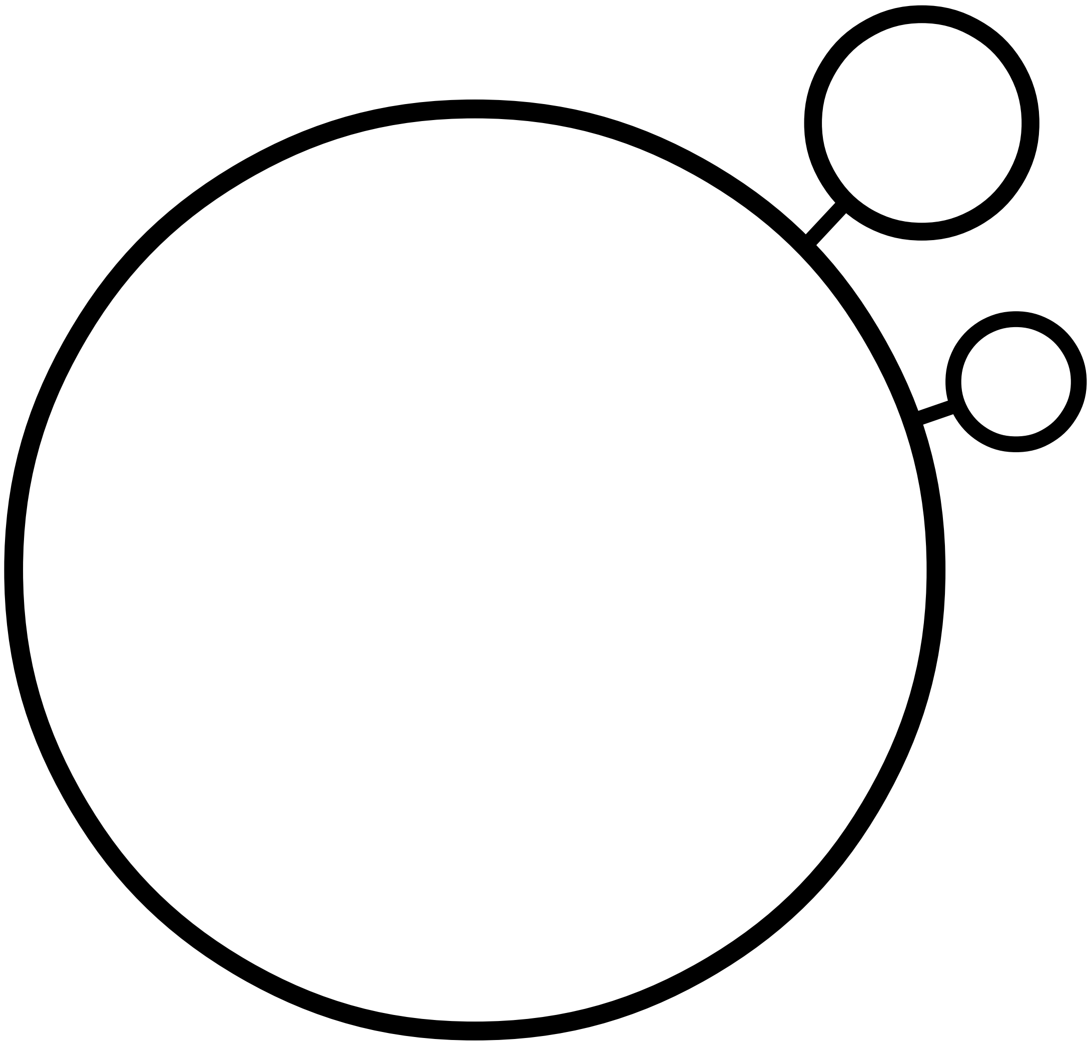
Ports

esoteric programming language documentation



See https://github.com/anima-libera/ports or https://esolangs.org/wiki/Ports
doc version 1.0 – lang version 1.0

Presentation, definitions, execution

Presentation

The Ports programming language is imperative, esoteric and Turing-complete. It was designed in 2020. If one finds a bug in a reference implementation, or if one thinks that this is an amazing esolang, one can send an email to anim.libera@gmail.com.
This programming language deals with the concept of spaces, ports and links, which are defined in the definition paragraph.

