Ableton Programming Tasks

In this document, you will find three programming tasks to choose from. All three tasks are somewhat related to our product Ableton Live, but we have tried to make the descriptions self-contained to ensure that no in-depth knowledge of Ableton Live is required for their proper completion.

Your submission should meet the following requirements:

- The resulting command line application should read input as described in the language specification from standard input. No extra input (e.g. command line options) should be necessary to successfully run the application.
- All program output should be written to standard output and adhere to the specification. Your program should not output any additional information or metadata.
- Your submission should be written in ISO C++, Go, Python, or any other language of your choice, and should have unit tests. Use a language-appropriate testing framework for your tests. Our recommended language for the solution is C++, because the open positions are C++ focused.
- Please include a README file containing instructions for building and running your program and tests.

You can assume that your program will only be run with valid inputs. There is no need to handle errors which might arise from malformed input.

Please choose one (and only one) of the following three tasks to complete.

Please craft your solution with the same standards you would apply when writing production code. We don’t mean that you should invent requirements, or over-engineer a solution in anticipation of running in a “production environment”. What we mean is: show us what “quality code” means to you, in the sense of readability, conciseness and completeness within the context of the task as described.

Task 1: Modular Madness

In audio software, sound processing is done using a network of modules, each executing a simple task like applying a filter or summing signals. In this task, we process strings instead of audio using a similar module-based approach.

The input defines a network of modules and a stream of input into this network. This input stream should be fed into the first module (in the order of definition) and the output of the program should correspond to the output of the last module. If there are multiple input connections for a module, they should be "summed" before feeding to the module. Summing works by appending the strings in the order in which the input connections have been made. The individual modules should be processed in the order they have been defined.[1] The network should only process one string at a time. Each `process` statement should let the network run empty (see also the example below).

If you encounter ambiguities while solving this task, make an assumption about the behavior of the network and document it in an appropriate way.

Input description

Each input line consists of a keyword followed by arguments. There are three kinds of input lines:

1. *Module definition*, adds a new module to the network

```
module <name> <operation>
```

2. *Connection command*, connects the output of one module to the input of another

```
connect <name1> <name2>
```

3. *Process command*, feeds input into the first module in the network

```
process <...list of strings to be processed>
```

The *module definition* defines a new module to be added to the network. The `name` argument of the `module` command is an arbitrary name (no whitespace) for the module and `operation` specifies what the module should do with its input. The following operations should be supported:

- `echo` : The output string is the input string concatenated to itself.
- `reverse` : The output string is the input string reversed.
- `delay`: The output string is the previous input string. The initial output is "hello".
- `noop`: The input appears unchanged at the output.

[1] In a "real" audio application, modules have to be processed in the order of their dependencies. Such an application also has to deal with cyclic module dependencies by inserting unit delays in appropriate places to get an acyclic graph to process. While we also accept solutions using this alternate approach (provided they handle all input correctly), please note that this makes this task *considerably* harder.

The *connect command* connects the output of module `name1` to the input of module `name2`. Modules will always be defined before they are connected.

The *process command* feeds the rest of the line into the first module. This input consists of one or more strings made of the characters a-z. Individual strings are separated with a single space character. If there are several `process` lines, each of them should behave the same way as if input alone.

Output description

For each `process` line in the input, a line should be printed that contains the outputs of the last defined module separated by a single space character. The number of output strings must be limited to sixteen times the number of input strings in the corresponding `process` line.

Example
Input

```
module alpha reverse
module beta delay
connect alpha beta
process hello world
```

Output

```
hello olleh dlrow
```

Task 2: Follow me

A *Clip* in Live contains musical content, ranging from a short sound to an entire song. Some clips play once, while others loop forever. Clips start playing when they are triggered. Clips can also have *Follow Actions*, which allow creating groups of clips that can trigger each other. A clip can have one or two Follow Actions that define what happens after a clip has been playing for a set amount of time.

Every Follow Action has an associated Chance value that controls the likelihood of each of the two Follow Actions occurring. For example, if `action1` has Chance set to 1, and `action2` has Chance set to 0, `action1` will be chosen every time. If, on the other hand, `action2` has Chance set to 10 in this scenario, `action1` will be chosen ten times less often than `action2` on average.

