

User's Guide to the HP DMF User Interface and Macro Language

Evan Kirshenbaum

Thursday, February 10, 2022

*What this document is
and what it is not*

This document explains how to install and run the user interface software for interactive control of the prototype HP digital microfluidics board. It also covers the syntax and concepts for the DMF macro language that can be used for more complex control via the user interface.

Other documents will cover the installing and running the physical board as well as the Python API that allows the simple specification of complete protocols using the board.

Installing the software

Python version

This section assumes that you have a copy of the Python source code, probably obtained as a zip file.

The code has been tested with Python version 3.9.4. Versions less than 3.9 are known to not work.

Required packages

The following packages are required. They can be installed using **pip**:

- `antlr4-python3-runtime`
- `clipboard`
- `matplotlib`
- `pyserial`

Setting \$PYPATH

The following directories must be included in your \$PYTHONPATH environment variable:

- `${ROOT}/src`
- `${ROOT}/tools`
- `${ROOT}/target/generated-sources/antlr4`

Depending on how the zip file was generated \${ROOT} will either be the root directory of the zip file or a subdirectory called mpam.

Running the software

This section assumes that you are trying to control a “Wombat” board (an HP DMF board controlled via an OpenDrop controller). In this configuration only the following are available (assuming a board orientation with the OpenDrop controller on the top):

- The bottom two wells on each side. On the full boards, the wells are numbered from 0 to 3 down the left side and 4 to 7 down the right side. Thus, wells 2, 3, 6, and 7 are available.
- The three pads leading away from (in the same rows as the exit pads for) wells 2, 3, and 7.
- The two pads leading away from (in the same row as the exit pad for) well 6.
- All of the pads in rows 1 through 5 (with zero being the bottom row).

The UI is designed to be run from the command line. The basic structure is

```
python wombat task arguments...
```

where the **wombat** program can be found in the **tools** directory. For simple interactive control, the task you want to run is called **display-only**, which can be abbreviated to **display**.¹

```
python wombat display arguments ...
```

Command-line arguments

The tool takes several command-line arguments. A full set can be obtained by giving the **--help** argument:

```
python wombat display --help
```

Several of these arguments take time values, which must be specified as a number followed by a time unit. Recognized units include

| | |
|--------------------------------------|--------------|
| ns, nsec | nanoseconds |
| us, usec | microseconds |
| ms, msec | milliseconds |
| s, sec, secs, second, seconds | seconds |
| min, minute, minutes | minutes |
| hr, hour, hours | hours |
| day, days | days |

Note that a time value is a single argument, so if a space is included between the number and the unit, the argument must be enclosed in quotes or the space must be escaped using a backslash.

The arguments taken by the program include the following.² Note that long arguments (which start with two dashes) can be abbreviated to any unique prefix.

| Long | Short | Type | Default | Description |
|---------------------------------|---------------------|------|----------|---|
| --help | | | | Print the help on the task and exit |
| --clock-speed <i>T</i> | -cs <i>T</i> | Time | 100 ms | The amount of time between clock ticks |
| --paused | | | false | Don't start the clock |
| --initial-delay <i>T</i> | | Time | 5 sec | The amount of time to wait before running the task. |
| --min-time <i>T</i> | | Time | 5 min | The minimum amount of time to leave the display on, even if the operation has finished. |
| --max-time <i>T</i> | | Time | no limit | The maximum amount of time to leave the display up. |

¹ Note that it cannot be abbreviated to “disp”, as that will try to run the “dispense” task.

² Plus arguments specific to the task. The **display** task has no additional arguments.

| | | | | |
|----------------------------|-----|------|-------|--|
| --no-display | -nd | | | Run the task without the on-screen display |
| --update-interval <i>T</i> | | Time | 20 ms | The maximum amount of time between display updates |
| --macro-file | | File | | A file containing DMF macro definitions |

For the wombat exerciser, there is also

| Long | Short | Type | Default | Description |
|-----------------|-------------|--------|---------|---|
| --port <i>P</i> | -p <i>P</i> | String | | The USB port to use to talk to the board. |

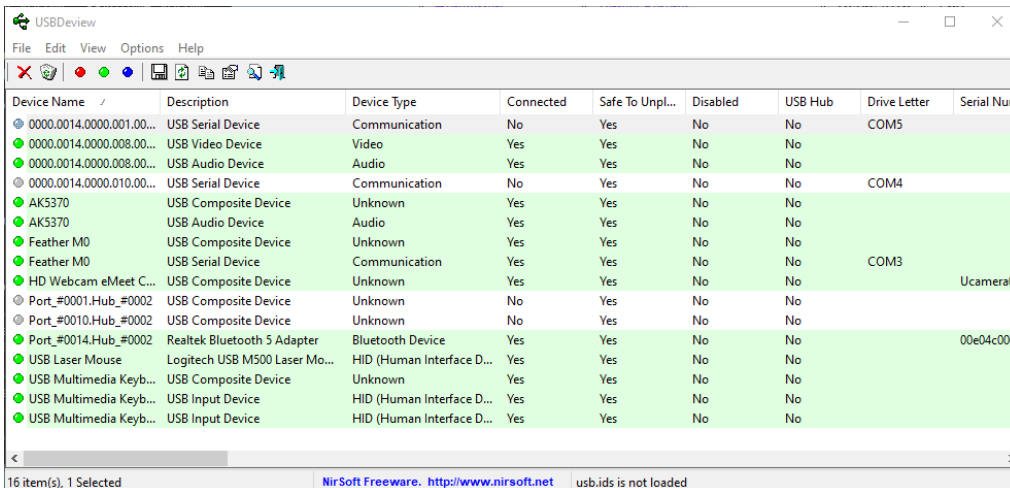
For interactive control, you will probably want to set --min-time to a largish value (e.g., 1 day) to prevent the program from exiting on you unexpectedly. (The display-only task takes no time to execute.) You can adjust the clock speed and whether the clock is paused from the display, so you need not specify it on the command line.

If you are planning on using macro language expressions, you will probably want to specify a macro file containing macros that you may want to make use of. The macro language will be described in detail below and an example file can be found in inputs/macros.dmf.

Finding the USB address

To control an actual Wombat board, the --port argument must be provided. (If it is not, the display will come up and you can interact with it, but it will have no effect on the physical board.)

To determine the USB address, the easiest way is to run a tool called USBDevice, which can be obtained at https://www.nirsoft.net/utils/usb_devices_view.html. The download link (near the bottom of the page) will download a zip file, and the program can be run directly from this file (without extracting it). Running it will bring up a display like



| Device Name | Description | Device Type | Connected | Safe To Unpl... | Disabled | USB Hub | Drive Letter | Serial Num |
|--------------------------|-------------------------------|---------------------------|-----------|-----------------|----------|---------|--------------|------------|
| 0000.0014.0000.001.00... | USB Serial Device | Communication | No | Yes | No | No | COM5 | |
| 0000.0014.0000.008.00... | USB Video Device | Video | Yes | Yes | No | No | | |
| 0000.0014.0000.008.00... | USB Audio Device | Audio | Yes | Yes | No | No | | |
| 0000.0014.0000.010.00... | USB Serial Device | Communication | No | Yes | No | No | COM4 | |
| AK5370 | USB Composite Device | Unknown | Yes | Yes | No | No | | |
| AK5370 | USB Audio Device | Audio | Yes | Yes | No | No | | |
| Feather M0 | USB Composite Device | Unknown | Yes | Yes | No | No | | |
| Feather M0 | USB Serial Device | Communication | Yes | Yes | No | No | COM3 | |
| HD Webcam eMeet C... | USB Composite Device | Unknown | Yes | Yes | No | No | | Ucamera00 |
| Port_#0001.Hub_#0002 | USB Composite Device | Unknown | No | Yes | No | No | | |
| Port_#0010.Hub_#0002 | USB Composite Device | Unknown | No | Yes | No | No | | |
| Port_#0014.Hub_#0002 | Realtek Bluetooth 5 Adapter | Bluetooth Device | Yes | Yes | No | No | | 00e04c0000 |
| USB Laser Mouse | Logitech USB M500 Laser Mo... | HID (Human Interface D... | Yes | Yes | No | No | | |
| USB Multimedia Keyb... | USB Composite Device | Unknown | Yes | Yes | No | No | | |
| USB Multimedia Keyb... | USB Input Device | HID (Human Interface D... | Yes | Yes | No | No | | |
| USB Multimedia Keyb... | USB Input Device | HID (Human Interface D... | Yes | Yes | No | No | | |

