This document describes briefly the language “Atom”, the assembly language it compiles to, the CPU (“cortex”) executing the assembly and how this CPU is integrated into the VM “Up”.

All the informations given here, including: … well, everything, are purely indicative and may change at any time without warning.

This document does NOT cover the filesystem of “Up”, the archive format, “Jade”, the image format, the APIs of “Up” or its physical specifications and limitations.

Even what is covered is only briefly described and omits entire blocks of informations that would be required to build a compliant system.

A program will need to interact with the console and will likely be split in different parts. To import functions from other modules, use the “import” instruction:

```
I
  foo: path/to/module/function1,
  bar: path/to/other_module/function2
;
```

In that example, we import two functions from other module and assign them to the symbols “foo” and “bar”. Locally, those symbols will be used to call these functions, regardless of the name they in their respective modules.

You can create aliases for types. To create alias, use the “typedef” instruction:

```
T
  foo: type1,
  bar: type2
;
```

In that example, we create two aliases, foo and bar, to different types. What types can be is described below.

You’ll need variables. Variables are declared with the “variable” instruction:

```
V
  foo: type1,
  bar: type2
;
```

In that example we declare two variables named foo and bar. By default, variables can be declared inside functions. A language extension called “module memory” allows variables to be declared outside functions. Such variables are only accessible by functions of the same module.

To assign a variable, use the “assign” instruction:

```
A
  foo: expression1,
  bar: expression2
;
```

In that example, expression1 is assigned to foo and expression2 is assigned to bar. Note that the assignment is not sequential. The instruction

```
A
  foo: bar,
  bar: foo
;
```

swaps foo and bar. The semantic is that all expressions and destinations are evaluated first, without modifying anything. When all the informations are known, the actual assignment is performed.

Assignments can also have “with” clauses. The syntax is as follow:

```
A
  foo: expression1,
  bar: expression2
W
  bla: expression3,
  bli: expression4,
  blu: expression5
W
  blou: expression6
;
```

In that example, the symbols “bla”, “bli”, “blu” and “blou” **aren’t** variables. They are symbols only existing for the duration of the assignment. Their type is deduced from the expression they are assigned to.
Here, “expression6” is evaluated first. Once its value is known, “expression3”, “expression4” and “expression5” are evaluated. “blou” can appear in those expressions, as its already known.
Once they are all known, “expression1”, “expression2”, “foo” and “bar” are evaluated. “bla”, “bli”, “blu” and “blou” can appear in those.
Only one expression with side effect can appear in each “with” clause, and will be evaluated after all the expressions without side effect of the same “with” clause.

To declare a function, use the “function” instruction:

```
F
  foo:param1:result1
    instructions
  end,
  bar:param2:result2
    instructions
  end
;
```

In that example, we declare two functions named “foo” and “bar”.
Each function (here and in general in this language) takes exactly one parameter and returns only one value. They can have a composed type though.
In a function, the parameter is known under the symbol “in” and the return value is called “out”.
To export a function, add an “E” before its name:

```
F
E foo:param1:result1
    instructions
  end
;
```

Types can be basic or composed. Basic types are: u8, i8, u16, i16, u32, i32, u64, i64, f32, f64, bool and resource<X>. Strings are a type of resource.

Implicit casts exist between numbers.
There is two kind of composed type: vectors and tuples.
Vectors are n elements of the same type, and are accessed with squared brackets.
Tuples are named elements of possibly different types, accessed by ".name".

A vector is declared like this:
```
(1, 2, 3)
```

A tuple is declared like this:
```
(foo: 1, bar: 2, baz: 3)
```

Operations (and assignments) are valid for composed types if they are valid for their the elements that compose them. For binary operators, the two operands must have the same structure.

```
(a, b) + (c, d)
```
is valid if and only if

```
(a+c, b+d)
```
is valid, and yields this result. The same is true for other operators.

The following operations are **not valid**
```
(1, 2) + (3, 4, 5) // not the same structure
(a: 1, b: 2) + (a: 3, b: 4, c: 5) // not the same structure
(a: 1, b: 2) + (b: 3, a: 4) // not the same structure
(a: 1, b: true) + (a: 2, b: 3) // "true + 3" isn't valid
```

A tuple's type is declared with the following syntax:
```
(a:u8, b:i16)
```

A vector's type is declared with the following syntax:
```
u16^5
```

There is no "pointer" or "reference" mechanism (which is why "module memory" is important to avoid passing the entire game state by copy every time you invoke a function working on it).

⚠ The following is still highly debatable as I don't like it ⚠

I made several dialects of Atom to toy with those concepts. Currently, I didn't like any of them. Concepts presented here are not necessarily in all dialects I made, so they may appear to be redundant. However, all have been implemented at least once.

Loops can be made in two ways:

```
O
  instructions
;
```

It’s a basic “while(true)” loop.
The other solution is to use the “restart” instructions, that places the instruction cursor at the beginning of the function.

Conditional statement are done with the “match” instruction:

```
M expression
  case e1:
    instructions1
  case e2:
    instructions2
;
```

It works with composed and basic types.

