This copy of the Lingua Franca handbook for the py target was created on Wednesday, June 22, 2022 against commit [839a79](#).

Table of Contents

A First Reactor

See the [requirements](#) for using this target.

Minimal Example

A minimal but complete Lingua Franca file with one reactor is this:

```
target Python;
main reactor {
    reaction(startup) {=
        print("Hello World.")
    =}
}
```

Every Lingua Franca program begins with a [target declaration](#) that specifies the language in which reactions are written. This is also the language of the program(s) generated by the Lingua Franca code generator.

Every LF program also has a **main** reactor, which is the top level of a hierarchy of contained and interconnected reactors. The above simple example has no contained reactors.

The **main** reactor above has a single **reaction**, which is triggered by the **startup** trigger. This trigger causes the reaction to execute at the start of the program. The body of the reaction, delimited by `{= ... =}`, is ordinary Python code which, as we will see, has access to a number of functions and variables specific to Lingua Franca.

Examples

Examples of Lingua Franca programs can be found in [the examples-lingua-franca repository](#).

The [regression tests](#) have a rich set of examples that illustrate every feature of the language.

Structure of an LF Project

The Lingua Franca tools assume that LF programs are put into a file with a `.lf` extension that is stored somewhere within a directory called `src`. To compile and run the above example, choose a **project root** directory, create a `src` directory within that, and put the above code into a file

called, say, `src/HelloWorld.lf`. You can compile the code on the [command line](#), within [Visual Studio Code](#), or within the [Epoch IDE](#). On the command line this will look like this:

```
> lfc src/HelloWorld.lf
... output from the code generator and compiler ...
```

Reactor Block

A **reactor** is a software component that reacts to input events, timer events, and internal events. It has private state variables that are not visible to any other reactor. Its reactions can consist of altering its own state, sending messages to other reactors, or affecting the environment through some kind of actuation or side effect (e.g., printing a message, as in the above `HelloWorld` example).

The general structure of a reactor definition is as follows:

```
[main or federated] reactor <class-name> [(<parameters>)] {
    input <name>
    output <name>
    state <name>(<value>)
    timer <name>([<offset>, [<period>]])
    logical action <name>
    physical action <name>
    [const] method <name>(<parameters>) {= ... body ...=}
    reaction(<triggers>) [<uses>] [=> <effects>] {= ... body ...=}
    <instance-name> = new <class-name>([<parameter-assignments>])
    <instance-name> [, ...] => <instance-name> [, ...] [after <delay>]
}
```

Contents within square brackets are optional, contents within `<...>` are user-defined, and each line may appear zero or more times, as explained in the next pages. Parameters, inputs, outputs, timers, actions, and contained reactors all have names, and the names are required to be distinct from one another.

If the **reactor** keyword is preceded by **main**, then this reactor will be instantiated and run by the generated code.

Any number of reactors may be defined in one file, and a **main** reactor need not be given a name, but if it is given a name, then that name must match the file name.

Reactors may extend other reactors, inheriting their properties, and a file may import reactors from other files. If an imported LF file contains a **main** reactor, that reactor is ignored (it will not

be imported). This makes it easy to create a library of reusable reactors that each come with a test case or demonstration in the form of a main reactor.

Comments

Lingua Franca files can have C/C++/Java-style comments and/or Python-style comments. All of the following are valid comments:

```
// Single-line C-style comment.
/*
 * Multi-line C-style comment.
 */
# Single-line Python-style comment.
'''
   Multi-line Python-style comment.
'''
```

Inputs and Outputs

In this section, we will endow reactors with inputs and outputs.

Input and Output Declarations

Input and output declarations have the form:

```
input <name>
output <name>
```

For example, the following reactor doubles its input and sends the result to the output:

```
target Python;
reactor Double {
    input x;
    output y;
    reaction(x) -> y {=
        y.set(x.value * 2)
    =}
}
```

Notice how the input value is accessed and how the output value is set. This is done differently for each target language. See the [Target Language Details](#) for detailed documentation of these mechanisms. Setting an output within a reaction will trigger downstream reactions at the same [Logical Time](#) that the reaction is invoked (or, more precisely, at the same [tag](#)). If a particular output port is set more than once at any tag, the last set value will be the one that downstream reactions see. Since the order in which reactions of a reactor are invoked at a logical time is deterministic, and whether inputs are present depends only on their timestamps, the final value set for an output will also be deterministic.

The **reaction** declaration above indicates that an input event on port `x` is a **trigger** and that an output event on port `y` is a (potential) **effect**. A reaction can declare more than one trigger or effect by just listing them separated by commas. For example, the following reactor has two triggers and tests each input for presence before using it:

```
target Python;
reactor Destination {
    input x;
    input y;
    reaction(x, y) {=
        sum = 0
        if x.is_present:
            sum += x.value
        if y.is_present:
            sum += y.value
        print(f"Received {sum}")
    =}
}
```

NOTE: if a reaction fails to test for the presence of an input and reads its value anyway, then the result it will get is target dependent. In the Python target, the value will be None if the input is not present.

Triggers, Effects, and Uses

The general form of a **reaction** is

```
reaction (<triggers>) <uses> -> <effects> {=
    <target language code>
=}
```

The **triggers** field can be a comma-separated list of input ports, output ports of contained reactors, timers, actions, or the special events **startup** and **shutdown**. There must be at least one trigger for each reaction. A reaction with a **startup** trigger is invoked when the program begins executing, and a reaction with a **shutdown** trigger is invoked at the end of execution.

The **uses** field, which is optional, specifies input ports (or output ports of contained reactors) that do not trigger execution of the reaction but may be read by the reaction.

The **effects** field, which is also optional, is a comma-separated lists of output ports ports, input ports of contained reactors, or actions.

Setting an Output Multiple Times

If one or more reactions set an output multiple times at the same tag, then only the last value set will be seen by any downstream reactors.

If a reaction wishes to test whether an output has been previously set at the current tag by some other reaction, it can test it in the same way it tests inputs for presence.

Mutable Inputs

Normally, a reaction does not modify the value of an input. An input is said to be **immutable**. The degree to which this is enforced varies by target language. Most of the target languages make it rather difficult to enforce, so the programmer needs to avoid modifying the input. Modifying an input value may lead to nondeterministic results.

Occasionally, it is useful to modify an input. For example, the input may be a large data structure, and a reaction may wish to make a small modification and forward the result to an output. To accomplish this, the programmer should declare the input **mutable** as follows:

mutable input <name>;

This is a directive to the code generator indicating that reactions that read this input may also modify the value of the input. The code generator will attempt to optimize the scheduling to avoid copying the input value, but this may not be possible, in which case it will automatically insert a copy operation, making it safe to modify the input. The target-specific reference documentation has more details about how this works.

Parameters and State Variables

Parameter Declaration

A reactor class definition can parameterized as follows:

```
reactor <class-name>(<param-name>(<expr>), ... ) {
    ...
}
```

Each parameter must have a *default value*, written `(<expr>)`. An expression may be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, a list of values, or target-language code enclosed in `{= ... =}`, for example. See [Expressions](#) for full details on what expressions are valid.

For example, the `Double` reactor on the [previous page](#) can be replaced with a more general parameterized reactor `Scale` as follows:

```
target Python;
reactor Scale(factor(2)) {
    input x;
    output y;
    reaction(x) -> y {=
        y.set(x.value * self.factor)
    =}
}
```

This reactor, given any input event `x` will produce an output `y` with value equal to the input scaled by the `factor` parameter. The default value of the `factor` parameter is 2, but this can be changed when the `Scale` reactor is instantiated.

Notice how, within the body of a reaction, the code accesses the parameter value. This is different for each target language.

State Declaration

A reactor declares a state variable as follows:

```
state <name>(<value>);
```

The `<value>` is an initial value and, like parameter values, can be given as an [expression](#) or target language code with delimiters `{= ... =}`. The initial value can also be given as a parameter name. The value can be accessed and modified in a target-language-dependent way as illustrated by the following example:

```
target Python;
reactor Count {
    state count(0);
    output y;
    timer t(0, 100 msec);
    reaction(t) -> y {=
        y.set(self.count)
        self.count += 1
    =}
}
```

This reactor has an integer state variable named `count`, and each time its reaction is invoked, it outputs the value of that state variable and increments it. The reaction is triggered by a **timer**, discussed in the next section.