The device name you want to look for for the OpenDrop controller is “Feather M0”, and the option value is found in the “Drive Letter” column, in this case COM3, so the argument to specify would be --port COM3.

Command-line summary

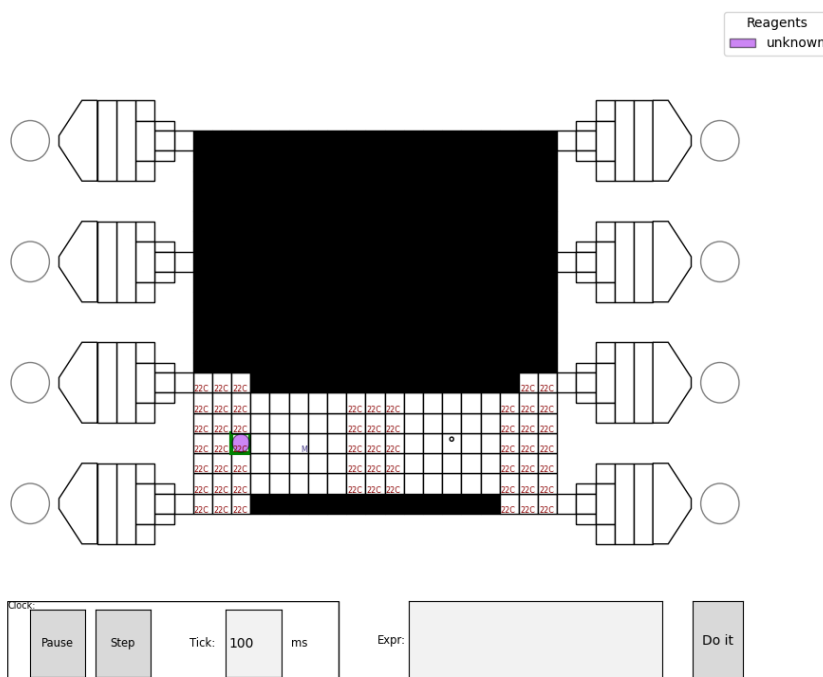
In summary, the most likely command line to specify for pure interactive use will be something like

```
python tools/wombat display-only
--port COM3
--min-time 1day
--macro-file inputs/macros.dmf
```

(on a single line, of course).

The display

When you run the program, you should see a display like this:



At the top of the display is a visual representation of the Wombat board with the parts of the board that are not controllable blacked out. Colored circles over pads represent drops, and the color indicates the presumed reagent (or mixture of reagents). The colors for reagents currently on the board are shown in the “Reagents” legend at the top right. (For interactive control, the only reagent you are likely to see is “unknown”.) The area of the circle is proportional to the size of the drop.

Note that drops shown on the board are inferred from mouse clicks and statements in the macro language—they do not necessarily correspond to actual physical drops on the board, as there is no actual drop sensing capability in the current version of the hardware.

Pads whose electrode is powered are shown with a green border. Those whose electrode is off are shown with a black border. When controlling the board via mouse clicks (described below), a thick border indicates the electrode state that will be transitioned to in the next clock tick (necessarily the opposite of its current state).

The numbers on some pads indicate the inferred temperature of thermal zones. These are inactive in the wombat board and should always report 22C (room temperature). The M on pad (2, 3) represents the state of a (similarly not yet controllable) magnet, and the small circle on pad (13, 3) represents an “extraction port” to be used for adding or removing fluid, typically under the control of a pipetting robot. All of these can be ignored at this point.

At the bottom of the display are buttons and text boxes that can be used to control the board. To the left are controls to pause (and resume) the clock, allow a paused clock to proceed for one clock tick, and to specify the clock speed in milliseconds. To the right is a text box used to enter macro language expressions.

Interactive control

Each of the sections of the display described above allows the user to interactively control the state of the physical board. On the board image, you can click on pads (including pads in wells) to turn them on and off. In the clock region, you can control the speed of the clock and whether or not it is running. And in the expression region you can issue commands written in the macro language to perform more complex actions.

Controlling the clock

The software maintains an internal clock that ticks at a steady rate.³ When multiple electrode state changes are specified at the same time, they all take place on the next clock tick. By default, the clock ticks once every 100 ms, but it can be sped up or slowed down either by use of the `--clock-speed` command-line argument (e.g., `--clock-speed 200ms`) or by specifying a number in the text box.⁴

The clock starts out running, but this can be changed by specifying the `--pause` command-line argument or by pressing the **Pause** button (which then turns into a **Run** button). When the clock is paused, multiple operations may be requested using the mouse or expression box, but they will not take effect until the **Run** button is pressed to restart the clock or the **Step** button is pressed to allow the clock to proceed for one tick and then reenter the paused state.

Using the mouse

You can click on the board to toggle the state of pads (including well pads). If the clock is running, the change of state happens on the next clock tick. If the clock is paused, the change of state will happen—well, still on the next clock tick, but this won't happen until you press **Run** or **Step**. Until then, you will see thick borders around the pads scheduled to toggle, with the color indicating their new state. If you click a second time on a pad, it deselects it from the list of changes.

The effect of clicking was meant to be unsurprising, but it sounds complicated when described. Basically, the system tries to model the effect of what changing pad state means (ideally) to the drops on the board. In particular:

- If you hold down **control** while clicking, you specify that the pad should be toggled.
- If you don't hold down **control** and you click on an empty pad next to a drop, it not only turns the clicked pad on but also schedules any neighboring pads that hold drops to turn off. The notion is that by not holding **control**, you are specifying that you want to “pull” the neighboring drop(s) toward the pad. Note that neighboring pads include diagonal neighbors.

The logic for what happens to modeled drops is as follows:

- If a pad is turned on and none of its neighbors hold a drop, a new drop is created at that pad. This can be useful to model physical drops added to the board that don't match the model.
- If a pad is turned off and the drop isn't pulled by a neighbor pad, the modelled drop is removed from the model. This can also be useful to correct errors.

³ The code is written in Python, so the rate is somewhat approximate, but it should be stable enough for the purposes of moving drops around.

⁴ For Wombat boards, it appears that limitations of the OpenDrop controller make it impossible to meaningfully set the clock interval less than 100 ms.