Definitions

A space is an object that has a code and a list of ports.
The root space is the initial space. It's code is the source code given to the interpreter. It is the only space that can have special ports.
A code is a sequence of instructions, some of these instructions are ports.
A port has a name. A port may be an instruction port, a space port or a special port. A port is said to be visible in the space where it lies.
A name is a string that may refer to a unique port in a space. In a given space, a name is said to refer to a port if the given space has a port that is named by this name, and a name is said to be available if the given space doesn't have a port that is named by this name.
An instruction port lies in a code as an instruction.
A space port is one of the two sides of an inter-space path that leads to an other space port in an other space. The other space port to which leads a space port is called its other side.
A special port is a port that leads to a mysterious code that lies outside of the root space and that can interact with the user's universe. A special port can only be visible in the root space.
Two ports of the same space can be linked together by a link. Ports can't be linked to more than one port.
A link chain is a sequence of one or more links that are chained by inter-space paths. Instruction ports and special ports are dead ends for link chains, however space ports are not since they lead to other space ports. For a given port being an edge of a link chain, the other edge of this link chain (which is also a port) is called the final linked port of the given port.
The spark is the fancy name of the instruction pointer. Saying that the spark increases its instruction pointer is a way of saying that the spark jumps to the instruction that follows the instruction it was on.

Execution

Initially, the root space is created, with the given source code as its code, and with the standard special ports as its ports in addition to the instruction ports of the code.
One of those special ports is the origin port, named o. It is initially linked to the first instruction port of the root space's code. The spark enters the root space through the origin port, follows the link up to the first instruction port of the code, and increment its instruction pointer to point to the instruction following the first instruction port of the code.
The spark executes the instruction it is on, then jumps to the next instruction of the code it is in, and this cycle repeats until it stops.... If the spark reaches the end of the code it is in, it jumps to the first instruction of this code. If the spark executes a port instruction and if this port is linked and if its final linked port is an instruction port or a special port, then the spark follows the link chain up to the final linked port, thus links allow for control flow.
Other instructions allow to create, move and cut links, as well as to create ports and spaces (however, ports and spaces, once created and named, cannot be renamed, moved nor destroyed).

Instructions

There are 7 instruction types: nop, cut-link, create-link, swap-link, create-port, create-space and port.

Nop

The nop instruction must be a dot character **.** and it does nothing. It may be optimized out and mainly serves as a nice statement separator in the source code.

Cut link

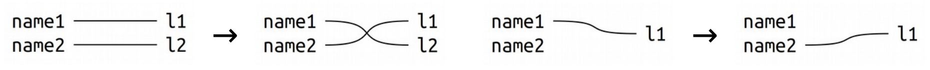
The cut-link instruction must match the format `name` where `name` is the name of a visible port. If this port named `name` is linked to another, then the link is cut. If the port named `name` isn't linked to another, then this instruction does nothing. If `name` isn't the name of a visible port, then the behavior of this instruction is undefined.

Create link

The create-link instruction must match the format `name1` **–** `name2` where `name1` is the name of a visible port and `name2` is the name of an other visible port. If the port named `name1` is linked to another port, the link is cut. If the port named `name2` is linked to another port, the link is also cut. Then, those two ports are linked together. If `name1` or `name2` isn't the name of a visible port, or if `name1` and `name2` are the same name, then the behavior of this instruction is undefined.