Time and Timers

Logical Time

A key property of Lingua Franca is **logical time**. All events occur at an instant in logical time. By default, the runtime system does its best to align logical time with **physical time**, which is some measurement of time on the execution platform. The **lag** is defined to be physical time minus logical time, and the goal of the runtime system is maintain a small non-negative lag.

The **lag** is allowed to go negative only if the [`fast` target property](#) or the [--fast](#) is set to `true`. In that case, the program will execute as fast as possible with no regard to physical time.

Time Values

A time value is given with units (unless the value is 0, in which case the units can be omitted). The allowable units are:

- For nanoseconds: `ns`, `nsec`, or `nsecs`
- For microseconds: `us`, `usec`, or `usecs`
- For milliseconds: `ms`, `msec`, or `msecs`
- For seconds: `s`, `sec`, `secs`, `second`, or `seconds`
- For minutes: `min`, `minute`, `mins`, or `minutes`
- For hours: `h`, `hour`, or `hours`
- For days: `d`, `day`, or `days`
- For weeks: `week` or `weeks`

The following example illustrates using time values for parameters and state variables:

```
target Python;
main reactor SlowingClock(start(100 msec), incr(100 msec)) {
    state interval(start);
    logical action a;
    reaction(startup) -> a {=
        a.schedule(self.start)
    =}
    reaction(a) -> a {=
        elapsed_logical_time = lf.time.logical_elapsed()
        print(
            f"Logical time since start: {elapsed_logical_time} nsec."
        )
        self.interval += self.incr
        a.schedule(self.interval)
    =}
}
```

This has two time parameters, `start` and `incr`, each with default value `100 msec`. This parameter is used to initialize the `interval` state variable, which also stores a time. The **logical action** `a`, explained [below](#), is used to schedule events to occur at time `start` after program startup and then at intervals that are increased each time by `incr`. The result of executing this program will look like this:

```
Logical time since start: 100000000 nsec.
Logical time since start: 300000000 nsec.
Logical time since start: 600000000 nsec.
Logical time since start: 1000000000 nsec.
...
```

Timers

The simplest use of logical time in Lingua Franca is to invoke a reaction periodically. This is done by first declaring a **timer** using this syntax:

```
timer <name>(<offset>, <period>);
```

The `<period>`, which is optional, specifies the time interval between timer events. The `<offset>`, which is also optional, specifies the (logical) time interval between when the program starts executing and the first timer event. If no period is given, then the timer event occurs only once. If neither an offset nor a period is specified, then one timer event occurs at program start, simultaneous with the **startup** event.

The period and offset are given by a number and a units, for example, `10 msec`. See the [expressions documentation](#) for allowable units. Consider the following example:

```
target Python;
main reactor Timer {
    timer t(0, 1 sec);
    reaction(t) {=
        print(f"Logical time is {lf.time.logical()}.")
    =}
}
```

This specifies a timer named `t` that will first trigger at the start of execution and then repeatedly trigger at intervals of one second. Notice that the time units can be left off if the value is zero.

Each target provides a built-in function for retrieving the logical time at which the reaction is invoked, FIXME . On most platforms (with the exception of some embedded platforms), the returned value is a 64-bit number representing the number of nanoseconds that have elapsed since January 1, 1970. Executing the above displays something like the following:

```
Logical time is 1648402121312985000.
Logical time is 1648402122312985000.
Logical time is 1648402123312985000.
...
```

The output lines appear at one second intervals unless the `fast` option has been specified.

Elapsed Time

The times above are a bit hard to read, so, for convenience, each target provides a built-in function to retrieve the *elapsed* time. For example:

```
target Python;
main reactor TimeElapsed {
    timer t(0, 1 sec);
    reaction(t) {=
        print(
            f"Elapsed logical time is {lf.time.logical_elapsed()}."
        )
    =}
}
```

See the [Target Language Details](#) for the full set of functions provided for accessing time values.

Executing this program will produce something like this:

```
Elapsed logical time is 0.
Elapsed logical time is 1000000000.
Elapsed logical time is 2000000000.
...
```

Comparing Logical and Physical Times

The following program compares logical and physical times:

```
target Python;
main reactor TimeLag {
    timer t(0, 1 sec);
    reaction(t) {=
        t = lf.time.logical_elapsed()
        T = lf.time.physical_elapsed()
        print(
            f"Elapsed logical time: {t}, physical time: {T}, lag: {T-t}"
        )
    =}
}
```

Execution will show something like this:

```
Elapsed logical time: 0, physical time: 855000, lag: 855000
Elapsed logical time: 1000000000, physical time: 1004714000, lag: 4714000
Elapsed logical time: 2000000000, physical time: 2004663000, lag: 4663000
Elapsed logical time: 3000000000, physical time: 3000210000, lag: 210000
...
```

In this case, the lag varies from a few hundred microseconds to a small number of milliseconds. The amount of lag will depend on the execution platform.

Simultaneity and Instantaneity

If two timers have the same *offset* and *period*, then their events are logically simultaneous. No observer will be able to see that one timer has triggered and the other has not.

A reaction is always invoked at a well-defined logical time, and logical time does not advance during its execution. Any output produced by the reaction will be **logically simultaneous** with the input. In other words, reactions are **logically instantaneous** (for an exception, see Logical Execution Time). Physical time, however, does elapse during execution of a reaction.

Timeout

By default, a Lingua Franca program will terminate when there are no more events to process. If there is a timer with a non-zero period, then there will always be more events to process, so the default execution will be unbounded. To specify a finite execution horizon, you can either specify a `timeout target property` or a `[--timeout command-line option](ocs/handbook/target-declaration#command-line-arguments). For example, the following` timeout` property will cause the above timer with a period of one second to terminate after 11 events:

```
target Python {
    timeout: 10 sec
}
```

Startup and Shutdown

To cause a reaction to be invoked at the start of execution, a special **startup** trigger is provided:

```
reactor Foo {
    reaction(startup) {=
        ... perform initialization ...
    =}
}
```

The **startup** trigger is equivalent to a timer with no *offset* or *period*.

To cause a reaction to be invoked at the end of execution, a special **shutdown** trigger is provided. Consider the following reactor, commonly used to build regression tests:

```
target Python;
reactor TestCount(start(0), stride(1), num_inputs(1)) {
    state count(start);
    state inputs_received(0);
    input x;
    reaction(x) {=
        print(f"Received {x.value}.")
        if x.value != self.count:
            sys.stderr.write(f"ERROR: Expected {self.count}.\n")
            exit(1)
        self.count += self.stride
        self.inputs_received += 1
    =}
    reaction(shutdown) {=
        print("Shutdown invoked.")
        if self.inputs_received != self.num_inputs:
            sys.stderr.write(
                f"ERROR: Expected to receive {self.num_inputs} inputs, but
            )
            exit(2)
    =}
}
```

This reactor tests its inputs against expected values, which are expected to start with the value given by the `start` parameter and increase by `stride` with each successive input. It expects to receive a total of `num_inputs` input events. It checks the total number of inputs received in its **shutdown** reaction.

The **shutdown** trigger typically occurs at microstep 0, but may occur at a larger microstep. See Superdense Time and Termination.

Composing Reactors

Contained Reactors

Reactors can contain instances of other reactors defined in the same file or in an imported file. Assume the `Count` and `Scale` reactors defined in [Parameters and State Variables](#) are stored in files `Count.lf` and `Scale.lf`, respectively, and that the `TestCount` reactor from [Time and Timers](#) is stored in `TestCount.lf`. Then the following program composes one instance of each of the three:

```
target Python {
    timeout: 1 sec,
    fast: true
}
import Count from "Count.lf";
import Scale from "Scale.lf";
import TestCount from "TestCount.lf";

main reactor RegressionTest {
    c = new Count();
    s = new Scale(factor = 4);
    t = new TestCount(stride = 4, num_inputs = 11);
    c.y -> s.x;
    s.y -> t.x;
}
```

Diagrams

As soon as programs consist of more than one reactor, it becomes particularly useful to reference the diagrams that are automatically created and displayed by the Lingua Franca IDEs. The diagram for the above program is as follows:

In this diagram, the timer is represented by a clock-like icon, the reactions by chevron shapes, and the **shutdown** event by a diamond. If there were a **startup** event in this program, it would appear as a circle.

Creating Reactor Instances

An instance is created with the syntax:

```
<instance_name> = new <class_name>(<parameters>)
```

A bank with several instances can be created in one such statement, as explained in the [banks of reactors documentation](#).

