal, else <code>false</code> .

Table 2.4: Comparison operators

```
1 print(true || false)
2 print(false && true)
3 print(true ^^ false)
```

This code yields the following:

```
1 True
2 False
3 True
```

2.3 Basic Functions

2.3.1 print

`print` takes one parameter and prints it out to the machine's output terminal. It is intended as a help when debugging a program.

```
1 print (" Hello ")  
2 print (3)
```

This yields the following:

```
1 Hello  
2 3
```

2.3.2 input

`input` takes no parameters and yields a value that's being sent as input to the program.

```
1 a = input ()  
2 print (a)
```

This program retrieves what's been sent to it and prints it out.

2.3.3 hasinput

`hasinput` yields true as long as there's unretrieved input.

```
1 while (hasinput ())  
2     print (input ())  
3 endwhile
```

This code reads from input until there's nothing left.

2.3.4 output

`output` sends a value to the output. This function is relevant when working on tasks that run as subprocesses.

```
1 a = 4 + 5
2 output(a)
```

This program does a simple calculation and sends the result of it to the output.

2.3.5 not

Inverts a boolean - `true` becomes `false` and vice versa.

```
1 a = false
2 print( not(a) )
3
4 print( not(a) && true )
```

This program yields the following:

```
1 True
2 True
```

2.3.6 string

Converts a value to a string

```
1 a = 3
2 b = 4
3 print("The number is " + string(a) + string(b))
```

This program yields the following:

```
1 The number is 34
```

2.3.7 int

Converts a value to an integer

```
1 a = "3"  
2 b = "4"  
3 print (int(a) + int(b))
```

This program yields the following:

```
1 7
```

2.3.8 .length

Returns the length of a string, list or buffer.

```
1 a = "Test"  
2 b = [1,2,3, "hello", 5]  
3 c = buffer(100)  
4  
5 print( a.length() )  
6 print( b.length() )  
7 print( c.length() )
```

This program yields the following:

```
1 4  
2 5  
3 100
```

2.3.9 eval

`eval` executes the parameter as code, this allows highly dynamic code executions.

```
1 eval("print(\ Hello\ ")")
```

This program yields the following:

```
1 Hello
```

2.3.10 sleep

Halts execution for the given number of execution cycles

```
1 sleep(5)
2 print("I waited")
```

This program prints out the following five execution cycles:

```
1 I waited
```

Chapter 3

Flow Control

3.1 Introduction

Flow control is elementary in making computer programs be more flexible, easier to read and actually able to make decisions.

3.2 Methods of flow control

AGR has three flow-control methods.

3.2.1 `if`, `endif`

The format for an `if` is as follows:

```
1 if (CONDITION)
2   CODEBLOCK
3 endif
```

CONDITION is any expression that can be evaluated as true or false. Such as true, `a==4` or "Hello".

CODEBLOCK is any number of executable code lines. The code CODEBLOCK will only get executed if the CONDITION evaluates as true.

The conditional `if`-block is terminated with an `endif`.

Example:

```
1 a = 3
2
3 if (a == 3)
4     print("a is 3")
5     print("This also gets executed")
6 endif
7
8 if (a == 2)
9     print("This will not get printed because a is not 2")
10 endif
```

The output of this program is:

```
1 a is 3
2 This also gets executed
```

The final `print` will not happen as `a` is not 2.

3.2.2 else

The `else`-keyword can be used as follows:

```
1 if (CONDITION)
2     CODEBLOCK
3 else
4     CODEBLOCK
5 endif
```

When `CONDITION` evaluates to `false`, the second `CODEBLOCK` is executed instead of the first one.

Example code:

```
1 a = 5
2
3 if (a == 2)
```

```
4 | print("a is 2")
5 | else
6 | print("a is not 2, in fact it is " + string(a))
7 | endif
```

The output of this program is:

```
1 | a is not 2, in fact it is 5
```

3.2.3 while, endwhile

The `while`-keyword can be used as follows:

```
1 | while (CONDITION)
2 |     CODEBLOCK
3 | endwhile
```

This works almost exactly like the `if`-statement, except `CODEBLOCK` is run repeatedly until `CONDITION` is false.

Example code:

```
1 | i = 0
2 | while (i < 5)
3 |     print(i)
4 |     i++
5 | endwhile
```

This code will output:

```
1 | 0
2 | 1
3 | 2
4 | 3
5 | 4
```

The loop will no longer execute as the condition (`i < 5`) in the parentheses is no longer true.

It is important to remember the `i++`-bit. If you do the following:

```
1 i = 0
2
3 while (i < 5)
4     print(i)
5 endwhile
```

This code will never terminate as `i` never changes.

While can also be used to read arbitrary number of input parameters, see 2.3.3.

