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

Ash Hodlan

Date: 13.08.1999

Contents

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 fixed length, always rely on the `hasinput`-function to figure out if your program should ask for more input.

- If a task asks you to operate on a given number of inputs, you may assume that the program will only be given that number of inputs.
- 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") $ Prints out "Hello world"
```

Language keywords in code listings will be highlighted with bold and blue (such as `print` above). When code-relevant characters/words appear in the text, they will use `this font`. Comments are declared using `$`. Anything written in capital letters inside a code example is a template (such as `CONDITION` or `FUNCNAME`), meaning that something needs to be written there instead by the programmer in the actual code.

2.1.1 Dictionary

Reference for meaning of various words

Word	Meaning
concatenation	The act of appending a string (or words) to another, such as `"a" + "b" = "ab"`. `+` is, in addition to being the addition-operator for numbers, also the concatenation operator for strings.

Word	Meaning
instantiation	The act of creating something, such as `b = buffer(100)` instantiates a buffer of length 100.
lhs	Left hand side of an operation. In `a=3`, `a` is the lhs
parsing	The act of interpreting text as code, converting it into something a machine can execute. Can also mean to read something as something else: "parsing an string as an int".
rhs	Right hand side of an operation. In `a=3`, `3` is the rhs
scope	A scope can be thought of as an environment. The global scope is default one where your operations happen. If you call a function, you will get a temporary scope that is discarded after the function returns. See section 5.4 and section 6.2 for more information.
yield(s)	The same as printing out

Table 2.1: Dictionary

2.1.2 Data types

AGR has the following data types:

Name	Description
string	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 `\"`. To put a literal `\` inside a string, use `\\`.
integer	A whole number declared using unquoted numbers, such as `42`.
boolean	A true/false state type. Result of comparisons.
list	A list of any kind of data. More information can be found in section 4.1 - Lists.
buffer	A fast processing character-buffer for parsing messages and more. Further information about these can be found in section 4.2 - Buffers.
bufferptr	A pointer into a fast processing buffer containing an offset. Further information about these can be found in section 4.2 - Buffers.
null	A special value returned from failed operations and functions that do not return anything. Synonymous with "empty" in everyday language. Evaluates to `false` if evaluated as a boolean. Can be declared by just writing an unquoted `null`. Not to be confused with 0, a numeric value.

Name	Description

Table 2.2: Data types

2.1.3 Key Bindings

AGOS has the following key bindings:

Key	Action
Ctrl-D	Delete everything in input area
Ctrl-O	Clean output area.
F11	Decrease font size.
F12	Increase font size.

Table 2.3: Key bindings

DEBUG VERSIONS ONLY (IF YOU CAN READ THIS, IT PROBABLY WORKS).
THIS WILL NOT BE A PART OF THE FINAL MANUAL.
Some of these will be moved up to the actual key bindings later as they are properly implemented (like ctr+c).

Key	Action
Ctrl+C	Rudimentary copy. Copies the entire code
Ctrl+V	Rudimentary paste. Removes the entire code and inserts clipboard
U	Node map: unlock node
L	Node map: lock node
Ctrl+U	Node map: Unlock entire map
Ctrl+L	Node map: Lock entire map
Ctrl+M	Reset machine (if something is really weird)

Table 2.4: Key bindings for debug

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 `bufferptr`s, 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 `4`, `b` is `9`, and `c` is 10.

2.2.4 Comparison Operators

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

Operator	Description

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

Table 2.5: 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
`||`	Returns true if either the lhs or rhs are `true`, else `false`.
`&&`	Returns true if both the lhs and rhs are `true`, else `false`.
`^^`	Returns true if the lhs and rhs are unequal, else `false`.

Table 2.6: 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 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` (can also be written as `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
```

If given a `null`, will return the string `<null>`.

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
```

If given a `null`, will return the integer `0`.

2.3.8 `eval`

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

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

This program yields the following:

```
1 Hello
```

2.3.9 `sleep`

`sleep` halts execution for the given number of execution cycles

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

This program prints out the following after doing nothing for five execution cycles:

```
1 I waited
```

2.3.10 `secret`

`secret` takes an int-parameter and returns a security token based on the access level of the machine.

```
1 print(secret(3))
```

In this particular case, the output was the following:

```
1 O>NCXKb
```

However, you will find that the output on your machine will be different - including each time you run your program.

2.4 Dot Functions

Dot functions are functions take the format `VALUE.FUNCTION(PARAMS)` or `VARIABLE.FUNCTION(PARAMS)`. They've been designed this way as they are intuitively associated with the data type that they operate on. Dot functions associated with advanced data types can be found in chapter 4.

2.4.1 `.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
```

2.4.2 .substring

`.substring` returns a part of a string. It takes a starting offset, followed by a length as parameters. The length-parameter is optional and if not provided, the rest of the string will be returned.

```
1 print("Hello".substring(1, 3))
```

Output:

```
1 ell
```

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` (can also be written `endIf`).