The `<parameters>` argument is a comma-separated list of assignments:

```
<parameter_name> = <value>, ...
```

Like the default value for parameters, `<value>` can be a numeric constant, a string enclosed in quotation marks, a time value such as `10 msec`, target-language code enclosed in `{= ... =}`, or any of the list forms described in [Expressions](#).

Connections

Connections between ports are specified with the syntax:

```
<source_port_reference> -> <destination_port_reference>
```

where the port references are either `<instance_name>.<port_name>` or just `<port_name>`, where the latter form is used for connections that cross hierarchical boundaries, as illustrated in the next section.

On the left and right of a connection statement, you can put a comma-separated list. For example, the above pair of connections can be written,

```
c.y, s.y -> s.x, t.x
```

The only constraint is that the total number of channels on the left match the total number on the right.

A destination port (on the right) can only be connected to a single source port (on the left). However, a source port may be connected to multiple destinations, as in the following example:

```
reactor A {
    output y
}
reactor B {
    input x
}
main reactor {
    a = new A()
    b1 = new B()
    b2 = new B()
    a.y -> b1.x
    a.y -> b2.x
}
```

Lingua Franca diagram

Lingua Franca provides a convenient shortcut for such multicast connections, where the above two lines can be replaced by one as follows:

```
(a.y)+ -> b1.x, b2.x
```

The enclosing `( ... )+` means to repeat the enclosed comma-separated list of sources however many times is needed to provide inputs to all the sinks on the right of the connection `->`.

Import Statement

An import statement has the form:

```
import <classname> as <alias> from "<path>"
```

where `<classname>` and `<alias>` can be a comma-separated list to import multiple reactors from the same file. The `<path>` specifies another `.lf` file relative to the location of the current file. The `as <alias>` portion is optional and specifies alternative class names to use in the **new** statements.

Hierarchy

Reactors can be composed in arbitrarily deep hierarchies. For example, the following program combines the `Count` and `Scale` reactors within on `Container`:

```
target Python;
import Count from "Count.lf";
import Scale from "Scale.lf";
import TestCount from "TestCount.lf";

reactor Container(stride(2)) {
    output y;
    c = new Count();
    s = new Scale(factor = stride);
    c.y -> s.x;
    s.y -> y;
}

main reactor Hierarchy {
    c = new Container(stride = 4);
    t = new TestCount(stride = 4, num_inputs = 11);
    c.y -> t.x;
}
```


Lingua Franca diagram

The `Container` has a parameter named `stride`, whose value is passed to the `factor` parameter of the `Scale` reactor. The line

```
s.y -> y;
```

establishes a connection across levels of the hierarchy. This propagates the output of a contained reactor to the output of the container. A similar notation may be used to propagate the input of a container to the input of a contained reactor,

```
x -> s.x;
```

Connections with Logical Delays

Connections may include a **logical delay** using the **after** keyword, as follows:

```
<source_port_reference> -> <destination_port_reference> after <time_val
```

where `<time_value>` can be any of the forms described in Expressions.

The **after** keyword specifies that the logical time of the event delivered to the destination port will be larger than the logical time of the reaction that wrote to source port. The time value is required to be non-negative, but it can be zero, in which case the input event at the receiving end will be one microstep later than the event that triggered it.

Physical Connections

A subtle and rarely used variant of the `->` connection is a **physical connection**, denoted `~>`. For example:

```
main reactor {
    a = new A();
    b = new B();
    a.y ~> b.x;
}
```

This is rendered in by the diagram synthesizer as follows:

Lingua Franca diagram

In such a connection, the logical time at the recipient is derived from the local physical clock rather than being equal to the logical time at the sender. The physical time will always exceed the logical time of the sender (unless fast is set to `true`), so this type of connection incurs a nondeterministic positive logical time delay. Physical connections are useful sometimes in [Distributed-Execution](#) in situations where the nondeterministic logical delay is tolerable. Such connections are more efficient because timestamps need not be transmitted and messages do not need to flow through through a centralized coordinator (if a centralized coordinator is being used).

Reactions and Methods

Reaction Order

A reactor may have multiple reactions, and more than one reaction may be enabled at any given tag. In Lingua Franca semantics, if two or more reactions of the same reactor are **simultaneously enabled**, then they will be invoked sequentially in the order in which they are declared. More strongly, the reactions of a reactor are **mutually exclusive** and are invoked in tag order primarily and declaration order secondarily. Consider the following example:

```
target Python {
    timeout: 3 secs
}
main reactor Alignment {
    state s(0);
    timer t1(100 msec, 100 msec);
    timer t2(200 msec, 200 msec);
    timer t4(400 msec, 400 msec);
    reaction(t1) {=
        self.s += 1
    =}
    reaction(t2) {=
        self.s -= 2
    =}
    reaction(t4) {=
        print(f"s = {self.s}")
    =}
}
```

Every 100 ms, this increments the state variable `s` by 1, every 200 ms, it decrements `s` by 2, and every 400 ms, it prints the value of `s`. When these reactions align, they are invoked in declaration order, and, as a result, the printed value of `s` is always 0.

Overwriting Outputs

Just as the reactions of the `Alignment` reactor overwrite the state variable `s`, logically simultaneous reactions can overwrite outputs. Consider the following example:

```
target Python;
reactor Overwriting {
    output y;
    state s(0);
    timer t1(100 msec, 100 msec);
    timer t2(200 msec, 200 msec);
    reaction(t1) -> y {=
        self.s += 1
        y.set(self.s)
    =}
    reaction(t2) -> y {=
        self.s -= 2
        y.set(self.s)
    =}
}
```

Here, the reaction to `t1` will set the output to 1 or 2, but every time it sets it to 2, the second
reaction (to `t2`) will overwrite the output with the value 0. As a consequence, the outputs will be
1, 0, 1, 0, ... deterministically.

Reacting to Outputs of Contained Reactors

A reaction may be triggered by the an input to the reactor, but also by an output of a contained
reactor, as illustrated in the following example:

```
target Python;
import Overwriting from "Overwriting.lf";
main reactor {
    s = new Overwriting();
    reaction(s.y) {=
        if s.y.value != 0 and s.y.value != 1:
            sys.stderr.write("ERROR: Outputs should only be 0 or 1!\n")
            exit(1)
    =}
}
```

This instantiates the above `Overwriting` reactor and monitors its outputs.

Method Declaration

A method declaration has one of the forms:

```
method <name>();
method <name>():<type>;
method <name>(<argument_name>:<type>, ...);
method <name>(<argument_name>:<type>, ...):<type>;
```

The first form defines a method with no arguments and no return value. The second form defines a method with the return type `<type>` but no arguments. The third form defines a method with a comma-separated list of arguments given by their name and type, but without a return value. Finally, the fourth form is similar to the third, but adds a return type.

Methods are particularly useful in reactors that need to perform certain operations on state variables and/or parameters that are shared between reactions or that are too complex to be implemented in a single reaction. Analogous to class methods, methods in LF can access all state variables and parameters, and can be invoked from all reaction bodies or from other methods. Methods may also recursively invoke themselves. Consider the following example:

```
target Python
main reactor Methods {
    state foo(2)
    method getFoo() {=
        return self.foo
    =}
    method add(x) {=
        self.foo += x
    =}
    reaction(startup){=
        print(f"Foo is initialized to {self.getFoo()}.")
        self.add(40)
        print(f"2 + 40 = {self.getFoo()}.")
    =}
}
```

This reactor defines two methods `getFoo` and `add` . The `getFoo` method receives no arguments and returns an integer (`int`) indicating the current value of the `foo` state variable. The `add` method returns nothing and receives one integer argument, which it uses to increment `foo` . Both methods are visible in all reactions of the reactor. In this example, the reaction to startup calls both methods in order to read and modify its state.

Causality Loops

Cycles

The interconnection pattern for a collection of reactors can form a cycle, but some care is required. Consider the following example:

```
target Python;
reactor A {
    input x;
    output y;
    reaction(x) -> y {=
        # ... something here ...
    =}
}
reactor B {
    input x;
    output y;
    reaction(x) {=
        # ... something here ...
    =}
    reaction(startup) -> y {=
        # ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x;
    b.y -> a.x;
}
```

This program yields the following diagram:

Lingua Franca diagram

The diagram highlights a **causality loop** in the program. At each tag, in reactor `B` , the first reaction has to execute before the second if it is enabled, a precedence indicated with the red dashed arrow. But the first can't execute until the reaction of `A` has executed, and that reaction cannot execute until the second reaction `B` has executed. There is no way to satisfy these requirements, so the tools refuse to generated code.