- If a pad with a drop is turned off and pulled to a single neighbor, the entire drop is moved to that neighbor pad.
 - If the neighbor pad is a well gate pad, the drop is removed from the board and the volume of the well is increased.
 - If the pad is the exit pad from a well and the well's gate was on, a new drop is created at the exit pad and the volume of the well is decreased.
- If a pad is turned off and pulled to multiple neighbors, the drop is split evenly and pulled in all of the directions.
 - If the drop is pulled toward a neighbor, but its own electrode is to remain on, it is similarly split.
- If multiple drops (or parts of drops) are moved to the same pad, they are merged into one larger drop at the destination pad.

Using macro language commands

Finally, in the text box at the bottom right, you can type macro language expressions. The next section will go into the language in detail, but some examples:

- `d = drop @ (2,3)`
 - Bind variable `d` to a drop at the pad at column 2, row 3 (with the bottom left corner being (0,0). If there is no drop there, one is created in the model.
- `d : right 3`
 - Move the drop `d` right three pads.
- `d + 2 up : on`
 - Turn on the pad two rows above drop `d`.
- `d : right 2 : up 3`
 - Move the drop two columns to the right and then 3 rows up.
- `rectangle(d, 2 right, 3 up)`
 - Call the macro bound to `rectangle`, which takes three arguments, a drop, and two “deltas” describing the directions and lengths of the first two sides.
- `w = well #3`
 - Bind variable `w` to well number three (the bottom well on the left-hand side).
- `d = dispense(w)`
 - Call the macro bound to `dispense`, passing in well `w` as an argument. Bind the resulting drop to `d`.
- `d = w : dispense`
 - The same things. The `who : what` “injection” syntax can be used for macros that take one argument.
- `p = d + 1 up`
 - Bind `p` to the pad immediately above `d`.
- `[[d : turn off; p : turn on;]]`
 - In parallel, turn off the pad under drop `d` and turn on the pad at `p`.

When you enter an expression (either by hitting **ENTER** or by clicking the **Do it** button), the expression will be printed to the console output. When the expression completes, the result will also be printed. So for the first expression

```
d = drop @ (2,3)
```

the program would print

```
Interactive cmd: d = drop @ (2,3)
```

```
Interactive cmd val (DROP): Drop[Pad(2,3), 0.5 µl of unknown]
```

This tells you that the result of the expression was a drop and that the value was the drop at pad (2,3), presumed to contain 0.5 µl of an unknown reagent.

If the expression contains an error, but it is reasonably clear what the mistake was, it will still be executed. So, for the input

```
d = drop @ (2,3
```

the program would print

```
Interactive cmd: d = drop @ (2,3
line 1:15 missing ')' at '<EOF>'
Interactive cmd val (DROP): Drop[Pad(2,3), 0.5 µl of unknown]
```

If the interpreter cannot figure out what you meant or if you try to combine objects that cannot be combined (e.g., asking to multiply a drop and an integer), it will say

```
Expression contained error, not evaluating.
```

In the case of a type error, the error message will tell you what sorts of things it could have done:

```
Interactive cmd: d*3
line 1:0 Cannot compute DROP * INT: d * 3
expected one of:
  FLOAT * FLOAT -> FLOAT
  FLOAT * TICKS -> TICKS
  FLOAT * TIME -> TIME
  FLOAT * VOLUME -> VOLUME
  INT * INT -> INT
  TICKS * FLOAT -> TICKS
  TIME * FLOAT -> TIME
  VOLUME * FLOAT -> VOLUME
Expression contained error, not evaluating.
Interactive cmd val (NONE): None
```

Typing expressions First off, yes, the input text box is terrible. There is no line wrap and no clipping at the box boundaries. The display package⁵ that is being tortured into doing unnatural things for this UI really isn't designed for this sort of thing. At some point, it will be replaced. But for now, it's good enough to type shortish expressions. For anything more complicated, you're almost certainly going to want to use a text editor (e.g., Emacs, Notepad, Eclipse) to create a file of macro definitions that you can refer to. This file will be read in and executed when the program is run if you give it as the value following the `--macro-file` command-line argument.

To help a bit, the following are available:

- The left and right arrows allow you to move forward and backward within the text. Sometimes you have to click in the box first. I don't know why.
- The up and down arrows scroll backward and forward through the expression history. This is useful when you typed something wrong and want to go back and fix it or when you want to go back and repeat an earlier command, perhaps with modifications.
- Ctrl-X and Ctrl-C cut and copy the whole text in the box. There is no way to select partial text. This can be useful both to pull pieces from earlier commands to paste into one you're building and also to grab snippets to paste into a file using a text editor.

⁵ Matplotlib.

- Ctrl-V pastes the contents of the system clipboard at the current cursor position.

The macro language

The DMF macro language allows you to define complex actions that can be used in interactive expressions. It is not (yet, at least) a full language—for example, there are (ditto) no looping constructs other than those built into the drop motion operators and no user-defined types—and it is not expected that full protocols will be built with it,⁶ but it should make the process of interactively testing the board easier.

OVERVIEW

The language has been designed as a sort of “warm-up exercise” for a follow-on language for building protocols, and that language is aimed at non-programmers, so the macro language has a couple of features that are a bit strange for a programming language.

Note: This document is being written before I leave on vacation, so I'm documenting the language as it has currently been released. This necessarily means that there are quite a number of places where I'm looking at it and saying, “That needs to be fixed.” Since time is short, I'm documenting things as they are, even if the fix looks to be trivial. I'll update the document when I get back and fix things.

Natural language-ish

First of all, there are constructs that are more common in English than in programming languages.

For example, rather than saying something like `w.exit_pad` to obtain the exit pad for well `w` (the pad on the board proper immediately in front of it), in this language, you say `w's exit pad`. The `'s` operator is used to obtain an attribute of an object, and some built-in attributes and operations are specified with multiple words.

Similarly, there are sometimes “noise words”—optional words that allow you to be more verbose. For example, the action of turning on a pad's electrode can be specified as `turn on` or simply as `on`. Removing a drop from the board is minimally `remove`, but it can also be `remove from board` or `remove from the board`.

Along the same lines, there are often multiple ways to say things. The direction toward the top of the board can be `up` or `north`. If `dir` holds that direction as a value, you can get the value `right` (or `east`) by saying `dir turned right` or `dir turned clockwise`. The word `column` can be abbreviated as `col`. A time can be specified as `2 s` or `2 sec` or `2 seconds`.

Finally, while you are required to specify the type of each parameter to a defined macro, giving them names is optional. If you prefer, you can simply refer to them as, e.g., `the drop` (or simply `drop`) or `pad 3`.

DMF-relevant types

In addition to normal programming types like integers, floating point numbers, and boolean values, the macro language also supports types for concepts that are important in the domain. Drops, pads, and wells are first-class objects, as are directions (e.g., `right`), deltas (e.g., `3 up`), time (e.g., `100ms`), and motions (e.g., to row 5)

PRELIMINARIES

Before getting into the meat of the language, here are a couple of concepts that don't really fit elsewhere.

Comments

Comments come in two forms. First, everything from `//` to the end of the line is ignored:

```
n = n+1; // add one to n
```

⁶ There is a whole underlying Python API for that whose description is outside the scope of this document.

Second, everything between `/*` and `*/` is ignored:

```
n /* the new value */ = n /* the old value */ + 1;
```

The second form of comment can span multiple lines:

```
/*  
 * Add one to n  
 */  
n = n + 1;
```

Note that the second form does not nest. It ends at the first `*/` it sees.

Variables Variables start with a letter or an underscore and contain letters, numbers, and underscores, optionally ending with a question mark. If a variable name begins with an underscore, it must be followed by a letter, number, or underscore.