Swap link

The swap-link instruction must match the format `name1` **/** `name2` where `name1` is the name of a visible port and `name2` is the name of the same port or the name of an other visible port. Let `l1` be the port to which the port named `name1` is linked (if the port named `name1` is not linked to an other port, `l1` is nothing), and let `l2` be the same thing but for the port named `name2` instead of `name1`. Link the port named `name1` to `l2` and link the port named `name2` to `l1`. If the port named `name1` and the port named `name2` are the same port or are linked together, then this instruction does nothing. If the port named `name1` is non-linked and so is the port named `name2`, then this instruction does nothing as well. If `name1` and `name2` are the same name, then this instruction may be optimized out. If `name1` or `name2` isn't the name of a visible port, then the behavior of this instruction is undefined.



Create port

The create-port instruction must match the format `name1` **:** `name2` **|** `name3` where `name1` is the name of a visible space port, `name2` is an available name and `name3` is an available name in the space in which is the port that is the other side of the port named `name1`. A new port named `name2` is created in the space in which is the spark, and an other new port named `name3` is created in the space in which is the port that is the other side of the port named `name1`. Each one of those two new ports is the other side of the other. If `name1` is not the name of a visible space port, or if `name2` is not an available name, or if `name3` is not an available name in the space in which is the port that is the other side of the port named `name1`, then the behavior of this instruction is undefined.

Create space
The create-space instruction must match the format `name1` **:** `name2` **| {** `code` **}** where `name1` is an available name, `name2` is a name and `code` is a sequence of instructions ; or must match the format `name1` **:** `name2` **| [** `filepath` **]** where `name1` is an available name, `name2` is a name and `filepath` is the path of an existing file that is a Ports source file. Either the `code` instruction sequence is empty, or it meets the criteria of well formed code (defined in an other paragraph) and doesn't contains a port instruction named `name2`. If the format used is the second one, then the instruction behave exactly like if the format used was the first one with `code` being the content of the file `filepath` as it was before the spark even enters the root space through the origin port at the beginning of execution. This allows to spread the source code along multiple files and to use libraries. A new space is created. If the `code` instruction sequence contains at least one instruction, then the new space's code is `code`. If the `code` instruction sequence is empty, then the new space's code is (a copy of) the code in which lies this instruction (only the instructions are copied, the links of the port instructions are not). The `code` instruction sequence is not considered empty if it contains at least one instruction, even if all its instructions are optimized out like may be nop instructions. A new port named `name1` is created in the space is which is the spark, and an other new port named `name2` is created in the new space. Each one of those two new ports is the other side of the other. The new port named `name2` is linked to the first instruction port of the new space's code. If `name1` is not an available name, or if the `code` instruction sequence is not empty and doesn't meet the criteria of well formed code, or if the `code` instruction sequence contains a port instruction named `name2`, or if `filepath` isn't the path of an existing file that is a Ports source file, then the behavior of this instruction is undefined.

Port
The port instruction must match the format `name` ***** where `name` is a name. This instruction is a visible instruction port named by the name `name`. As any port, it can be linked to an other port of the space in which it is in, in such a case this instruction port is an edge of a link chain which leads to its final linked port (the other edge of the link chain). Let `flp` be the final linked port of this instruction (if any). If this instruction port is not linked, or if the `flp` is a space port (necessarily non-linked in such a case), then this instruction does nothing. If the `flp` is an instruction port, the spark jumps to it (and increases its instruction pointer after the execution of this instruction as usual). If the `flp` is a special port, then the spark goes through it and lands on the mysterious code it leads to (the mysterious code is then executed).

Syntax and criteria of well formed code

Syntax

The source code is a string of Unicode characters (but the use of non-ASCII characters is limited to comments). A line comment starts with `#` and ends with a newline character or the end of the source code. A block comment starts and ends with `###` and can include newline characters. Comments are treated like whitespace. Any whitespace character is allowed. Whitespace is ignored but may be used to separate names. A name is a string that uses only the latin lowercase letters `abcdefghijklmnopqrstuvwxyz` and the base 10 digits `0123456789`, for example `y0`, `o` and `42` are names. Note that the underscore character and upper case letters are not allowed in names. The source code should be a sequence of instructions, each of them must match a format of an instruction (described in an other paragraph). Any unexpected character is illegal and triggers an undefined behavior.