Cycles with Delays

One way to break the causality loop and get an executable program is to introduce a [logical delay](#) into the loop, as shown below:

```
target Python;
reactor A {
    input x;
    output y;
    reaction(x) -> y {=
        # ... something here ...
    =}
}
reactor B {
    input x;
    output y;
    reaction(x) {=
        # ... something here ...
    =}
    reaction(startup) -> y {=
        # ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x after 0;
    b.y -> a.x;
}
```


Lingua Franca diagram

Here, we have used a delay of 0, which results in a delay of one [microstep](#). We could equally well have specified a positive time value.

Reaction Order

Frequently, a program will have such cycles, but you don't want a logical delay in the loop. To get a cycle without logical delays, the reactions need to be reordered, as shown below:

```
target Python;
reactor A {
    input x;
    output y;
    reaction(x) -> y {=
        # ... something here ...
    =}
}
reactor B {
    input x;
    output y;
    reaction(startup) -> y {=
        # ... something here ...
    =}
    reaction(x) {=
        # ... something here ...
    =}
}
main reactor {
    a = new A();
    b = new B();
    a.y -> b.x;
    b.y -> a.x;
}
```

Lingua Franca diagram

There is no longer any causality loop.

Extending Reactors

Extending a Base Reactor

Actions

Action Declaration

An action declaration has one of the following forms:

```
logical action <name>(<min_delay>, <min_spacing>, <policy>)
physical action <name>(<min_delay>, <min_spacing>, <policy>)
```

The `min_delay` , `min_spacing` , and `policy` are all optional. If only one argument is given in parentheses, then it is interpreted as an `min_delay` , if two are given, then they are interpreted as `min_delay` and `min_spacing` . The `min_delay` and `min_spacing` are time values. The `policy` argument is a string that can be one of the following: `"defer"` (the default), `"drop"` , or `"replace"` . Note that the quotation marks are needed.

Logical Actions

Timers are useful to trigger reactions once or periodically. Actions are used to trigger reactions more irregularly. An action, like an output or input port, can carry data, but unlike a port, an action is visible only within the reactor that defines it.

There are two kinds of actions, **logical** and **physical**. A **logical action** is used by a reactor to schedule a trigger at a fixed logical time interval d into the future. The time interval d, which is called a **delay**, is relative to the logical time t at which the scheduling occurs. If a reaction executes at logical time t and schedules an action `a` with delay d, then any reaction that is triggered by `a` will be invoked at logical time $t + d$. For example, the following reaction schedules something (printing the current elapsed logical time) 200 msec after an input `x` arrives:

```
target Python;
reactor Schedule {
    input x;
    logical action a;
    reaction(x) -> a {=
        a.schedule(MSEC(200))
    =}
    reaction(a) {=
        elapsed_time = lf.time.logical_elapsed()
        print(f"Action triggered at logical time {elapsed_time} nsec after
    =}
}
```


Lingua Franca diagram

Here, the delay is specified in the call to schedule within the target language code. Notice that in the diagram, a logical action is shown as a triangle with an **L**. Logical actions are always scheduled within a reaction of the reactor that declares the action.

The time argument is required to be non-negative. If it is zero, then the action will be scheduled one **microstep** later. See [Superdense Time](#).

The arguments to the `a.schedule()` method is a time. The action `a` has to be declared as an effect of the reaction in order to reference it in the body of the reaction. If you fail to declare it as an effect (after the `->` in the reaction signature), then you will get a runtime error message.

The time argument to the `a.schedule()` method expects an integer. A collection of convenience functions is provided like the `MSEC` function above to specify time values in a more readable way. The provided functions are `NSEC`, `USEC` (for microseconds), `MSEC`, `SEC`, `MINUTE`, `HOUR`, `DAY`, and `WEEK`. You may also use the plural of any of these, e.g. `WEEKS(2)`.

An action may carry data, in which case, the **payload** data value is just given as a second argument to the `.schedule()` method. See the [Target Language Details](#).

Physical Actions

A **physical action** is used to schedule reactions at logical times determined by the local physical clock. If a physical action with delay *d* is scheduled at *physical* time *T*, then the *logical time* assigned to the event is *T* + *d*. For example, the following reactor schedules the physical action `p` to trigger at a **logical time** equal to the **physical time** at which the input `x` arrives:

```
target Python;
reactor Physical {
    input x;
    physical action a;
    reaction(x) -> a {=
        a.schedule(0)
    =}
    reaction(a) {=
        elapsed_time = lf.time.logical_elapsed()
        print(f"Action triggered at logical time {elapsed_time} nsec after
    =}
}
```


Lingua Franca diagram
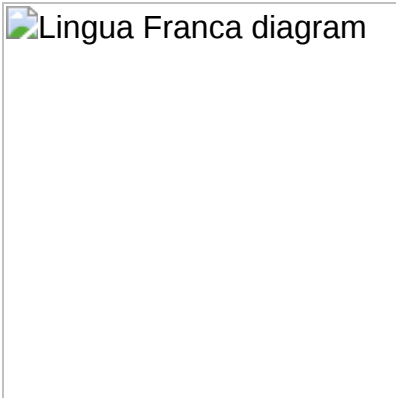
If you drive this with a timer, using for example the following structure:

Lingua Franca diagram

then running the program will yield an output something like this:

```
Action triggered at logical time 201491000 nsec after start.
Action triggered at logical time 403685000 nsec after start.
Action triggered at logical time 603669000 nsec after start.
...
```

Here, logical time is lagging physical time by a few milliseconds. Note that, unless the [fast option](#) is given, logical time t chases physical time T, so $t < T$. Hence, the event being scheduled in the reaction to input `x` is assured of being in the future in logical time.

Whereas logical actions are required to be scheduled within a reaction of the reactor that declares the action, physical actions can be scheduled by code that is outside the Lingua Franca system. For example, some other thread or a callback function may call `schedule()`, passing it a physical action. For example:

```
target Python;
main reactor {
    preamble {=
        import time
        import threading
        # Schedule an event roughly every 200 msec.
        def external(self, a):
            while (True):
                self.time.sleep(0.2)
                a.schedule(0)
    =}
    state thread;
    physical action a(100 msec);

    reaction(startup) -> a {=
        # Start a thread to schedule physical actions.
        self.thread = self.threading.Thread(target=self.external, args=(a,)
        self.thread.start()
    =}

    reaction(a) {=
        elapsed_time = lf.time.logical_elapsed()
        print(f"Action triggered at logical time {elapsed_time} nsec after
    =}
}
```


Lingua Franca diagram

Physical actions are the mechanism for obtaining input from the outside world. Because they are assigned a logical time derived from the physical clock, their logical time can be interpreted as a measure of the time at which some external event occurred.

Triggering Time for Actions

An action will trigger at a logical time that depends on the arguments given to the schedule function, the `<min_delay>`, `<min_spacing>`, and `<policy>` arguments in the action declaration, and whether the action is physical or logical.

For a **logical** action `a`, the tag assigned to the event resulting from a call to `schedule()` is computed as follows. First, let *t* be the *current logical time*. For a logical action, *t* is just the logical time at which the reaction calling `schedule()` is called. The **preliminary time** of the action is then just *t* + `<min_delay>` + `<offset>`. This preliminary time may be further modified, as explained below.

For a **physical** action, the preliminary time is similar, except that *t* is replaced by the current *physical* time *T* when `schedule()` is called.

If a `<min_spacing>` has been declared, then it gives a minimum logical time interval between the tags of two subsequently scheduled events. If the preliminary time is closer than `<min_spacing>` to the time of the previously scheduled event (if there is one), then `<policy>` (if supported by the target) determines how the minimum spacing constraint is enforced.

The `<policy>` is one of the following:

- `"defer"`: (**the default**) The event is added to the event queue with a tag that is equal to earliest time that satisfies the minimal spacing requirement. Assuming the time of the preceding event is *t_prev*, then the tag of the new event simply becomes *t_prev* + `<min_spacing>`.
- `"drop"`: The new event is dropped and `schedule()` returns without having modified the event queue.
- `"replace"`: The payload (if any) of the new event is assigned to the preceding event if it is still pending in the event queue; no new event is added to the event queue in this case. If the preceding event has already been pulled from the event queue, the default `"defer"` policy is applied.

Note that while the `"defer"` policy is conservative in the sense that it does not discard events, it could potentially cause an unbounded growth of the event queue.