Alternatively, you can use the plain ol’ good “if” construction:

```
if <boolean expression1> then
  instructions1
elseif <boolean expression2> then
  instructions2
elseif …
else
  instructions3
end
```

This section describes the compilation process.

The code is turned into an AST with “peggy” (the successor or peg.js). There isn’t much specific to say about it, except for expressions.
Expressions are defined as “expression = fragment (operator fragment)*”, i.e. a sequence of fragment separated by operators, a fragment being a literal, a variable, or something in parentheses etc.
This produces an array, which is passed through a shunting yard algorithm to handle operator precedence correctly.

The next big step **would** be type checking and implicit casting, but something is preventing us from going there directly: we don’t have the types (parameter and return type) of the imported functions.

The module waits for other modules to arrive until it has all the informations it needs to proceed to the next steps. When leaving this first step, it acquires the “last_known_global_id” described in the section about functions, and assign local ids to all its functions.

The type checking and implicit casting is a bit more complex than what you would imagine because implicit cast can be added between composed types:

```
(a:1, b:2.3) + (a:4.5, b:6)
```

results in a tuple in which both “a” and “b” are floats, which is different from both the types of the operands of “+”.
If it’s not a complex step, it’s still a long one.

The next step turns the now-valid AST into assembly code. Some minor optimizations are performed.

The next and last step builds the final archive. It implies building several tables (global function tables, resource tables), converting images and generating the necessary plumbing.

The game can be started as soon as the main module finishes the last step. It may not be the first module finishes it.

The section describes the assembly.

The assembly uses a very limited number of opcode. It is more easily understandable is read in conjunction with the description of the machine.
Note that information can move from registers to the stack but not the other way around. This is by design.

For readability, commands are referenced by their name (as opposed to “by their letter”).

The commands “pick_1”, “pick_2” and “pick_3” pick respectively “param1”, “left” and “right”. The picked register is set to 0.

The commands “seti_0” to “seti_15” push bits into the picked register, shifting to the left (in a big endian representation) the bits already there.

The command “cast” has two different usages, both relying on the content of the first byte of param1 (the least significant byte). This byte is split in two nibbles representing the destination type (the most significant nibble) and the source type (the least significant nibble).
If the source is 0, the command only changes “mod”.
If the source is not 0, the value at address “right” is casted from “source” to “destination” and stored at address “left”. Then, “right” is incremented by the size of “source” and “left” is incremented by the size of “destination”.

The commands “opa_<X>” and “opr_<Y>” performs operations on values at address “left” and at address “right” as if both operands where of type “mod” and stores the result at address “left”.

The command “jumpif_Y” set “pc” to “left” if the byte (least significant) in “right” is not 0.
The command “jumpif_N” set “pc” to “left” if the byte (least significant) in “right” is 0.

The command “call” performs a function call. The address at which call the function must be stored in “left” and the magic number of the function must be stored in “right”.
The details about calls are given in a dedicated section below.

The command “sb” stores the byte (least significant) in “right” at address “left”. Then, the most significant byte of “right” is copied in its least significant byte and “left” is increased by one. This process is repeated “param1” more times.

This section describes the machine.

The machine has three registers of 2 bytes: “param1”, “left” and “right”.
It has several other informations that aren’t registers:
1/ the “pc” which holds the next instruction to execute.
2/ the “fp” which hold the function offset on the stack.
3/ the “mod” which contains the type the ALU will use while performing operations.
4/ the “permits” which contains the number of instructions the program will be allowed to run.
5/ the “cmds” which contains the buffer of instructions to execute.

It also has a “status”, which is a structure contains informations about the state of the machine.

The CPU works in a “give up” fashion: when a function call is encountered, when an illegal opcode is met, when a function returns, when the program exhausts its permits … the CPU “gives up”.
It writes the reason in the status then stops and gives back the handle.

The rest of the console can use the informations in the status to take measures and, eventually, restarts the CPU for more instructions.

This allows for a CPU agnostic to mechanisms of functions calls, errors recovery, illegal opcode handling etc.

Another information could be added representing the “ring” of the program (as in “ring-0”).
Currently the OS is in Js entirely (and so largely isolated from the programs running on the CPU) but it could change in the future.

The section describes function calls

The call convention is:
1/ at address ADDR, let enough space to store the return value of the function.
2/ at address ADDR + (size of the return value) store the parameter of the function.
3/ store ADDR in register “left”.
4/ store the magic number of the function in register “right”
5/ use the opcode “call”.

Registers are saved and restored by the system. They are set to 0 from the point of view of the called function.

There is three types of functions
1/ local functions
2/ global functions
3/ system functions

“System functions” are a specific type of global functions.
The distinction between local and global functions resides in the id of the function (the “magic number”): each module has an information attached to it, “last_known_global_id”, which acts like a barrier between local and global functions. Any id smaller than or equal to this “last_known_global_id” identifies a global function. Any id greater is considered local.