3.2.4 for, endfor

The `for`-keyword can be used as follows:

```
1 for (INITIALIZER ; CONDITION ; ENDFORACTION)
2     CODEBLOCK
3 endfor
```

`INITIALIZER` is executed when the `for`-statement is reached. `CONDITION` is then checked and if `true`, `CODEBLOCK` is executed.

When `endfor` is reached `ENDFORACTION` is executed, `CONDITION` is checked again and if true `CODEBLOCK` is executed again.

This repeats until `CONDITION` is `false`

Example code:

```
1 for (i = 0; i < 5; i++)
2     print(i)
3 endfor
```

This code is equivalent to the above example `while`-code. The output is the following:

```
1 0
2 1
3 2
4 3
5 4
```

CHAPTER 3. FLOW CONTROL 3.2. METHODS OF FLOW CONTROL

The following code prints every other number (starting from 0) up to 10 (but not including).

```
1 for (i = 0; i < 10; i+=2)
2   print(i)
3 endfor
```

Output:

```
1 0
2 2
3 4
4 6
5 8
```

Chapter 4

Advanced Data Structures

Buffers are used for faster processing of information stored in strings.

4.1 Lists

Lists contain any number of ordered values of any type. These are essential when performing operations on groups of values.

4.1.1 Instantiation

Lists are declared using brackets - [and] - with comma-separated elements inside.

```
1 list1 = []
2
3 a = 10
4 list2 = [1, 5, 7, "test", a]
```

4.1.2 Accessing elements

List elements are access using brackets - [and] - using integer values. Lists are indexed from zero.

```
1 a = [5, 7, 9]
2
3 print(a[1])
```

The above code will print the following:

```
1 7
```

4.1.3 Modifying elements

List elements are also modified using brackets.

```
1 a = [1, 2, 3]
2
3 a[2] = 10
```

The above code would change the list to contain the values 1, 2, and 10.

4.1.4 .length

Length is defined for lists in the common length function in 2.3.8. It is a crucial method when iterating over a list.

```
1 a = [1, 2, "Hello"]
2
3 for (i = 0; i < a.length(); i++)
4     print(a[i])
5 endfor
```

Output:

```
1 1
2 2
3 Hello
```

The code above makes it possible to print out any list as it handles an arbitrary list length.

4.1.5 `.add`

Any number of elements can be added to the end of a list using `.add`.

```
1 a = [1, 2, 3]
2 a.add(4, 5)
```

The resulting list will contain the values 1, 2, 3, 4, and 5.

4.1.6 `.remove`

Any number of elements can be removed from the list using `.remove` by index.

```
1 a = [1, 2, 3, 4, 5]
2 a.remove(0, 1)
```

The resulting list will contain the values 2, 4, and 5.

It is important to note that order of execution is important: `.remove(0,1)` is not the same as `.remove(1,0)`. In the first instance, the element in position 0 is removed first, the list positions are recalculated, followed by a removal of the element in position 1.

In the example above, `.remove(1,0)` would result in the list having the values 3, 4, and 5 instead.

4.2 Buffers

Buffers are fixed length fast access data structures used to process messages.

4.2.1 Instantiation

Buffers are instantiated using the `buffer`-function.

```
1 a = buffer(10)
2 print(a)
```

The output from this program shows what data the buffer is initialized with:

```
1 < Buffer length: 10 data: "          " >
```

4.2.2 Pointer Instantiation

Pointers are used to operate on buffers. These are instantiated the following way:

```
1 a = buffer(10)
2 ptr = a[5]
3 print(ptr)
```

After this code, `ptr` is a pointer pointing at position 5 in the buffer. This pointer can now be used with various functions.

```
1 < BufferPtr offset: 5 buffer: < Buffer length: 10 data: "
    " > >
```

4.2.3 `.insert`

`.insert` can be used to insert strings into the buffer through a pointer. The offset of the pointer is incremented by the size of the inserted string.

```
1 a = buffer(10)
2 ptr = a[5]
3 ptr.insert("test")
4
5 print(ptr)
```

As can be seen from this output, "test" has been inserted into the buffer and the offset is now 9.

```
1 < BufferPtr offset: 9 buffer: < Buffer length: 10 data: "
    test " > >
```


4.2.4 .token

`.token` returns the string that exists from the pointer's current offset to the token's position. It also increases the offset of the pointer used in the operation. If the token cannot be found, an empty string is returned instead and the pointer offset is set to out of bounds.

```

1 a = buffer(10)
2
3 ptr = a[0]
4 ptr.insert(" t1 | t2&")
5
6 ptr = a[0]
7
8 print(ptr.token("|"))
9 print(ptr.token("&"))
10 print(ptr)
11 print("Nothing will follow this:" + ptr.token("&"))
12 print(ptr)