Testing an Action for Presence

When a reaction is triggered by more than one action or by an action and an input, it may be necessary to test within the reaction whether the action is present. Just like for inputs, this can be done in the Python target with `a.is_present`, where `a` is the name of the action.

Superdense Time

Tag vs. Time

The model of time in Lingua Franca is a bit more sophisticated than we have hinted at. Specifically, a **superdense** model of time is used. In particular, instead of a **timestamp**, LF uses a **tag**, which consists of a **logical time** *t* and a **microstep** *m*.

A [logical action](#) may have a `<min_delay>` of zero, and the `<offset>` argument to the `schedule()` function may be zero. In this case, the call to `schedule()` appears to be requesting that the action trigger at the *current logical time*. Here is where superdense time comes in. The action will indeed trigger at the current logical time, but one microstep later. Consider the following example:

```
target Python;
main reactor {
    state count(1);
    logical action a;
    reaction(startup, a) {=
        print(
            f"{self.count}. Logical time is {lf.tag().time}. "
            f"Microstep is {lf.tag().microstep}."
        )
        if self.count < 5:
            a.schedule(0)
        self.count += 1
    =}
}
```


Lingua Franca diagram
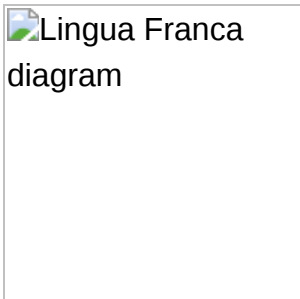
Executing this program will yield something like this:

```
1. Logical time is 1649607749415269000. Microstep is 0.
2. Logical time is 1649607749415269000. Microstep is 1.
3. Logical time is 1649607749415269000. Microstep is 2.
4. Logical time is 1649607749415269000. Microstep is 3.
5. Logical time is 1649607749415269000. Microstep is 4.
```

Notice that the logical time is not advancing, but the microstep is (the logical time, in this case, gives the number of nanoseconds that have elapsed since January 1, 1970). The general rule is that **every** call to `schedule()` advances the tag by at least one microstep.

Logical Simultaneity

Two events are **logically simultaneous** only if *both* the logical time and the microstep are equal. The following example illustrates this:

```
target Python;
reactor Destination {
    input x;
    input y;
    reaction(x, y) {=
        print(
            f"Time since start: {lf.time.logical_elapsed()}, "
            f"microstep: {lf.tag().microstep}"
        )
        if x.is_present:
            print("  x is present.")
        if y.is_present:
            print("  y is present.")
    =}
}
main reactor {
    logical action repeat;
    d = new Destination();
    reaction(startup) -> d.x, repeat {=
        d.x.set(1)
        repeat.schedule(0)
    =}
    reaction(repeat) -> d.y {=
        d.y.set(1)
    =}
}
```

Lingua Franca diagram

The `Destination` reactor has two inputs, `x` and `y`, and it reports in a reaction to either input what is the logical time, the microstep, and which input is present. The main reactor reacts to **startup** by sending data to the `x` input of `Destination`. It then schedules a `repeat` action with an `<offset>` of zero. The `repeat` reaction is invoked **strictly later**, one **microstep** later. The output printed, therefore, will look like this:

```
Time since start: 0, microstep: 0
  x is present.
Time since start: 0, microstep: 1
  y is present.
```

The reported elapsed logical time has not advanced in the second reaction, but the fact that `x` is not present in the second reaction proves that the first reaction and the second are not logically simultaneous. The second occurs one microstep later.

Alignment of Logical and Physical Times

Recall that in Lingua Franca, logical time "chases" physical time, invoking reactions at a physical time close to their logical time. For that purpose, the microstep is ignored.

Modal Reactors

The basic idea of *modal reactors* is to partition a reactor into disjoint subsets of reactions (or other components) that are associated with mutually exclusive *modes*. In a modal reactor, only a single mode can be active at a particular logical time instant, meaning that activity in other modes is automatically suspended. Transitioning between modes switches the reactor's behavior and provides control over continuing or resetting the previous history of the entered mode.

Syntax

Modes can be defined in any reactor, except federated ones. Each mode requires a unique (per reactor) name and can declare contents that are local to this mode. There must be exactly one mode marked as **initial**.

```
reactor TwoModes {
    ...

    initial mode One {
      ...
    }
    mode Two {
      ...
    }
}
```

A mode can contain local state variables, timers, actions, reactions, reactor instantiations, and connections. While modes cannot be nested in modes directly, hierarchical composition is possible through the instantiation of modal reactors. The main exception in allowed contents in modes are port declarations, as these are only possible on reactor level. Yet, modes share the scope with their reactor and, hence, can access ports, state variables, and parameters of the reactor. Only the contents of other modes are excluded. A modal reactor can still have reactions, reactor instantiations, etc., that are not located in modes and will consequently be executed independently from mode activity.

Mode transitions are declared within reactions. If a reactor has modes, reactions inside modes are allowed to list them as effects. Such an effect enables the use of the target language API to set the next mode.

```
reaction(trig) -> Two {=
    if trig.value:
        Two.set()
=}
```

You can also specify the type of the transition by adding the modifier `reset(<mode>)` or `continue(<mode>)` in the effect. The `reset` variant is implicitly assumed when the mode is listed without modifier.

Execution Semantics

The basic effect of modes is that only parts that are contained in the currently active mode (or outside any mode) are executed at any point in time. This also holds for parts that are nested in multiple *ancestor modes* due to hierarchy; consequently, all those ancestors must be active in order to execute. Reactions in inactive modes are simply not executed, while all components that model timing behavior, namely timers, scheduled actions, and delayed connections, are subject to a concept of *local time*. That means while a mode is inactive, the progress of time is suspended locally. How the timing components behave when a mode becomes active depends on the transition type. A mode can be *reset* upon entry, returning it to its initial state. Alternatively, it may *continue*, this only has an actual effect if the mode was active before. In the latter case all timing components will continue their delays or period as if no time had passed during inactivity of the mode. The following section will provide a more detailed explanation of this effect.

Upon reactor startup, the initial mode of each modal reactor is active, others are inactive. If at a tag *(t, m)*, all reactions of this reactor and all its contents have finished executing, and a new mode was set in a reaction, the current mode will be deactivated and the new one will be activated for future execution. This means no reaction of the newly active mode will execute at tag *(t, m)*; the earliest possible reaction in the new mode occurs one microstep later, at *(t, m+1)*. Hence, if the newly active mode has for example a timer that will elapse with an offset of zero, it will trigger at *(t, m+1)*. In case the mode itself does not require an immediate execution in the next microstep, it depends on future events, just as in the normal behavior of LF. Hence, modes in the same reactor are always mutually exclusive w.r.t. superdense time.

A transition is triggered if a new mode is set in a reaction body. As with setting output ports in reaction, a new mode can be set multiple times in the same or different reactions. In the end, the fixed ordering of reactions determines the last effective value that will be used. The new mode does not have to be a different one; it is possible for a mode to reset itself via a reset transition.

In case a mode is entered with the reset behavior:

- all contained modal reactors are reset to their initial mode (recursively),

- all contained timers are reset and start again awaiting their initial offset,

- all contained state variables that are marked for automatic reset are reset to their initial value,

- any contained reactions triggered by `reset` are executed, and

- all events (actions, timers, delayed connections) that were previously scheduled from within this mode are discarded.

Note that *contained* refers to all contents defined locally in the mode and in local reactor instances (recursively) that are not otherwise enclosed in modes of lower levels.

Whenever a mode is entered with a reset transition, the subsequent timing behavior is as if the mode was never executed before. If there are state variables that need to be reset or reinitialized, then this can be done in a reaction triggered by `reset` or by marking the state variable for automatic reset (e.g., `reset state x(0)`). State variables are not reset automatically to their initial conditions because it is idiomatic for reactors to allocate resources or initialize subsystems (e.g., allocate memory or sockets, register an interrupt, or start a server) in reactions triggered by the `startup`, and to store references to these resources in state variables. If these were to be automatically reset, those references would be lost.

On the other hand, if a mode has been active prior and is then re-entered via a `continue` transition, no reset is performed. Events originating from timers, scheduled actions, and delayed connections are adjusted to reflect a remaining delay equal to the remaining delay recorded at the instant the mode was previously deactivated. As a consequence, a mode has a notion of local time that elapses only when the mode is active.