Example:

```
a = 3

if (a == 3)
  print("a is 3")
  print("This also gets executed")
endif

if (a == 2)
  print("This will not get printed because a is not 2")
endif
```

The output of this program is:

```
a is 3
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:

```
if (CONDITION)
  CODEBLOCK
else
  CODEBLOCK
endif
```

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

Example code:

```
a = 5

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 `elsif`

`elsif` can be to check other conditions in an `if`.

```
1  if (CONDITION)
2    CODEBLOCK
3  elsif(CONDITION)
4    CODEBLOCK
5  elsif(CONDITION)
6    CODEBLOCK
7    ...
8  else
9    CODEBLOCK
10 endif
```

If the first `CONDITION` is `false`, each `CONDITION` inside every `elsif` will be checked in turn from the top. If one is found to be `true`, the corresponding `CODEBLOCK` will be executed and execution will then jump to the `endif`. If no `CONDITION` is `true`, the `else-CODEBLOCK` (if it exists) will be executed instead.

```
1  a = 4
2  if (a == 3)
3    print("a is 3")
4  elsif (a < 10)
5    print("a is less than 10")
6  elsif (a < 20)
7    print("a is less than 20")
8  else
9    print("a is greater than 20")
10 endif
```

Output:

```
a is less than 10
```

3.2.4 `while`, `endwhile`

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

```
while (CONDITION)
  CODEBLOCK
endwhile
```

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

Example code:

```
i = 0
while (i < 5)
  print(i)
  i++
endwhile
```

This code will output:

```
0
1
2
3
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:

```
i = 0

```

```
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 an arbitrary number of input parameters, see subsection 2.3.3.
`endwhile` can also be written as `endWhile`.

3.2.5 `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 `CODEBOCK` 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
```

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
```

`endfor` can also ben written as `endFor`.

3.2.6 `Break`

`break` jumps to the end of the inner-most `for`/`while`-loop the execution is currently in.

```
1 for (i = 0; i < 10; i++)
2   if (i == 5)
3     print("Break")
4     break
5   endif
6   print(i)
7 endfor
```

Output:

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

3.2.7 `Continue`

`continue` jumps to the top of the inner-most `for`/`while`-loop the execution is currently in. If it is a for-loop, `ENDFORACTION` is also triggered.

```
1  for (i = 0; i < 10; i++)
2    if (i%2 == 0)
3      print("Continue")
4      continue
5    endif
6    print(i)
7  endfor
```

Output:

```
1  Continue
2  1
3  Continue
4  3
5  Continue
6  5
7  Continue
8  7
9  Continue
10 9
```

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. They 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 accessed 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 subsection 2.4.1. 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, quick access data structures that are 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:

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

4.2.2 Pointer Instantiation

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

```
a = buffer(10)
ptr = a[5]
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.

```
< 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.

```
a = buffer(10)
ptr = a[5]
ptr.insert("test")

print(ptr)
```

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

```
< 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 provided token's position. It also increases the offset of the pointer used in the operation to the position after the provided token. If the token cannot be found, a `null` is returned instead, and the pointer's 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("<null> will follow this: " + string(ptr.token("&")
       ))
12 print(ptr)
```

Output:

```
1  t1
2  t2
3  < BufferPtr offset: 6 buffer: < Buffer length: 10 data: "
       t1|t2&     " > >
4  <null> will follow this: <null>
5  < BufferPtr offset: 10 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:

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

4.2.6 Pointer Arithmetic Operations

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

```
a = buffer(10)
ptr = a[0]

ptr += 8
print(ptr)

ptr %= 3
print(ptr)
```

Output:

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

4.2.7 `.offset`

`.offset` returns the offset of a BufferPtr.

```
b = buffer(10)
ptr = b[5]
ptr -= 1

print(ptr.offset())
```

Output:

```
4
```

4.2.8 `.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("whee, functions!")
```

Output:

```
I am a function
Value received: 3
I am a function
Value received: whee, functions!
```

Instead of writing four `print`-statements, only two were needed.
`endfunc` can also be written as `endFunc`.