These are all valid variable names:

```
a  
test  
n_2  
_private  
is_waste?
```

The only exception is that generally things that could have been variable names but are used as keywords are not recognized as names. Most of the time there's no reason they couldn't be (i.e., it wouldn't make things ambiguous), and I plan on making most of them usable as names, but for now only some of the short keywords (e.g., `s`, `ms`, `x`, and `y`) are carved out as exceptions. If the parser complains, choose another name.

Variables can also be a type followed (as a separate token) by a positive integer constant:⁷

```
drop 2  
pad 7  
direction 5
```

Within a macro that declares a parameter as `drop` or `a drop`, the parameter can be referred to as `drop` or `the drop`. Such parameters cannot be assigned to.⁸

Variables are strongly typed. If the type is not explicitly given when the variable is declared (or carried in the variable name), it will take the type of the expression used to initialize it.

Scoping Variables are statically scoped (as they are in languages like C++ and Java). The top-level (global) scope contains names defined at the top level of imported files and in the UI's text box. Within a macro definition, the parameters form a nested scope, and each nested block starts its own scope.

⁷ Within error messages, these will print as something like `***DROP_2***`.

⁸ This may change.

Variables declared within nested scopes may *shadow* those with the same name in outer scopes. Such shadowing variables have no relation to the shadowed variable and may have different types. Note that a variable is considered declared once its initializer has finished being evaluated, so that in

```
test = macro(int n) {
  print "param:", n;
  {
    n = n+1;
    print "still the param:", n;
    float n = 2*n;
    print "local:", n;
    n=n+1;
    print "local again:", n;
  }
  print "back to the param:", n
}
```

we would expect `test(5)` to print

```
param: 5
still the param: 6
local: 12.0
local again: 13.0
back to the param: 6
```

Within the block, `n` starts out referring to the parameter. The shadowing `n` is initialized based on the parameter `n` and prints as a floating-point number, but as soon as the block ends, `n` reverts to referring to the parameter.

As a special case, since declarations for numbered variables can look like their assignments, if such a form is found in a scope in which a variable with that name is already known (possibly from an enclosing scope), it is taken as an assignment rather than a new declaration. For example, in

```
pad 2 = drop 2 + 3 north;
if pad 2 has a drop {
  pad 2 = pad 2 + 2 north;
}
```

the form in the if-statement's block is read as an assignment to the variable declared outside the block. Since, presumably, there was no `pad 2` variable declared before the first line, that line will be read as a declaration rather than an assignment.⁹

If a shadowing variable with that name is desired, it can be signaled by adding the optional `local` keyword:

```
pad 2 = drop 2 + 3 north;
if pad 2 has a drop {
  local pad 2 = pad 2 + 2 north;
  // Do other stuff
}
```

This makes it clear that the line is a declaration of a new variable rather than an assignment to an existing one. The new variable will be in scope until the end of the block, at which point `pad 2` will revert to referring to the outer variable.

⁹ Hopefully, this behavior, while tricky to explain, is what a programmer would actually expect.

If a variable is used before it is declared, the compilation will fail. If it is assigned before it is declared, this is taken as an untyped declaration if the expression being compiled is at the top level of a file or typed in to the UI. In other scopes, it will also be taken as a declaration for backwards compatibility purposes, but this is deprecated¹⁰ and a warning will be printed.

One (hopefully temporary) consequence of variables needing to be declared before being used (combined with macros being functional objects bound to names) is that it's currently tricky to declare recursive (or mutually recursive) macros. Consider

```
fib = macro(int n) {  
  if n == 0 {  
    0;  
  } else if n == 1 {  
    1;  
  } else {  
    fib(n-1) + fib(n-2);  
  }  
};
```

The name `fib` isn't declared until the assignment, so within the macro, it's undefined. This can be kludged around by adding a definition like

```
fib = macro(int n) { 0; };
```

before the real definition. This tells the interpreter that `fib` is defined (as a macro taking an integer and returning an integer) at the global scope, and the real definition's assignment overwrites the dummy one. Note that in the real definition, the last line simply says, "Use the definition of `fib` in the global scope", which will be the real one when the function is called.¹¹

Order of operations

Arguments to operators and parameters to macro calls are evaluated from left to right, with each operator's evaluation complete before the next one starts.¹²

For example, in

```
(x=2) * (x=x+1)
```

the answer is guaranteed to be six, and `x` is guaranteed to contain 3 after it is done.

The only exception to this rule is for the `if...else` operator, in which the middle (test) argument is evaluated first and exactly one of the other two expressions is evaluated based on the test's value.

Note that when parallel blocks are used, the statements they contain "race" each other, so in

```
int x = 1;  
[[  
  x = 2;  
  fn(x);  
]]
```

¹⁰ And, therefore, can be expected to go away in the not too distant future.

¹¹ There will be a better solution for this in the future.

¹² The compiler may relax this when it can prove that evaluating them in another order wouldn't affect anything.

the argument to `fn` could be either 1 or 2, depending on whether or not the assignment in the first statement has happened yet when the argument to the function call is evaluated. Since the parallel block does not end until all of the parallel statements are done, the value after the example is guaranteed to be 2.

EXPRESSIONS

For the most part, expressions will be described along with the types they apply to. For example, arithmetic expressions on numbers and time values, as well as addition of deltas to pads. Here we will touch on some of the more general concepts.

Equality and inequality All values of the same type (or compatible types) can be compared for equality (`==`) and inequality (`!=`), which return boolean values:

```
x == y
x != y
```

Drops, pads, and wells are compared by identity. Other values are compared by content. For example, any two expressions evaluating to a volume of 2 uL will compare as equal, as will any two expressions evaluating to a delta of 3 down.

Attribute values As mentioned above, attributes are accessed by using the `'s` operator. For example, a pad `p`'s row can be obtained by `p's row`.

In some cases, the object may or may not contain a value specified by an attribute. For example, a pad may or may not contain a drop. If you ask for `p's drop` and pad `p` doesn't have a drop, an error will be signaled. To guard against that possibility, you can say, e.g.,

```
if p has a drop {
    p's drop : up 2;
}
```

The `has a` (or `has an`) expression returns true or false depending on whether the object has a value for the attribute in question.

Assignment expressions Variables are assigned values (after their declarations) by expressions of the form

```
var = expressions
```

Note the difference between the assignment operator `=` and the equality operator `==`:

```
x = 5      // assign 5 to x
x == 5     // Does x equal 5?
```

The value of the assignment expression is the value assigned to the variable, and the type of the assignment expression is the type of the variable, not the type of the expression. So, in

```
float f;
print (f = 2)+1;
```

the new value of `f` will be float-point 2.0 and what will be printed will be 3.0, not 3.

As can be seen in the above example, assignments can appear as parts of larger expressions. The assignment takes place immediately, so

```
int x = 0;
print (x=1)+x
```

what will print will be 2.

Some attributes (mainly those having to do with drops and wells) can be assigned to, e.g.,

d's volume = 1 uL;

Note that the only attributes that can be assigned to are those having to do with the *model*. Attributes such as a pad's *state* can only be modified by asking the system to communicate with the board (in this case, by saying, e.g., turn on).

Conditional expressions To choose one value or another based on a third, the if...else expression can be used:

x if x > y else y

The middle expression is evaluated first, and based on its value, either the first or last expression is executed. Note that the other expression is not evaluated.

The test expression must evaluate to a boolean value, and the two value expressions must evaluate to compatible types. (For example, one could be an integer and the other a floating-point number.) If they are compatible, but not identical, the value of the expression will be the most specific type that is compatible with both values.