Local Time

From the perspective of timers and actions, time is suspended when a mode is inactive. This also applies to indirectly nested reactors within modes and connections with logical delays, if their source lies within a mode.

Illustration of local time (model)

The above LF model illustrates the different characteristics of local time affecting timers and actions in the presence of the two transition types.

It consists of two modes `One` (the initial mode) and `Two`, both in the `Modal` reactor. The `next` input toggles between these modes and is controlled by a reaction at the top level that is triggered by the timer `T`. After one second, a mode switch is triggered periodically with a period one second. Each mode has a timer `T1`/`T2` that triggers a reaction after an initial offset of 100 msec and then periodically after 750 msec. This reaction then schedules a logical action with a delay of 500 msec (the actual target code does not add an additional delay over the minimum specified). This action triggers the second reaction, which writes to the output `out`. The main difference between the modes is that `One` is entered via a history transition, continuing its behavior, while `Two` is reset. (Continue behavior is indicated by an "H" on the transition edge because it enters into the entire history of the mode.)

Illustration of local time (trace)

Above is the execution trace of the first 4 seconds of this program. Below the timeline is the currently active mode and above the timeline are the model elements that are executed at certain points in time, together with indicating triggering and their relation through time. For example, at 100 msec, the initial offset of timer `T1` elapses, which leads to the scheduling of the logical action in this mode. The action triggers the reaction 500 msec later, at 600 msec, and thus causes an output. The timing diagram illustrates the different handling of time between history transitions and reset transitions. Specifically, when mode `One` is re-entered via a history transition, at time 2000 msec, the action triggered by `T1` before, at time 850 msec, resumes. In contrast, when mode `Two` is re-entered via a reset transition, at time 3000 msec, the action triggered by `T2` before, at time 1850 msec, gets discarded.

Illustration of local time (plot)

The above plot illustrates the relation between global time in the environment and the localized time for each timer in the model. Since the top-level reactor `TimingExample` is not enclosed by any mode, its time always corresponds to the global time. Mode `One` is the initial mode and hence progresses in sync with `TimingExample` for the first second. During inactivity of mode One the timer is suspended and does not advance in time. At 2000 msec it continues relative to this time. `T2` only starts advancing when the mode becomes active at 1000 msec. The reentry via reset at 3000 msec causes the local time to be reset to zero.

Startup and Shutdown

A challenge for modal execution is the handling `startup` and `shutdown` behavior. These are commonly used for managing memory for state variables, handling connections to sensors or actuators, or starting/joining external threads. If reactions to these triggers are located inside modes they are subject to a special execution regime.

First, `startup` reactions are invoked at most once at the first activation of a mode. Second, `shutdown` reactions are executed when the reactor shuts down, *irrespective* of mode activity, but only if the enclosing modes have been activated at least once. Hence, every startup has a corresponding shutdown. Third, as mentioned before, the new `reset` trigger for reactions can be used, if a startup behavior should be re-executed if a mode is entered with a reset transition.

Note that this may have unexpected implications:

- Startup behavior inside modes may occur during execution and not only at program start.

- Multiple shutdown reactions may be executed, bypassing mutual exclusion of modes.

- Reactors that are designed without consideration of modes and use only `startup` (not `reset`) to trigger an execution chain, may not work in modes and cease to function if re-entered with a reset.

Deadlines

Lingua Franca includes a notion of a **deadline**, which is a constraint on the relation between logical time and physical time. Specifically, a program may specify that the invocation of a reaction must occur within some *physical* time interval of the *logical* time of the message. If a reaction is invoked at logical time 12 noon, for example, and the reaction has a deadline of one hour, then the reaction is required to be invoked before the physical-time clock of the execution platform reaches 1 PM. If the deadline is violated, then the specified deadline handler is invoked instead of the reaction.

Purposes for Deadlines

A deadline in an LF program serves two purposes. First, it can guide scheduling in that a scheduler may prioritize reactions with deadlines over those without or those with longer deadlines. For this purpose, if a reaction has a deadline, then all upstream reactions on which it depends (without logical delay) inherit its deadline. Hence, those upstream reactions will also be given higher priority.

Second, the deadline mechanism provides a **fault handler**, a section of code to invoke when the deadline requirement is violated. Because invocation of the fault handler depends on factors beyond the control of the LF program, an LF program with deadlines becomes **nondeterministic**. The behavior of the program depends on the exact timing of the execution.

There remains the question of when the fault handler should be invoked. By default, deadlines in LF are **lazy**, meaning that the fault handler is invoked at the logical time of the event triggering the reaction whose deadline is missed. Specifically, the possible violation of a deadline is not checked until the reaction with the deadline is ready to execute. Only then is the determination made whether to invoke the regular reaction or the fault handler.

An alternative is an **eager deadline**, where a fault handler is invoked as soon as possible after a deadline violation becomes inevitable. With an eager deadline, if an event with tag (t, m) triggers a reaction with deadline D, then as soon as the runtime system detects that physical time $T > t + D$, the fault handler becomes enabled. This can occur at a logical time *earlier* than t. Hence, a fault handler may be invoked at a logical time earlier than that of the event that triggered the fault.

Note: As of this writing, eager deadlines are not implemented in any LF target language, so all deadlines are lazy.

Lazy Deadline

A lazy deadline is specified as follows:

```
target Python;
reactor Deadline {
    input x;
    output d; // Produced if the deadline is violated.
    reaction(x) -> d {=
        print("Normal reaction.")
    =} deadline(10 msec) {=
        print("Deadline violation detected.")
        d.set(x.value)
    =}
}
```

This reactor specifies a deadline of 10 milliseconds (this can be a parameter of the reactor). If the reaction to `x` is triggered later in physical time than 10 msec past the timestamp of `x`, then the second body of code is executed instead of the first. That second body of code has access to anything the first body of code has access to, including the input `x` and the output `d`. The output can be used to notify the rest of the system that a deadline violation occurred. This reactor can be tested as follows:

```
target Python;
import Deadline from "Deadline.lf";
preamble {= import time =}
main reactor {
    logical action a;
    d = new Deadline();
    reaction(startup) -> d.x, a {=
        d.x.set(0)
        a.schedule(0)
    =}
    reaction(a) -> d.x {=
        d.x.set(0)
        time.sleep(0.02)
    =}
    reaction(d.d) {=
        print("Deadline reactor produced an output.")
    =}
}
```

Lingua Franca diagram

Running this program will result in the following output:

```
Normal reaction.
Deadline violation detected.
Deadline reactor produced an output.
```

The first reaction of the `Deadline` reactor does not violate the deadline, but the second does. Notice that the sleep in the **main** reactor occurs *after* setting the output, but because of the deterministic semantics of LF, this does not matter. The actual value of an output cannot be known until every reaction that sets that output *completes* its execution. Since this reaction takes at least 20 msec to complete, the deadline is assured of being violated.

Notice that the deadline is annotated in the diagram with a small clock symbol.

Deadline Violations During Execution

Whether a deadline violation occurs is checked only *before* invoking the reaction with a deadline. What if the reaction itself runs for long enough that the deadline gets violated *during* the reaction

execution? For this purpose, a target-language function is provided to check whether a deadline is violated during execution of a reaction with a deadline.

Consider this example:

WARNING: No source file found: ../code/py/src/CheckDeadline.lf

Multiports and Banks

Lingua Franca provides a compact syntax for ports that can send or receive over multiple channels and another syntax for multiple instances of a reactor class. These are respectively called **multiports** and **banks of reactors**.

Multiports

To declare an input or output port to be a **multiport**, use the following syntax:

where `<width>` is a positive integer. This can be given either as an integer literal or a parameter name. Consider the following example:

```
target Python;
reactor Source {
    output[4] out;
    reaction(startup) -> out {=
        for i, port in enumerate(out):
            port.set(i)
    =}
}
reactor Destination {
    input[4] inp;
    reaction(inp) {=
        sum = 0
        for port in inp:
            if port.is_present: sum += port.value
        print(f"Sum of received: {sum}.")
    =}
}
main reactor {
    a = new Source();
    b = new Destination();
    a.out -> b.inp;
}
```

Executing this program will yield:

```
Sum of received: 6.
```

The `Source` reactor has a four-way multiport output and the `Destination` reactor has a four-way multiport input. These channels are connected all at once on one line, the second line from the last. Notice that the generated diagram shows multiports with hollow triangles. Whether it shows the widths is controlled by an option in the diagram generator.