Local functions are functions located in the same module as the one currently executed. A table inside the module associated local ids to offsets and lengths inside the module.

For global functions, a table associate global ids to pairs of <module_id, local_id>.

To clarify: there is one “local functions” table per module, but only one “global functions” table per program.

A global function is a function located in another module.

A system function is a global function whose “module_id” is -1 (or its positive 2-complement representation rather). It means that the separation of system functions into system modules is only cosmetic: there is actually only one system and one system module, containing all the system functions.
(because ids for modules are distributed from 1 and going upward, it would be possible to have several system module without changing the code too much).

This allows to store all the system functions in one big array, indexed by the local_id.

This section contains a BNF-like description of a dialect of Atom. It has been made by stripping almost all the actions of the PEG parser of this dialect.

For reminder:
“X*” means “X, 0 or plus times”.
“X+” means “X, 1 or plus times”.
“X?” means “X, 0 or 1 time”.
Strings enclosed by simple quotes (‘) are literals.
“X/Y” means “X or Y”.
“[a-x]” means “any character from a to x”.
“[^abc]” means “any character that isn’t a, b or c”.
“a:X” means “X will be named a in the action”. I let that in the rule for expression.
“& X” is a trick used in peggy (the peg parser generator) to lookup ahead tokens and fail if they are found. It’s used in “_” to say “A multi-line comment is /* then (anything but */) then */”.

The symbol “_” is a rule like any other. However, it is so commonly used that I think describing what it means is important: it’s whitespaces. More precisely, it’s “any sequence of whitespaces, multi-lines comments and single-line comments”.

```
program = _? top_level_instructions

top_level_instructions = top_level_instruction*
instructions = instruction*
instructions_block = '{' _? instructions '}' _?

top_level_instruction =
  import_declarations
/ typedef_declarations
/ variable_declarations
/ function_declarations

import_declarations =
 'I' _ import_declaration (',' _? import_declaration)* ';' _?

typedef_declarations =
 'T' _ typedef_declaration (',' _? typedef_declaration)* ';' _?

variable_declarations =
 'V' _ variable_declaration (',' _? variable_declaration)* ';' _?

function_declarations =
 'F' _ function_declaration (',' _? function_declaration)* ';' _?

import_declaration = id _? ':' _? path '/' id

typedef_declaration = idtype
variable_declaration = idtype
function_declaration = idtype ('->' _? type)? instructions_block

idtype = id _? ':' _? type _?
path = id ( '/' id )*
id = [a-zA-Z_][a-zA-Z0-9_]*
type = (
  base_type
/ id
/ tuple_type
) _? ('*' _? base10 _?)*


base_type =
  'bool'
/ 'u8'
/ 'i8'
/ 'u16'
/ 'i16'
/ 'u32'
/ 'i32'
/ 'u64'
/ 'i64'
/ 'f32'
/ 'f64'

tuple_type = '(' idtype (',' _? idtype)* ')'

_ = (
  [ \n\r\t]
/ '//' [^\n]* '\n'
/ '/*' (& '*/' .) '*/'
)+

instruction =
  variable_declarations
/ inst_assignments
/ inst_if
```

```
/ inst_while
/ inst_do_until

inst_assignments =
  'A' _ inst_assignment (',' _? inst_assignment)* with_clause* ';' _?

inst_assignment =
  lval ':' _? assign_value

assign_value = expression / function_call

with_clause = 'W' _ with_assignment (',' _? with_assignment)*

with_assignment = id _? ':' _? assign_value

inst_if =
  'if' _ expression _? 'then' _ instructions
  ('elseif' _ expression _? 'then' _ instructions)*
  ('else' _ instructions)?
  'end:if' _

inst_while = 'while' _ expression _ 'do' _ instructions _ 'end:while' _

inst_do_until =
  'do' _ instructions 'until' _ expression 'end:do' _

lval = id _? (
    ('.' _? id _?)
  / ('[' _? expression ']' _?)
  )*

expression = a:inexpr { return shunting_yard(a) }

inexpr =
  fragment (operator _? fragment)*

fragment =
  lval
/ tuple
/ vector
/ literal


tuple = '(' _? (tuple_entry (',' _? tuple_entry)*)? ')' _?
tuple_entry = id _? ':' _? expression
vector = '(' _? expression (',' _? expression)+ ')' _?

literal = (
  number
) _?

function_call = id _? ':' _? expression

number =
  _float
/ _int

_int =
  base16
/ base2
/ base8
/ base10

base16 = '0x' [0-9a-fA-F]+
base2  = '0b' [01]+
```

```
base8  = '0' [0-7]*
base10 = [1-9][0-9]*

_float =
  base10 '.' base10? exponent?
/ '0'? '.' base10 exponent?

exponent = [eE][+-]?base10
operator =
  '<<'
/ '>>'
/ '&&'
/ '||'
/ '=='
/ '!='
/ '>='
/ '<='
/ '+'
/ '*'
/ '-'
/ '/'
/ '%'
/ '&'
/ '|'
/ '<'
/ '>'
/ '^'
```