Function calls Function calls, comprise a “functor” and a comma-separated list of parameters in parentheses:

```
fn(1, d, x+2)
(up 2: down 3)(d1)
(macro1 if test1 else macro2)(d2, 3 north, pad1+right)
round(compute(d1, 2, down))
```

If the functor is a built-in function (e.g., floor or str), it must be a single name.¹³ Otherwise it should be an expression (including, of course, a single name, referring to a variable bound to one of these), returning one of the following (see below for explanation):

- a macro,
- a delta or direction
- a drop motion specification such as to column 5

There is no provision (yet) for optional or named parameters, so the number and types of the parameters must match that of the function. (Except that, e.g., you can pass in an integer to a function that expects a float or a drop to a function that expects a pad).

Injectons A function that takes a single argument may be called using a simpler “injection” syntax, with the argument being “injected” separated from the function by a colon.¹⁴

```
d : to row (n + d's row)
n : squared
```

¹³ Built-in functions can be polymorphic, and this simplifies the type checking at compile time. It will probably be relaxed in the future.

¹⁴ The computer scientist in me kind of wants this to be a vertical bar (“|”) like the Unix pipe operator, but I think the colon will probably be more understandable.

Note that unlike the full function call syntax, the function doesn't have to be bound to a name. What happens is that first, the argument is computed, then the function is computed, and finally, the function is called with the argument as its only parameter.

The value of an injection is the value of the function call, unless the function doesn't return an argument (e.g., a macro whose block is a parallel block, an if statement whose blocks don't evaluate to the same type, or a sequential block whose last form doesn't have a value). In this case, the value of the injection is the value of the argument.

Since the colon operator is left associative, injections can be chained:

```
d : left 2 : up 5 : pause 2 seconds : right 5
w : dispense : right 3
```

In the last form, **w** is a well and **dispense** is a macro that dispenses a drop and returns it. The dispensed drop (the result of **w : dispense**) is then walked three pads to the right and returned as the value of the entire injection chain.

When the left-hand side of the colon doesn't return a value that can be passed to the callable value on the right but is itself a callable form, the behavior is a bit different (but hopefully intuitive). In this case, the colon returns a value which is itself a callable function which, when called, passes in its arguments to the left-hand function and then the result (or pass-through argument) to the right-hand function. For example

```
walk_path = left 2 : up 5
```

makes **walk_path** into a value that can be used as the injection target for a drop:

```
d : walk_path
```

The left-hand side can take more (or less) than one argument, in which case, the full function call syntax must be used on the result. For example, if **rectangle** is a macro that takes a drop and two deltas describing the first two sides of a rectangular path,

```
rectangle_plus = rectangle : 2 down
```

is a path that walks the rectangle and then continues on downward for two pads. It would be called like

```
rectangle_plus(d, 3 right, 2 up)
```

TYPES

In this section, I'll go over the various types of data that the macro language knows about. For each type, I'll detail

- the name of the type (e.g., in macro parameter declarations),
- how you create a value of the type,
- what sorts of expressions you can perform on it, and
- what attributes it has.

Numbers

Numbers come in two flavors: integers (**int**) and floating-point numbers (**float**). Integers can be used any place that expects floating-point numbers.

Integer constants consist of a digit followed by zero or more digits or underscores:

```
0
52
5_000
```

A floating-point number is an integer followed by an exponent (e.g., **e-5**) or by a decimal point, an optional integer, and, optionally, an exponent:

```
3.0
0.5e5
1e7
2.
12.123_456
```

Note that a leading digit is required. **.5** (point five) is not a legal floating-point number.

Numbers can be added (+), subtracted (-), multiplied (*) and divided (/). For addition, subtraction, and multiplication, if both arguments are integers, the result is an integer, otherwise it is a floating-point number. The result of division is always a floating-point number, even if the arguments are both integers. The negation (prefixed **-**) of a number is of the same type as the number.

To convert a floating-point number to an integer, use one of

```
round(n) // rounds n to the nearest integer
floor(n) // returns the largest integer less than or equal to n
ceil(i)  // returns the smallest integer greater than or equal to n
```

Numbers may be compared using **<**, **<=**, **>**, and **>=**. These operators return boolean values.

Boolean values Boolean (**bool**) values represent truth or falsity. Constant truth may be specified as **True**, **true**, **TRUE**, **Yes**, **yes**, or **YES**. Similarly, constant falsity may be specified as **False**, **false**, **FALSE**, **No**, **no**, or **NO**.

If **b1** and **b2** are boolean values, **not b1** returns the opposite of **b1**'s value, **b1 and b2** returns true if both **b1** and **b2** are true, false otherwise, and **b1 or b2** returns false if both **b1** and **b2** are false, true otherwise. Note that **and** and **or** are “short circuiting” operators. If the interpreter can tell based on the value of **b1** what the value of the entire expression will be, **b2** is not evaluated.

Strings Strings (**string**) are delimited by double quotes:

“This is a string.”

They may not contain newlines or carriage returns. Escape sequences are prefixed by backslashes:

| | |
|---------------------|---|
| <code>\t</code> | tab |
| <code>\r</code> | carriage return |
| <code>\n</code> | newline |
| <code>\"</code> | double quote |
| <code>\\</code> | backslash |
| <code>\uHHHH</code> | Unicode character with four-digit hexadecimal code. Letters can be upper- or lowercase. |

Some examples:

```
"Multi\inline\nstring"
"String\twith\ttabs"
"lowercase mu: \"\u00B5\""
```

Any value can be converted to a string by calling the `str` function:

```
d = 0.7 uL of reagent "R2" @ (12,3);
s = str(d);
```

This will set `s` to a string with contents

```
Drop[Pad(12,3), 0.7 µl of R2]
```

Strings are concatenated by adding (+) them together. A number can also be added to a string. The length of the string (i.e., the number of characters it contains) is

```
s's length
```

Physical quantities

Rather than using bare numbers for physical quantities such as times and volumes, the macro language treats these as separate types and requires that units be specified, as in

```
1 uL
2.5 seconds
(n+1) drops
d's volume + 0.5 uL
```

The magnitude of a physical quantity is a floating-point number.

Physical quantities of like dimensions can be added to one another, subtracted from one another, and compared with one another. They can also be multiplied and divided by floating-point numbers.

To extract the magnitude of a physical quantity as a number, you need to specify the desired units:

```
v's magnitude in uL
```

Similarly, to convert a physical quantity to a string, you need to specify the desired units:

```
v as a string in mL // the 'a' is optional
```

Directions

Directions (**direction** or **dir**) are represented by the constants `up`, `down`, `left`, `right`, `north`, `south`, `east`, and `west`.

Given a direction `d`, you can find related directions by using

```
d turned right
d turned clockwise // the same
d turned left
d turned counterclockwise // the same
d turned around
```

A direction can be used whenever a delta (see below) is required and is transformed into a delta of one step in the given direction.

Deltas

Deltas (**delta**) represent a number of steps (an integer) in a particular direction. The number of steps for a delta `d` can be found as

```
d's distance
```


and the direction can be found as

```
d's dir
d's direction
```

Deltas in a constant direction are formed by, e.g.,

```
5 up
right 3
```

The direction can come either first or last, but there is a subtle distinction¹⁵—the form with the distance first binds tighter than arithmetic operators, while the form with the direction first binds weaker. So

```
right n + 2
```

means $n+2$ steps to the right, while

```
p + 2 up
```

means 2 steps up from pad p. (That is, it adds the delta 2 up to the pad value p.)