NOTE: In `Destination`, the reaction is triggered by `in`, not by some individual channel of the multiport input. Hence, it is important when using multiport inputs to test for presence of the input on each channel, as done above with the syntax . An event on any one of the channels is sufficient to trigger the reaction.

The `Source` reactor also specifies `out` as an effect of its reaction using the syntax `-> out`. This brings into scope of the reaction body a way to access the width of the port and a way to write to each channel of the port.

In the Python target, multiports can be iterated on in a for loop (e.g., `for p in out`) or enumerated (e.g., `for i, p in enumerate(out)`) and the length of the multiport can be obtained by using the `len()` (e.g., `len(out)`) expression.

Parameterized Widths

The width of a port may be given by a parameter. For example, the above `Source` reactor can be rewritten

```
reactor Source(width:int(4)) {
    output[width] out:int;
    reaction(startup) -> out {=
        ...
    =}
}
```

Connecting Reactors with Different Widths

Assume that the `Source` and `Destination` reactors above both use a parameter `width` to specify the width of their ports. Then the following connection is valid:

```
main reactor {
    a1 = new Source(width = 3);
    a2 = new Source(width = 2);
    b = new Destination(width = 5);
    a1.out, a2.out -> b.in;
}
```

The first three ports of `b` will received input from `a1`, and the last two ports will receive input from `a2`. Parallel composition can appear on either side of a connection. For example:

```
    a1.out, a2.out -> b1.out, b2.out, b3.out;
```

If the total width on the left does not match the total width on the right, then a warning is issued. If the left side is wider than the right, then output data will be discarded. If the right side is wider than the left, then inputs channels will be absent.

Any given port can appear only once on the right side of the `->` connection operator, so all connections to a multiport destination must be made in one single connection statement.

Banks of Reactors

Using a similar notation, it is possible to create a bank of reactors. For example, we can create a bank of four instances of `Source` and four instances of `Destination` and connect them as follows:

```
main reactor {
    a = new[4] Source();
    b = new[4] Destination();
    a.out -> b.in;
}
```


Lingua Franca diagram

If the `Source` and `Destination` reactors have multiport inputs and outputs, as in the examples above, then a warning will be issued if the total width on the left does not match the total width on the right. For example, the following is balanced:

```
main reactor {
    a = new[3] Source(width = 4);
    b = new[4] Destination(width = 3);
    a.out -> b.in;
}
```

There will be three instances of `Source`, each with an output of width four, and four instances of `Destination`, each with an input of width 3, for a total of 12 connections.

To distinguish the instances in a bank of reactors, the reactor can define a parameter called **bank_index**. If such a parameter is defined for the reactor, then when the reactor is instanced in a bank, each instance will be assigned a number between 0 and n-1, where n is the number of reactor instances in the bank. For example, the following source reactor increments the output it produces by the value of `bank_index` on each reaction to the timer:

```
target Python;
reactor MultiportSource(
    bank_index(0)
) {
    timer t(0, 200 msec);
    output out;
    state s(0);
    reaction(t) -> out {=
        out.set(self.s)
        self.s += self.bank_index
    =}
}
```

The width of a bank may also be given by a parameter, as in

```
main reactor(
    source_bank_width:int(3),
    destination_bank_width:int(4)
) {
    a = new[source_bank_width] Source(width = 4);
    b = new[destination_bank_width] Destination(width = 3);
    a.out -> b.in;
}
```

Initializing Bank Members from a Table

It is often convenient to initialize parameters of bank members from a table. Here is an example:

```
target Python;
preamble {=
    table = [4, 3, 2, 1]
=}
reactor A(bank_index(0), value(0)) {
    reaction (startup) {=
        print("bank_index: {:d}, value: {:d}".format(self.bank_index, self.
    =}
}
main reactor {
    a = new[4] A(value = {= table[bank_index] =})
}
```

The global `table` defined in the **preamble** is used to initialize the `value` parameter of each bank member. The result of running this is something like:

```
bank_index: 0, value: 4
bank_index: 1, value: 3
bank_index: 2, value: 2
bank_index: 3, value: 1
```

Contained Banks

Banks of reactors can be nested. For example, note the following program:

```
target Python;
reactor Child (
    bank_index(0)
) {
    reaction(startup) {=
        print(f"My bank index: {self.bank_index}.")
    =}
}
reactor Parent (
    bank_index(0)
) {
    c = new[2] Child();
}
main reactor {
    p = new[2] Parent();
}
```


Lingua Franca diagram

In this program, the `Parent` reactor contains a bank of `Child` reactor instances with a width of 2. In the main reactor, a bank of `Parent` reactors is instantiated with a width of 2, therefore, creating 4 `Child` instances in the program in total. The output of this program will be:

```
My bank index: 0.
My bank index: 1.
My bank index: 0.
My bank index: 1.
```

The order of these outputs will be nondeterministic if the execution is multithreaded (which it will be by default) because there is no dependence between the reactions, and, hence, they can execute in parallel.

The bank index of a container (parent) reactor can be passed down to contained (child) reactors. For example, note the following program:

```
target Python;
reactor Child (
    bank_index(0),
    parent_bank_index(0)
) {
    reaction(startup) {=
        print(
            f"My bank index: {self.bank_index}. "
            f"My parent's bank index: {self.parent_bank_index}."
        )
    =}
}
reactor Parent (
    bank_index(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
}
main reactor {
    p = new[2] Parent();
}
```

In this example, the bank index of the `Parent` reactor is passed to the `parent_bank_index` parameter of the `Child` reactor instances. The output from this program will be:

```
My bank index: 1. My parent's bank index: 1.
My bank index: 0. My parent's bank index: 0.
My bank index: 0. My parent's bank index: 1.
My bank index: 1. My parent's bank index: 0.
```

Again, note that the order of these outputs is nondeterministic.

Finally, members of contained banks of reactors can be individually addressed in the body of reactions of the parent reactor if their input/output port appears in the reaction signature. For example, note the following program:

```
target Python;
reactor Child (
    bank_index(0),
    parent_bank_index(0)
) {
    output out;
    reaction(startup) -> out {=
        out.set(self.parent_bank_index * 2 + self.bank_index)
    =}
}
reactor Parent (
    bank_index(0)
) {
    c = new[2] Child(parent_bank_index = bank_index);
    reaction(c.out) {=
        for i, child in enumerate(c):
            print(f"Received {child.out.value} from child {i}.")
    =}
}
main reactor {
    p = new[2] Parent();
}
```


Lingua Franca diagram

Running this program will give something like the following:

```
Received 0 from child 0.
Received 1 from child 1.
Received 2 from child 0.
Received 3 from child 1.
```

Note that `len(c)` can be used to get the width of the bank, and `for p in c` or `for (i, p) in enumerate(c)` can be used to iterate over the bank members.

Combining Banks and Multiports

Banks of reactors may be combined with multiports, as in the following example:

```
target Python;
reactor Source {
    output[3] out;
    reaction(startup) -> out {=
        for i, port in enumerate(out):
            port.set(i)
    =}
}
reactor Destination(
    bank_index(0)
) {
    input inp;
    reaction(inp) {=
        print(f"Destination {self.bank_index} received {inp.value}.")
    =}
}

main reactor MultiportToBank {
    a = new Source();
    b = new[3] Destination();
    a.out -> b.inp;
}
```

Lingua Franca diagram

The three outputs from the `Source` instance `a` will be sent, respectively, to each of three instances of `Destination`, `b[0]`, `b[1]`, and `b[2]`. The result of the program will be something like the following:

```
Destination 0 received 0.
Destination 1 received 1.
Destination 2 received 2.
```

Again, the order is nondeterministic in a multithreaded context.

The reactors in a bank may themselves have multiports. In all cases, the number of ports on the left of a connection must match the number on the right, unless the ones on the left are iterated, as explained next.

Broadcast Connections

Occasionally, you will want to have fewer ports on the left of a connection and have their outputs used repeatedly to broadcast to the ports on the right. In the following example, the outputs from an ordinary port are broadcast to the inputs of all instances of a bank of reactors:

```
reactor Source {
        output out:int;
        reaction(startup) -> out {=
                ... write to out ...
        =}
}
reactor Destination {
        input in:int;
        reaction(in) {=
                ... read from in ...
        =}
}
main reactor ThreadedThreaded(width:int(4)) {
        a = new Source();
        d = new[width] Destination();
        (a.out)+ -> d.in;
}
```

The syntax `(a.out)+` means "repeat the output port `a.out` one or more times as needed to supply all the input ports of `d.in` ." The content inside the parentheses can be a comma-separated list of ports, the ports inside can be ordinary ports or multiports, and the reactors inside can be ordinary reactors or banks of reactors. In all cases, the number of ports inside the parentheses on the left must divide the number of ports on the right.