Criteria of well formed code

A well formed code doesn't triggers an undefined behavior (because relying on undefined behavior is bad practice and not portable). A well formed code contains at least one instruction port (because every space, once created, has an initial space port that is linked to the first instruction port of its code). A well formed code doesn't contain two or more instruction ports with the same name (because, in each space, each name should name one and only one port). A well formed code doesn't contains any instruction that are sure to trigger an undefined behavior when executed, even if those instructions cannot be reached by the spark at run time (for example, a code containing the instruction `o-o` is not well formed). A well formed code doesn't contain any create-space instruction that contains a non-empty code that doesn't meet the criteria of well formed code (for example, a code containing the instruction `i|o{o-o}` is not well formed). The source code is not well formed if it contains an instruction port named by the name of a special port. Note that, for example, the source code `m*x` is not well formed because it contains an instruction that is sure to trigger undefined behavior (the cut-link instruction `x` expect the name `x` to be the name of a visible port, but it isn't the name of a special port and there isn't any instruction that can create a port named `x`), but a code containing the instruction `i|o{m*x}` may be well formed if there is an instruction somewhere else that can create a port named `x` where it should so that undefined behavior can no longer be triggered (the instruction `i|o{m*x}` isn't sure to trigger undefined behavior (for the well formed code criteria), even if the spark is sure to execute it before any port named `x` is created where it should ; this criteria is a parse-time criteria, not a run-time criteria).

File encoding

The source files are expected to use UTF-8 or ASCII encoding.

Standard special ports

The shared bit buffer

The mysterious codes may interact with the shared bit buffer, which is a bit queue.

The shared mode

The mysterious codes that interact with the shared bit buffer also interact with the shared mode. Before the shared bit buffer is read or modified, the shared mode is set to a value related to the operation that will be performed on the shared bit buffer. If the shared mode was not already equal to the value it is set to, then the shared bit buffer is cleared before it is read or modified. This design ensures that the program doesn't use the shared bit buffer as a bit queue to store information (its only purpose it to hold information during its transfer from the program to the outer world, or from the outer world to the program).

o

The name `o` is the name of a special port that is also called the origin port. At the beginning of execution, this port is linked to the first instruction port of the root space's code, and the spark enters the root space through this port. If the spark goes through this port, then it doesn't come back and the execution ends. It is the normal way of ending the execution. Note that this port must be implemented, and it is the only special port that must be implemented.

o0

The name `o0` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`OUT`**, then a `0` is appended to the shared bit buffer, and then the spark goes back through this port. Note that this port may not be implemented.

o1

The name `o1` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`OUT`**, then a `1` is appended to the shared bit buffer, and then the spark goes back through this port. Note that this port may not be implemented.

of

The name `of` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`OUT`**, then the shared bit buffer is printed out as a UTF-8 encoded string, then the shared bit buffer is cleared, and then the spark goes back through this port. Note that this port may not be implemented.

os

The name `os` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`OUT`**, then the shared bit buffer is sent to the system as a UTF-8 encoded system command, then the shared mode is set to **`IN`** while the shared bit buffer is cleared, then the exit code of the command execution is appended to the shared bit buffer as a 32 bits unsigned integer, then the intercepted *stdout* output of the command execution is appended to the shared bit buffer as a UTF-8 encoded string (not necessarily ended by a null character), then if the intercepted *stderr* output of the command execution is not empty then a null byte is appended to the shared bit buffer and then the intercepted *stderr* output of the command execution is appended to the shared bit buffer as a UTF-8 encoded string (not necessarily ended by a null character), and then the spark goes back through this port. Note that even if the command execution doesn't return a success exit code, the interpreter doesn't care and doesn't raise an error for that. Note that this port may not be implemented, or may be implemented but may not allow the execution of any command unless the interpreter was given the explicit permission to do so from the user.