As an alternative, you can use `rows` as a substitute for `up` in the distance-first form:

```
p + 2 rows
```

and, similarly, `columns` (or `cols`) as a substitute for `right`. When the distance is the constant 1, the keywords `row`, `column`, or `col` may (but need not) be used.

When the direction is computed at runtime, the `in direction` (or `in dir`) operator is used, e.g.,

```
2 in direction w's exit dir
```

As with directions, deltas can be turned:

```
d turned right // or left, clockwise, counterclockwise, around
```

The result keeps the same number of steps in the turned direction.

When a direction is used as a function (e.g., as an injection target), it takes a drop, walks the drop the given number of steps in the given direction, and returns the drop.

Note that when walking a drop, the underlying engine delays any step that would bring the drop next to or on top of another modeled drop.¹⁶ If you want to do the walk even if it means getting too close to another drop, use

```
drop : unsafe_walk(delta)
```

¹⁵ It represents my intuitions. It may be a bit too subtle.

¹⁶ It's actually a bit more complicated than that to do the right thing (most of the time) when moving drops want to cross paths. Essentially, a motion request waits until it can reserve the pad it wants to walk to, and it can only reserve the pad if neither the pad nor any of its other neighbors is occupied or reserved (indicating that some other drop is planning on moving there on the next clock tick).

Volumes, A volume (**volume**) is a physical quantity specified with units

uL, ul, microliter, microlitre, microliters, microlitres
mL, ml, milliliter, millilitre, milliliters, millilitres
drop, drops

A drop (in this context) is the basic volume dispensed by the board.¹⁷

Reagents A reagent (**reagent**) represents a liquid that can be contained in a drop or well. There are two predefined reagents, specifiable as

the unknown reagent // or "unknown reagent" or simply "unknown"
the waste reagent // or "waste reagent" or simply "waste"

Other reagents are modeled by giving a name (a string):

reagent "r1"
reagent named "r1"
the reagent "r1"
the reagent named "r1"
a reagent named "r1"
a reagent named "r1"

Two reagents with the same name are considered to be the same reagent. Note that this implies that all **unknown** reagents are the same, as are all **waste** reagents.

Reagents can also be computed as the combination of other reagents:

$r4 = \text{mixture}(2*r1, r2, 3*r3)$

r4 is now a reagent that represents two parts **r1**, one part **r2**, and three parts **r3**. It will print as

2 r1 + 1 r2 + 3 r3.

Mixtures can currently be made of up to eight reagents. Note that if any of the reagents in the mixture is the **waste** reagent, the result will be the **waste** reagent.

Adding two (possibly scaled) reagents together

$r = r1 + 2*r2$

is equivalent to

$r = \text{mixture}(r1, 2*r2)$

Note that **r1+r2+r3**, is equivalent to

$\text{mixture}(\text{mixture}(r1, r2), r3)$

and will therefore be 1 r1 + 1 r2 + 2 r3, not 1 r1 + 1 r2 + 1 r3.

¹⁷ Currently assumed to be 0.5 μL for Joey/Wombat boards, although this may change as we learn more about them.

Liquids

A liquid¹⁸ (**liquid**) is a combination of a volume and a reagent. Values are created by using the **of** operator:

1 drop of r1
2 uL of the unknown reagent
20 drops of (r1+r2)

Liquids can be added together. The result is a liquid with the combined volumes of the two arguments and a reagent that is a mixture of the two reagents scaled by their respective volumes. For example

2 drops of r1 + 1 drop of r2

is

1.5 µl of 2 r1 + 1 r2

Unlike with reagents, more than two liquids can be added together without concern.

If a liquid is divided by a floating-point number, the result is a new liquid with the same reagent and the volume divided by the number.

The volume and reagent of a liquid **liq** can be obtained by

liq's volume
liq's reagent

Pads

Pads (**pad**) represent spaces in the electrode grid on the board proper (i.e., not in the wells). Pads are characterized by a column (x coordinate) and row (y coordinate), both integers. The lower left-hand pad in the grid is coordinate (0,0), and the numbers increase upward and to the right.

A pad can be specified by a comma-separated pair of integer expressions in parentheses:

(2, 5)
(n, x+2)

It's row and column (as integers) can be obtained by

p's row
p's y coord
p's y coordinate
p's column
p's col
p's x coord
p's x coordinate

A pad may or may not currently have a drop on it. If it does, the drop can be obtained as

p's drop

To test whether there is a drop there, use **p has a drop**.

¹⁸ There's probably a better name for this. Maybe "sample" or "quantity". I can't use "volume" here.

A pad may or may not be the exit pad for a well. If it is, the well can be obtained as

p's well

To test whether the pad is an exit pad, use **p has a well**.

A delta may be added to or subtracted from a pad to get another pad the specified number of steps in the specified direction, for example

p + 2 right
p + (2*n) down
p - 2 rows

Note that there is no guarantee that the resulting pad is actually on the board or a pad that can be controlled if it is, and there is (as yet) no way to test.

Electrodes Pads (along with well pads, described below) are a subtype of “electroeds” (**electrode**). The primary feature of a component is that it has an electrode, and you can ask for the electrode's current state by¹⁹

p's state

Its electrode state may be modified by injecting the pad into a special action function:

p : turn on // or p : on
p : turn off // or p : off
p : toggle state // or p : toggle

Drops Drops (**drop**) represent physical drops on the pads of the board.²⁰ A drop has a current pad, which can be obtained by

d's pad

or simply by using **d**, as drops can be provided anywhere that pads are expected. This means, that you can say, for example,

d + 1 right : turn on

to turn on the pad one pad to the right of the drop's current pad.

A drop's *modeled* pad can be set by

d's pad = (2,5)

Note that this only changes where the system thinks the drop is. It doesn't actually control the board to move the drop to the desired pad. Aside from being occasionally needed if you determine from the display that a drop is being modeled in the wrong place, this is primarily useful to move a drop back onto the board after it has been removed in the process of a merge-and-split operation.

¹⁹ Unfortunately, it appears that you can't do anything with this at the moment, as there's no way to either test against a constant value or use it to set the state of another pad. Sigh.

²⁰ Or, more accurately, they represent *modeled* drops believed by the program to be on the board.

Drops also have associated contents (a liquid) and, via their contents, a reagent and volume:

```
d's contents
d's reagent
d's volume
```

All of these are settable.

The drop at a pad (if one exists) can be obtained by

```
p's drop
```

To add a new drop to the model, use the @ (or at) operator:

```
1 drop @ (2,3)
1.2 uL at p + 2 right
```

An error will be signaled if the pad already contains a drop. The left-hand side of the @ operator can take either a liquid or a volume. If a volume is given, the unknown reagent is assumed.

For backward compatibility,²¹ the form

```
drop @ p
```

can be used to obtain the modeled drop if it exists otherwise create one. It is essentially equivalent to

```
p's drop if p has a drop else 1 drop of unknown @ p
```

A drop's identity is maintained even as it moves, so after

```
d = 1 drop @ (2,3)
d : right 2
```

d's pad would now be (4,3) and

```
(4,3)'s drop == d
```

would be true.

As that last example illustrates, a drop can be injected into a delta, and this will cause the drop to move the specified number of steps in the specified direction. It can also be injected into a direction:

```
d : right
```

causing it to move one step in that direction.

A drop can also be asked to move to a specified row or column:

```
d : to row 5
d : to col 3 // or to column 3
```

²¹ Unless I get pushback, I'm going to declare this to be deprecated and remove it soon.