```

Output:

```

1 t1
2 t2
3 < BufferPtr offset: 6 buffer: < Buffer length: 10 data: "
   t1 | t2&    " > >
4 Nothing will follow this:
5 < BufferPtr offset: 11 buffer: < Buffer length: 10 data:
   "t1 | t2&    " > >

```

4.2.5 .base

`.base` returns the buffer a pointer is pointing at.

```

1 a = buffer(10)
2 ptr = a[5]
3
4 print(ptr.base())

```

Output:

```
1 < Buffer length: 10 data: "          " >
```

4.2.6 Pointer Arithmetic Operations

All arithmetic operations are defined for pointers. They will affect the offset.

```
1 a = buffer(10)
2 ptr = a[0]
3
4 ptr += 8
5 print(ptr)
6
7 ptr -= 3
8 print(ptr)
```

Output:

```
1 < BufferPtr offset: 8 buffer: < Buffer length: 10 data: "
   " > >
2 < BufferPtr offset: 2 buffer: < Buffer length: 10 data: "
   " > >
```

4.2.7 .clear

`.clear` clears the contents of a buffer as if it had been just initialized.

```
1 a = buffer(10)
2 ptr = a[0]
3 ptr.insert("test")
4 a.clear()
5
6 print(a)
```

Output:

```
1 < Buffer length: 10 data: "          " >
```

Chapter 5

Functions

Functions are useful tools for re-use in a program. Functions are isolated snippets of executable code that can be used from elsewhere in the program.

5.1 Function declaration and use

The general template for declaring a function is:

```
1 func FUNCNAME(PARAM1, PARAM2 ... , PARAMN)
2     CODEBLOCK
3 endfunc
```

It can then later be called using:

```
1 FUNCNAME(PARAM1, PARAM2 ... , PARAMN)
```

A function can be declared and used as follows:

```
1 func myFunc(v)
2     print("I am a function")
3     print("Value received: " + string(v))
4 endfunc
5
6 myFunc(3)
7 myFunc("wee, functions!")
```

Output:

```
1 I am a function
2 Value received: 3
3 I am a function
4 Value received: wee, functions!
```

Instead of writing four `print`-statements, only two were needed.

5.2 return

`return` can be used to pass a value back to the outside of the function.

```
1 func double(v)
2     return(v+v)
3 endfunc
4
5 print(double(3))
6 print(double("ring"))
```

Output:

```
1 6
2 ringring
```

5.3 Recursion

You may call any function within a function:

```
1 func recursiveCountdown(i)
2     if (i <= 0)
3         print("Launch!")
4     else
5         print(i)
6         recursiveCountdown(i-1)
7     endif
8 endfunc
```

```
9  
10 recursiveCountdown(5)
```

Output:

```
1 5  
2 4  
3 3  
4 2  
5 1  
6 Launch!
```

5.4 Limitations

A function cannot access a variable that isn't passed or declared inside the function.

The following code will fail when `a` is not found inside the function:

```
1 a = 3  
2  
3 func test()  
4     print(a)  
5 endfunc  
6  
7 test()
```

The following code will not output what the user might expect:

```
1 a = 3  
2  
3 func test()  
4     a = 6  
5 endfunc  
6  
7 test()  
8 print(a)
```

Output:

1

3

Chapter 6

Network Communication

6.1 Description

Network communication is simplified by AGCE to allow simple names to be used in code. While many things can be magically fixed by one's development environment, we can't remove constraints on communication and hence messages sent will not be received by the recipient at once. You may assume it takes roughly the time it takes to `sleep(1)` for a message to be delivered.

6.2 Functions

6.2.1 `send`

`send` takes up to two parameters, at least one: the first parameter is the message content, the second parameter destination, defaults to the other/next machine in the environment if not provided/empty.

```
1 send(" Hello ")  
2 send(" Anyone there?", "server")
```

This code will send `Hello` to the other machine in the network, followed by `Anyone there?` to a machine named "server" (which happens to be the one which also received the `Hello`-message).

6.2.2 receive

`receive` takes up to two parameters: the first parameter is a target variable where the received message will be stored, the second parameter is the source we want to listen to messages from, defaults to any machine if not provided/empty.

```
1 receive(a)
2 receive(b, "client")
3 print(a)
4 print(b)
```

If this code was run at the same environment as the example in the `send`-description, you would get the following output:

```
1 Hello
2 Anyone there?
```

It is important to note that `receive` does an implicit `sleep(1)` whenever it cannot receive a message.

6.2.3 last

Whenever a message is received, the `last`-variable is set to the name of the machine that the message was received from. To reply to a message in an environment with multiple machines, one can do the following:

```
1 receive(a)
2 send(a + " received!", last)
```

6.2.4 broadcast

Broadcast is a function that can be used to send a message to all machines in an environment:

```
1 broadcast("I am here!")
```