ia
The name `ia` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`IN`**, then the used is asked to provide a string which is appended to the shared bit buffer as a UTF-8 encoded string (not necessarily ended by a null character), and then the spark goes back through this port. Note that this port may not be implemented.

ir
The name `ir` is the name of a special port. If the spark goes through this port, then the shared mode is set to **`IN`**, then if the shared bit buffer is empty then the spark goes back through this port, if the shared bit buffer is not empty and the front bit is a `0` then this front bit is popped and the spark goes back through the special port named `o0`, if the shared bit buffer is not empty and the front bit is a `1` then this front bit is popped and the spark goes back through the special port named `o1`. Note that this port may not be implemented.

Miscellaneous

Good practice principles

→ As Ports is an esoteric language, any source code produced with non-educative purposes in mind should look esoteric too. This implies, but is not limited to, the use of non-well-chosen port names with very few characters.
→ If a space looks like it shouldn't be a space, it should be replaced by a nop instruction. Using the nop instruction may also help the source code to look like natural English language (like COBOL, it really increase readability).
→ Indent using anything but spaces (like tabs, nops, block comments, etc) (this is objective and not a personal opinion) (really).

Turing-completeness

Any Minsky machine program can be translated into a Ports program that does exactly the same operations, and Minsky machines with at least two registers are Turing-complete, thus Ports is Turing-complete. The swap-link instruction is not used in the proof, the nop instruction could be deleted from the proof, and every cut-link instruction `x` could be replaced by a create-link instruction `x-y` where y is a space port never used elsewhere and that has an other side that is never used (so never linked). Thus the Ports subset made of the create-link, create-port, create-space and port instructions is Turing-complete.
The proof is the `mm2ports.py` script in the GitHub repository.

Reference implementation usage

The command **`python3 ports.py file/path.ports`** should execute the source code that is in the file "file/path.ports".
The command **`python3 ports.py -h`** should display a help message containing the available options appendable to the first command.

Expected behavior of Ports implementations

A Port implementation should behave like this specification said it should, and should raise an error as an undefined behavior except where it makes sense not to do so.

Recommended steps to get started

1. Make sure the minimal Ports program `m*` doesn't raise an error and does nothing as expected.
2. Make a program that prints out a character.
3. Create a tool that automates the production of Ports code that prints out a given string.
4. Make a cat program that works for one ascii character strings.
5. Get a pen and some paper if not already done.
6. Make a cat program that works for strings of any length, or analyze the one given as an example.
7. Make a unsigned integer support library, with addition and subtraction.
8. Do what you want, you are ready now.

Example cat program (obfuscated)

```
h*t|o{h*o:p|mm.o:r|i.o:y|x.o:u|z.o:gd|l.o:2|d.p-a.r-3.y-dz.4-1.s1-v.v*1h*y-x.u-
gs.gs*8*u-s.s*g*y-x.gd-kx.kx*00*gd-m.m*dz*2-t4.t4*j*2-9.9*0*tm*4*0-pl.pl*1*t|o{
}t-v4.v4*z-8.l-00.d-j.4.0-b.b*a*tm-c.c*y-1h.p-mm.r-i.dz-k.k*3*tm-7.7*y-g.p-mm.r
-i.dz-6.6*s1*}ia-v.v*t-gs.gs*o0-a.o1-3.58*d-58.ir-s.s*z-bv.l-oy.d-s1.15-kx.kx*a
*mm-m.m*3*i-t4.t4*15*x-9.9*bv*o0-pl.pl*15-v4.v4*oy*o1-b.b*15-c.c*s1*o0-k.k*o0-7
.7*o0-6.6*o0-n8.n8*o1-ee.ee*o0-sz.sz*o1-1c.1c*o0-hz.hz*of-ft.ft*
```