Interleaved Connections

Sometimes, we don't want to broadcast messages to all reactors, but need more fine-grained control as to which reactor within a bank receives a message. If we have separate source and destination reactors, this can be done by combining multiports and banks as was shown in [Combining Banks and Multiports](#). Setting a value on the index n of the output multiport, will result in a message to the n-th reactor instance within the destination bank. However, this pattern gets slightly more complicated, if we want to exchange addressable messages between instances of the same bank. This pattern is shown in the following example:

```
target Python;
reactor Node(
    num_nodes(4),
    bank_index(0)
) {
    input[num_nodes] inp;
    output[num_nodes] out;

    reaction (startup) -> out {=
        out[1].set(42)
        print(f"Bank index {self.bank_index} sent 42 on channel 1.")
    =}

    reaction (inp) {=
        for i, port in enumerate(inp):
            if port.is_present:
                print(
                    f"Bank index {self.bank_index} received {port.value} on
                )
    =}
}
main reactor(num_nodes(4)) {
    nodes = new[num_nodes] Node(num_nodes=num_nodes);
    nodes.out -> interleaved(nodes.inp);
}
```


Lingua Franca diagram

In the above program, four instance of `Node` are created, and, at startup, each instance sends 42 to its second (index 1) output channel. The result is that the second bank member (`bank_index` 1) will receive the number 42 on each input channel of its multiport input. Running this program gives something like the following:

```
Bank index 0 sent 42 on channel 1.
Bank index 1 sent 42 on channel 1.
Bank index 2 sent 42 on channel 1.
Bank index 3 sent 42 on channel 1.
Bank index 1 received 42 on channel 0.
Bank index 1 received 42 on channel 1.
Bank index 1 received 42 on channel 2.
Bank index 1 received 42 on channel 3.
```

In bank index 1, the 0-th channel receives from `bank_index` 0, the 1-th channel from `bank_index` 1, etc. In effect, the choice of output channel specifies the destination reactor in the bank, and the input channel specifies the source reactor from which the input comes.

This style of connection is accomplished using the new keyword **interleaved** in the connection. Normally, a port reference such as `nodes.out` where `nodes` is a bank and `out` is a multiport, would list all the individual ports by first iterating over the banks and then, for each bank index, iterating over the ports. If we consider the tuple (b,p) to denote the index b within the bank and the index p within the multiport, then the following list is created: (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3). However, if we use **interleaved** `(nodes.out)` instead, the connection logic will iterate over the ports first and then the banks, creating the following list: (0,0), (1,0), (2,0), (3,0), (0,1), (1,1), (2,1), (3,1), (0,2), (1,2), (2,2), (3,2), (0,3), (1,3), (2,3), (3,3). By combining a normal port reference with a interleaved reference, we can construct a fully connected network. The figure below visualizes this how this pattern would look without banks or multiports:

Lingua Franca diagram

If we were to use a normal connection `nodes.out -> nodes.in;` instead of the **interleaved** connection, then the following pattern would be created:

Lingua Franca diagram

Effectively, this connects each reactor instance to itself, which isn't very useful.

Preambles

Preamble

Reactions may contain arbitrary target-language code, but often it is convenient for that code to invoke external libraries or to share procedure definitions. For either purpose, a reactor may include a **preamble** section.

For example, the following reactor uses the `platform` module to print the platform information and a defined method to add 42 to an integer:

```
main reactor Preamble {
        preamble {=
                import platform
                def add_42(self, i):
                        return i + 42
        =}
        timer t;
        reaction(t) {=
                s = "42"
                i = int(s)
                print("Converted string {:s} to int {:d}.".format(s, i))
                print("42 plus 42 is ", self.add_42(42))
                print("Your platform is ", self.platform.system())
        =}
}
```

On a Linux machine, this will print:

```
Converted string 42 to int 42.
42 plus 42 is 84
Your platform is Linux
```

By putting import in the **preamble**, the module becomes available in all reactions of this reactor using the `self` modifier.

Note: Preambles will be put in the generated Python class for the given reactor, and thus is part of the instance of the reactor. This means that anything you put in the preamble will be specific to a particular reactor instance and cannot be used to share information between different instantiations of the reactor (this is a feature, not a bug, because it helps ensure determinacy).

For more information about implementation details of the Python target, see [Implementation Details](#).

Alternatively, you can define a **preamble** outside any reactor definition. Such a **preamble** can be used for functions such as import or to define a global function. The following example shows importing the [hello](#) module:

```
target Python {
    files: include/hello.py
};
preamble {=
    import hello
=}
```

Notice the usage of the `files` target property to move the `hello.py` module located in the `include` folder of the test directory into the working directory (located in `src-gen/NAME`).

For another example, the following program uses the built-in Python `input()` function to get typed input from the user:

```
target Python
main reactor {
    preamble {=
        import threading
        def external(self, a):
            while (True):
                from_user = input() # Blocking
                a.schedule(0, from_user)
    =}
    state thread
    physical action a
    timer t(2 secs, 2 secs)

    reaction(startup) -> a {=
        self.thread = self.threading.Thread(target=self.external, args=(a,)
        self.thread.start()
        print("Type something.")
    =}

    reaction(a) {=
        elapsed_time = lf.time.logical_elapsed()
        print(f"A time {elapsed_time} nsec after start, received: ", a.valu
    =}

    reaction(t) {=
        print("Waiting ...")
    =}
}
```

Within the **preamble**, we specify to import the `threading` Python module and define a function
that will be started in a separate thread in the reaction to **startup**. The thread function named
`external` blocks when `input()` is called until the user types something and hits the return or
enter key. Usually, you do not want a Lingua Franca program to block waiting for input. In the
above reactor, a **timer** is used to repeatedly trigger a reaction that reminds the user that it is
waiting for input.

Distributed Execution

Termination

Shutdown Reactions

There are several mechanisms for terminating a Lingua Franca in an orderly fashion. All of these mechanisms result in a **final tag** at which any reaction that declares **shutdown** as a trigger will be invoked (recall that a **tag** is a tuple (**logical time**, **microstep**)). Other reactions may also be invoked at this final tag, and the order in which reactions are invoked will be constrained by the normal precedence rules.

If a reaction triggered by **shutdown** produces outputs, then downstream reactors will also be invoked at the final tag. If the reaction schedules any actions by calling `schedule()`, those will be ignored. In fact, any event after the final tag will be ignored. After the completion of the final tag, the program will exit.

There are four ways to terminate a program:

- **Timeout**: The program specifies the last logical time at which reactions should be triggered.
- **Starvation**: At the conclusion of some tag, there are no events in the event queue at future tags.
- **Stop request**: Some reaction requests that the program terminate.
- **External signal**: Program is terminated externally using operating services like control-C or `kill`.

We address each of these in turn.

Timeout

The [target property `timeout`](#) specifies the last logical time at which reactions should be triggered. The last invocation of reactions will be at tag (`timeout`, 0).

There is a significant subtlety when using [physical connections](#), which are connections using the syntax `~>`. Such connections specify that the tag at the receiving end will be based on the physical time at which the message is received. If the tag assigned at the receiving end is greater than the final tag, then the message is lost. Hence, **messages sent near the `timeout` time are likely to be lost!**

Starvation

If a Lingua Franca program has no [physical actions](), and if at any time during execution there are no future events waiting to be processed, then there is no possibility for any more reactions to occur and the program will exit. This situation is called **starvation**. If there is a **timer** anywhere in the program with a period, then this condition never occurs.

One subtlety is that reactions triggered by **shutdown** will be invoked one microstep later than the last tag at which there was an event. They cannot be invoked at the same tag because it is only after that last tag has completed that the runtime system can be sure that there are no future events. It would not be correct to trigger the **shutdown** reactions at that point because it would be impossible to respect the required reaction ordering.

Stop Request

If a reaction calls the built-in `request_stop()` function, then it is requesting that the program cease execution as soon as possible. This cessation will normally occur in the next microstep. The current tag will be completed as normal. Then the tag will be advanced by one microstep, and reactions triggered by **shutdown** will be executed, along with any other reactions with triggers at that tag, with all reactions executed in precedence order.

External Signal

A control-C or other kill signal to a running Lingua Franca program will cause execution to stop immediately.