It can also be asked to move to a specified pad:

d : to p + 2 up

In this case it will first walk to the target row and then to the target column. That is, this is equivalent to

_p = p + 2 up
d : to row _p's row : to column _p's column

If a drop is no longer on the board (e.g., because it has entered a well), it can be removed from the model by saying

d : remove from the board
d : remove from board
d : remove

It is an error to try to use a drop that has been removed without first assigning its pad. This can be useful when modeling mixing and splitting, in which two drops merge and then split. See the `mix` macro below for an example of this.

Wells Wells (**well**) are identified as

well #2
well #(n+1)

The wells on the Joey board are numbered from 0 through 3 down the left-hand side and 4 through 7 down the right-hand side. On a Wombat board, only wells 2, 3, 6, and 7 are available.

Wells have the following attributes:

w's number
w's gate
w's exit pad
w's exit direction // or exit dir
w's contents
w's reagent
w's volume
w's capacity
w's remaining capacity

The **exit pad** is the pad on the grid immediately in front of the well. The **gate** is the well pad immediately adjacent to the exit pad. And the **exit direction** is the direction from the **gate** to the **exit pad**.

The contents, reagent, and volume can be set.

Well pads Well pads (**well pad**) represent the electrodes in the well. Like pads, they are components (see above) and have state which can be queried or modified. Unlike pads, they don't have coordinates, but they do know their well:

wp's well

A well's gate is obtained by

w's gate

Its interior pads are obtained by

w[n]

where **n** is an integer identifying the well pad. On the Joey board, well pads 0–2 are (from top to bottom) the well pads in the column adjacent to the gate. The next column contains well pads 3–5, and then the larger well pads 6, 7, and 8.

Note that the analogous interior pads on each side of the Joey board share their electrode state, so after saying **well #2[0] : on**, it will be the case that **well #3[0]** will also be on. Note further, however that these two well pads are not the same object, and they have different values for their **well** attributes.

Time, ticks, delays, and pauses

It is sometimes useful to be able to specify that execution pause for a certain amount of time, and that can be done by saying

```
pause 100ms;
pause 3 ticks;
```

As can be seen, the pause can be specified in elapsed time (**time**) or clock ticks (**ticks**). In the latter case, the pause is subject to the system clock being stopped, started, and stepped.

Time values are physical quantities, as described above. Time units are

```
s, sec, secs, second, seconds
ms, millisecond, milliseconds
```

Ticks literals are specified by an integer expression followed by **ticks** or **tick**. As with time values, ticks values can be added, subtracted, compared, and multiplied or divided by floating-point numbers. The number of ticks (an integer) can be obtained as

```
t's magnitude
```

Both time and ticks are subtypes of a more general delay type (**delay**) type, which is what pause expressions actually take and which can be specified as the type of a macro parameter.

Note that pause expressions mean different things when used as statements and when used in expressions. As a statement,

```
pause 3 ticks
```

means that the current thread of control should pause for three clock ticks. When used inside of an expression, it creates a functional object which takes one argument (of any type), pauses, and returns its argument. So, if you write

```
d : right 2 : pause 3 ticks : left 2
```

drop **d** will move two steps to the right, pause for three clock ticks, and then move back to its original position.

STATEMENTS

The set of statements in the macro language is rather impoverished (it doesn't for example, have any looping constructs), although it does have the notion of doing several things in parallel.

Note that aside from blocks, all statements end in semicolons.²²

²² If you forget the semicolon, the interpreter will probably be able to figure it out and forgive you, but it will print a warning.

Expression statements The most basic statement consists of an expression followed by a semicolon. This is primarily used in two situations.

First, an expression consisting of a function call or injection may be used for its side effects such as changing pad state or moving drops.

Second, an expression statement as the last statement in a sequential block becomes the value of that block.

Variable declarations Variable declarations consist of

1. the keyword `local` and/or a type name,
2. the name or number of the variable,
3. an optional initial value given as an equal sign followed by an expression, and
4. a final semicolon.

Some examples:

```
int n = 1;
pad p;
local x = 5.0;
drop 2 = pad 2's drop;
```

`n` is an integer variable whose initial value is 1. `p` is a pad-valued variable that is initially unassigned. Trying to use its value before assigning to it will result in an error. `x` is inferred to be a floating-point variable based on the type of its initialization function. Note that it is a compilation error to not specify either a type or an initialization function. That is,

```
local foo;            // illegal
```

will not compile, as the compiler wouldn't know what type to treat `foo` as.

The final form could be read as either a declaration of `drop 2` or an assignment to it. If a `drop 2` variable already exists in the scope, it will be read as an assignment, otherwise it will be a declaration. If you want a declaration to shadow an existing numbered variable, you have to add the `local` keyword:

```
local drop 2 = pad 2's drop;
```

Print statements Messages may be printed on the console²³ by means of the `print` keyword, which can be followed by any number of comma-separated argument:

```
print "Well number", n, "contains", (well #n)'s contents;
```

Note that a space will be printed between each of the printed values. If this is undesirable, use the `str` function to convert the arguments together and use string concatenation (i.e., addition) to combine the strings together.

Pause statements As mentioned above, the `pause` expression can be used as an expression:

```
pause 2 sec;
pause 3 ticks;
```

²³ At some point, I will probably want to provide a mechanism to print to the display.

When used as a statement in a sequential block, the next statement is not executed until the specified time has passed. Note that if the system clock is paused and the delay is specified in ticks, this will not happen until the clock is restarted or stepped the appropriate number of times.

When used in a parallel block, the entire block will not be considered to have terminated until (at least) the specified time has passed. If any of the other statements take longer, the block will end when the longest finishes.

Sequential blocks Sequential blocks are written as an opening brace {, a sequence of statements, and a closing brace }. The statements are executed one after the other, and the value of the block is the value of the last statement. For example,

```
{
  p = d's pad;
  d : right 2;
  p : toggle;
  d's y coordinate;
}
```

First d's pad is stored as p. This variable will be local to the block (unless p has already been established as a variable outside the block). Next d is walked two pads to the right. Then p's electrode is toggled. And finally, the y coordinate of d's current pad is established as the value of the block.

Parallel blocks Parallel blocks look like sequential blocks, except that instead of using braces, they use double square brackets [[and]].

In a parallel block, all of the statements are scheduled to happen at the same time. For those that involve modifications to the board, the first modifications in each statement will happen on the same clock tick. (Subsequent modifications do not necessarily line up.) The parallel block is considered completed when all of its statements are completed. A parallel block has no value.

Note that the contained statements race each other, so if two statements modify the same variable, unless care is taken, it will be impossible to predict which modification happens first. That is, in

```
[[
  x = 5;
  x = 6;
]]
```

after the block is done, x will be either five or six, but you can't predict which, and it might not be the same each time.

Parallel blocks are especially useful when the statements are electrode state changes. For example, in

```
[[
  w[0]: off; w[1]: off; w[2]: off;
  w[3]: off; w[4]: off; w[5]: off;
  w[6]: on; w[7]: on; w[8]: off;
  w's gate: off;
]]
```

All of well w's well pads are set to known values in a single clock tick. Similarly, in

```
[[
  d1 : right 4 : up 2;
  d2 : right 8 : down 2;
]]
```

two drops are walked, in parallel, along two separate paths. The block ends when the longer path is done.