We want you to write a program that simulates this behavior in a simplified way.

Input description

Each input line consists of a keyword followed by arguments. There are two kinds of input lines:

1. *Clip definition*, creates a named clip with a set of properties

```
clip <name> <ticks to play> <follow chance1> <follow chance2> <action1> <action2>
```

2. *Ticks command*, runs the program a certain number of ticks

```
ticks <ticks to play> <...optional list of floating point numbers>
```

The *clip definition* defines a clip and its Follow Actions. `ticks to play` is the number of steps that the clip should "play" before it triggers another clip as defined by the Follow Actions. There are two Follow Actions associated with each clip, and they are chosen randomly so that their relative occurrence corresponds to the specified `follow chances`.

Possible values for `action1` and `action2` are:

- `none` : Nothing happens – the clip keeps playing.
- `any` : Any clip (including the playing one) can be triggered.
- `other` : Any *other* clip can be triggered.
- `next` : The next clip (in the order they were created) will be triggered.
- `previous` : The previous clip will be triggered.

The list of clips should "wrap around". In other words, if the `next` action is executed on the last clip in the list, the first clip in the list should begin playing. Similarly, executing the `previous` action on the first clip should begin playback of the last clip.

If a clip is created with the same name as an existing clip, the new clip should replace the existing clip.

The *ticks command* indicates the number of times a clip is played. The `ticks` line may *optionally* contain floating point numbers after the number of ticks. The number of floating point numbers is two times the number of ticks. (The reason why two are supplied per tick is up to you to figure out.) For example, both of the following `ticks` commands are valid:

```
ticks 3
ticks 3 0.7 0.5 1.0 0.8 0.0 1.0
```

These pairs of numbers are optionally supplied to ensure that the output can be made deterministic. If these pairs of numbers are absent in the `ticks` input, random numbers should be used instead.

With every `ticks` command, the playback should advance the number of steps given. The very first `ticks` command will always trigger the first clip.

Output description

For each tick, a single output line is created containing the name of the currently playing clip.

Example

Input

```
clip hello 2 1.0 0.0 next none
clip world 3 1.0 0.0 previous none
ticks 10
```

Output

```
hello
hello
world
world
world
hello
hello
world
world
world
```

Task 3: It’s All Warped

In Live, audio samples can be stretched and squeezed to change their timing. We refer to this stretching and squeezing as *Warping*, and pins called *Warp Markers* lock a specific point in the sample (in sample time) to a specific place in a measure (in beat time). You can use any number of Warp Markers to create an arbitrary mapping of the sample’s inherent rhythm to a musical meter.

We want you to write a program which, given a set of Warp Markers, can map time values between the two value spaces: beat time (clock time) and sample time. Each Warp Marker defines a mapping point between the two spaces. Between two Warp Markers, there is a linear interpolation of the values, resulting in a constant speed of audio playback (referred to as tempo from here on). When a sample is squeezed, the tempo increases; when a sample is stretched, the tempo decreases. For the sake of this task, please assume the following behavior:

- An audio sample always has at least one Warp Marker.
- Between two Warp Markers, the tempo is constant.
- The tempo before the first Warp Marker is the same as the tempo after the first Warp Marker.
- The tempo after the last Warp Marker is specified separately in the input.
- Beat time is measured in beats, sample time is measured in seconds, and tempo is measured in beats per second.

Input description

Each input line consists of a keyword followed by numeric arguments. All numeric values are entered as floating point numbers without units. Warp Marker and tempo definitions affect only the conversions that come later in the input.

There are four kinds of input lines:

1. Warp Marker definition

```
marker <beat time> <sample time>
```

2. Definition of the tempo after the last marker

```
end_tempo <value>
```

3. Sample time to beat time conversion

```
s2b <sample time>
```

4. Beat time to sample time conversion

```
b2s <beat time>
```

At least one Warp Marker and the tempo after the last Warp Marker will be defined before the first conversion; otherwise, these can appear in any order.

Output description

For each of the `s2b` and `b2s` lines, the corresponding output time is printed without unit.

Example

Input

```
marker 0.0 0.0
marker 1.0 5.0
end_tempo 10.0
b2s 0.5
s2b 6.0
```

Output

```
2.5
11.0
```

The diagram below illustrates the resulting mapping