5.2 return

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

```
func double(v)
  return(v+v)
endfunc

print(double(3))
print(double("ring"))
```

Output:

```
6
ringring
```

5.3 Recursion

You may call any function within a function:

```
func recursiveCountdown(i)
  if (i <= 0)
    print("Launch!")
  else
    print(i)
    recursiveCountdown(i-1)
  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. In technical terms: functions have a scope and cannot access variables in the global scope or other functions' scopes.
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

Special Behaviours

This chapter documents important special behaviors in the language.

6.1 Integer limits

Integers in AGR must be between `-214783648` and `214783647` (inclusive) when declared as a variable or parsed using the `int`-function.
Going above or below these limits in calculations will cause the number "wrap around".

```
print(-2147483648 - 1)
```

Output:

```
2147483647
```

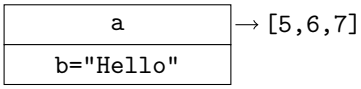
6.2 References

It is important to note that any data structure described in chapter 4 are pass by reference, while all other data types are pass by value.

Consider the following code:

```
1 a = [5, 6, 7]
2 b = "Hello"
```

The situation in memory will look like this:



What this means is that the string `Hello` is contained within the scope, while the list `[5,6,7]` is outside the scope and is being referenced by `a` from inside the scope.

This has consequences for how these values behave in various situations.

6.2.1 Assignment

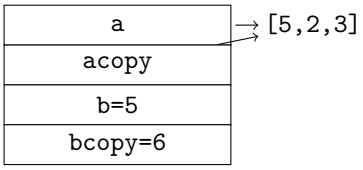
Read the following code:

```
1  a = [1,2,3]
2  b = 5
3
4  acopy = a
5  acopy[0] = 5
6
7  bcopy = b
8  bcopy += 1
9
10 print(a)
11 print(b)
```

The output is the following:

```
1 < List length: 3 data: [5, 2, 3] >
2 5
```

As you can see, the first number in `a` changed to `5` when it was modified through `acopy`. `b` on the other hand, was not changed.
To understand why, look at this figure that illustrates what the memory looks like after this code has been run:



However, consider the following code:

```
1  a = [1,2,3]
2
3  b = a
4  a = ["Hello"]
5
6  print(a)
7  print(b)
```

The output from this code is:

```
1  < List length: 1 data: [Hello] >
2  < List length: 3 data: [1, 2, 3] >
```

The reason `b` is not `a` is because `a` has been replaced with a reference to a different list.

6.2.2 References And Functions

Consider the following code:

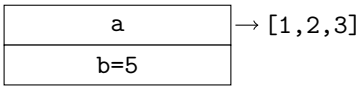
```
1  func listMod(l)
2    for (i = 0; i < l.length(); i++)
3      l[i] += 1
4    endfor
5  endfunc
6
7  func intMod(i)
8    i += 1
9  endfunc
10
11 a = [1, 2, 3]
12 b = 5
13
```

```
14 listMod(a)
15 intMod(b)
16
17 print(a)
18 print(b)
```

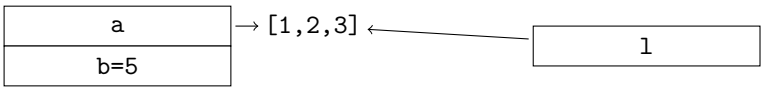
The output from this is:

```
1 < List length: 3 data: [2, 3, 4] >
2 5
```

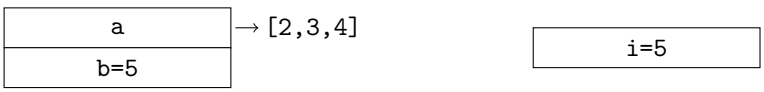
This is what the situation looks like outside the functions:



The column represents the scope we're in. Notice how `[1,2,3]` is outside of the scope (with `a` pointing at it), while `b` is inside the scope.
So when we call `listMod(a)`, this happens:



The left box represents the global scope, the right box represents the scope inside of `listMod`.
This means that any changes that happen to `l` actually happen to the same list that `a` is referencing.
Meanwhile, this happens inside `intMod`:



This means that any changes that happen to `i` inside of `intMod` do not apply to `b` in the global scope.

Chapter 7

Network Communication

7.1 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. Therefore, messages sent will not be received by the recipient at once. You should assume it takes roughly the time it takes to `sleep(1)` for a message to be delivered.

7.2 Functions

7.2.1 `send`

`send` takes up to two parameters, but at least one. The first parameter is the message content. The second parameter is the 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 the the one which also received the `Hello`-message).

7.2.2 `receive`

`receive` takes up to two parameters, but at least one. 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, and 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 in 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 to try again in one execution cycle. This repeats until it succeeds at receiving a message.

7.2.3 `nreceive`

Same as `receive`, but will not sleep if nothing is received. Instead, the variable is set to `null`.

7.2.4 `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)
```

7.2.5 broadcast

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

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