Parallel blocks and sequential blocks are often combined, especially for things like dispensing sequences from wells:

```
{
  [[ w[3]: on; w[4]: on; w[5]: on; ]]
  [[ w[0]: on; w[1]: on; w[2]: on; ]]
  w's gate: on;
  w's exit pad: on;
  [[ w's gate: off;
    w[0]: off;
    w[1]: off;
    w[2]: off;
    w[3]: off;
    w[5]: off;
  ]]
  [[ w[0]: on; w[1]: on; w[2]: on; ]]
  w: ready;
  drop@w's exit pad;
}
```

This putative dispensing sequence is a sequential block containing eight steps. The first six, including four parallel blocks, modify the state of the well's pads, the seventh injects the well into a macro that gets it back to a "ready" state, and the final statement creates a drop at the well's exit pad and returns this drop as the value of the sequential block.

Blocks can also be combined the other way, with a set of sequential blocks performed in parallel.

If statements Conditional code is specified by means of if statements:

```
      if x < d's column {
        d : to column x;
      } else if y < d's row {
        d : to row y;
      } else {
        d : to p + 2 down;
      }
```

There can be any number of **else if** clauses (including zero), and the final **else** clause is optional.

Note that each of the clauses must be a block (not a statement). It can be a sequential block or a parallel block, and the block may only contain a single statement (or even no statement), but it must be a block.

If there is an **else** clause and all clauses evaluate to objects of the same type (or which have a common supertype, e.g., integer and floating point or drop and pad), the value of the if statement will be the value of the executed clause. Otherwise the if statement has no value.

INTERACTIVE FORMS In the text box, you can type the following:

- expressions (including assignments),
- declarations,
- print statements,
- sequential blocks, and
- parallel blocks

Note that the final semicolon for the non-block forms is optional here.

MACROS Macros are functional objects that can be applied to arguments. They are created by expressions consisting of

- the keyword **macro**,
- a header, which describes the types and, optionally, names of the parameters to the macro, and
- a body, which may be a (parallel or sequential) block or an expression.

The value of a call to a macro is the value of its body.

Note that macros are values. Typically, they will need to be assigned to a variable in order to be useful. Note in particular that this assignment will require a final semicolon.

Examples Some examples:

```

        abs = macro(int x)
        x if x >= 0 else -x;

manhattan = macro(pad 1, pad 2) {
    int rdiff = abs(pad 1's row - pad 2's row);
    int cdiff = abs(pad 1's col - pad 2's col);
    rdiff + cdiff;
};

ready = macro(well w)
[[
    w[0]: off; w[1]: off; w[2]: off;
    w[3]: off; w[4]: off; w[5]: off;
    w[6]: on; w[7]: on; w[8]: off;
    w's gate: off;
]];

col_first = macro(drop, pad) {
    drop : to col (the pad's col) : to row : (the pad's row);
}
    
```

The first defines a macro **abs**, which computes the absolute value of an integer. This is so simple, it's reasonable to use a single expression for its body, although it could also be done as a single-statement block. The single parameter is declared to be an integer and is named **x**.

The second computes the Manhattan distance between two pads, calling the **abs** macro. Its two parameters are both pads and they are declared to be referred to by number rather than by name.

The third macro encapsulates the parallel block example we saw above, which gets its well parameter (named **w**) into a ready state.

The final macro does a column-first walk to a target pad. Here, neither of the parameters are named. In the body they are simply referred to as “drop” and “the pad”. (Either form is acceptable.)

Closures Variables referenced within macros that are not specified as parameters or first assigned within their bodies are taken to refer to variables of the same name in the surrounding context at the time the macro is defined. In particular, when macros are defined within macros,²⁴ they can refer to local variables within the

²⁴ Yes, this is possible. No, you will probably not need this.

enclosing macro, and these bindings persist even after the enclosing macro has finished executing. So, you can do things like

```
adder = macro(int n) {
  macro(int n) {
    x + n;
  };
};

add_one = adder(1);
add_five = adder(5);
```

Every time `adder` is called, it returns a new macro which adds its value to the value that was passed in to `adder`. So, `add_one(7)` will return eight, and `add_five(7)` will return twelve.

It's even possible to maintain state this way:

```
counter = macro() {
  int n = 0;
  macro() {
    n = n+1;
    n;
  };
};

times_called = counter();
```

When `counter` is called to define `times_called`, creates a new variable `n`, which its value will read. The first time `times_called` is called, it will increment this `n` and return one. The second time, it will return two, etc.

As a more complex (and possibly more useful) example, consider a macro that mixes a drop with another drop two steps in a given direction. This can be done by

```
mix = macro(dir) {
  macro(drop 1) {
    local pad 1 = drop 1's pad;
    local pad 2 = pad 1 + 2 in direction dir;
    local drop 2 = pad 2's drop;
    [[
      drop 1: unsafe_walk(dir);
      pad 2: off;
    ]]
    drop 2: remove from board;
    drop 1's contents = drop 1's contents + drop 2's contents;
    [[
      drop 1: dir turned around;
      pad 2: on;
    ]]
    drop 2's pad = pad 2;
    drop 1's contents = drop 1's contents/2;
    drop 2's contents = drop 1's contents;
    drop 1;
  };
};
```

The whole thing can be summarized as

```
mix = macro(dir) {
```

```
macro(drop 1) { /* ... */;
};
```

mix is called with a direction (e.g., **mix(left)**), and it returns a macro that takes a drop and does the mixing, with the variable **dir** bound to this direction (**left**, in this case). The result of calling the **mix** macro can be used as an injection target, e.g.,

```
d : left 5 : mix(up) : down 3;
```

Looking at the macro that actually does the mixing, we have

```
macro(drop 1) {
  local pad 1 = drop 1's pad;
  local pad 2 = pad 1 + 2 in direction dir;
  local drop 2 = pad 2's drop;
  [[
    drop 1: unsafe_walk(dir);
    pad 2: off;
  ]]
  drop 2: remove from board;
  drop 1's contents = drop 1's contents + drop 2's contents;
  [[
    drop 1: dir turned around;
    pad 2: on;
  ]]
  drop 2's pad = pad 2;
  drop 1's contents = drop 1's contents/2;
  drop 2's contents = drop 1's contents;
  drop 1;
};
```

First, we remember **drop 1**'s pad as **pad 1**. (We use the **local** keyword to guard against accidentally modifying any global variable named **pad 2**.) Next, we declare **pad 2** to be the pad two steps in the remembered direction, where we expect there to be another drop, and we ask **pad 2** for its drop, which we call **drop 2**.

Next, in parallel, we walk **drop 1** to the pad between the two drops and turn off **pad 2**. We use **unsafe_walk**, because otherwise **drop 1** would wait until **drop 2** moved out of the way before actually moving.

The result of these two parallel operations is that **pad 1** and **pad 2** are both turned off, and the middle pad is turned on. This should cause both drops to move to the middle pad and combine. But the model only knows that **drop 1** moved, so in the next two steps, we clean things up:

```
drop 2: remove from board;
drop 1's contents = drop 1's contents + drop 2's contents;
```

drop 2 is modeled as no longer being on the board, and **drop 1** is modeled as now containing the mixture of the contents of the two drops.

Next, in parallel, we walk **drop1** back to its original position and turn **pad2** back on. On the real board, this will cause the big drop in the middle to split, but it requires some clean-up to convey this to the model:

```
drop 2's pad = pad 2;
drop 1's contents = drop 1's contents/2;
drop 2's contents = drop 1's contents;
```

drop 2 is now back on the board, and both drops contents are set to half the volume of the larger drop's contents.

Finally, the original drop is returned as the value of the macro.