Introduction to DML

Straightforward specification of protocols for digital microfluidics boards

Evan Kirshenbaum

**Introduction to DML** © 2023 by Evan Kirshenbaum

**All rights reserved**

No part of this book may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

ISBN: 000-000-000000000

**Free book template downloaded from:**

https://usedtotech.com

admin@usedtotech.com

Contents

[Chapter One: Introduction 1](#_Toc144326033)

[Digital Microfluidics Context 1](#_Toc144326034)

[Guiding Principles 1](#_Toc144326035)

[The customer is not a programmer 2](#_Toc144326036)

[The customer’s informal description is the gold standard 2](#_Toc144326037)

[Physical quantities are not numbers 2](#_Toc144326038)

[English is better than Computerese 3](#_Toc144326039)

[Don’t worry about mistakes users won’t make 3](#_Toc144326040)

[The principle of least astonishment 4](#_Toc144326041)

[It's easier to learn a few new things that a lot of them 4](#_Toc144326042)

[There’s more than one way to do it 5](#_Toc144326043)

[Easy things should be easy, and hard things should be possible 5](#_Toc144326044)

[This isn’t the last version of the language 6](#_Toc144326045)

[Trade complexity below for simplicity above 6](#_Toc144326046)

[Rough Spots and Missing Pieces 6](#_Toc144326047)

[Chapter Two: A Tutorial Example 9](#_Toc144326048)

[The protocol 9](#_Toc144326049)

[The board 10](#_Toc144326050)

[The code 11](#_Toc144326051)

[Comments 11](#_Toc144326052)

[Names 12](#_Toc144326053)

[Function definitions 12](#_Toc144326054)

[Whitespace 12](#_Toc144326055)

[Step 0: Setting things up 13](#_Toc144326056)

[Variable declarations 13](#_Toc144326057)

[Reagents 14](#_Toc144326058)

[Numbered objects 14](#_Toc144326059)

[Numeric types and literals 14](#_Toc144326060)

[Injections 15](#_Toc144326061)

[Liquids 16](#_Toc144326062)

[Physical quantities 16](#_Toc144326063)

[require expressions 17](#_Toc144326064)

[Function calls 17](#_Toc144326065)

[The waste reagent 18](#_Toc144326066)

[Attributes 18](#_Toc144326067)

[Assignments 19](#_Toc144326068)

[Step 1: Mixing the first drop 19](#_Toc144326069)

[Pads 20](#_Toc144326070)

[Drops 20](#_Toc144326071)

[Dispensing drops 21](#_Toc144326072)

[Moving drops 21](#_Toc144326073)

[Directions 21](#_Toc144326074)

[Merging drops 21](#_Toc144326075)

[Simultaneous execution 23](#_Toc144326076)

[Time 24](#_Toc144326077)

[Declaring function parameters 24](#_Toc144326078)

[Unnamed parameters 25](#_Toc144326079)

[Declaring function return types 25](#_Toc144326080)

[Return statements 25](#_Toc144326081)

[Paths 25](#_Toc144326082)

[Compound injections 26](#_Toc144326083)

[Loops 26](#_Toc144326084)

[exit statements 28](#_Toc144326085)

[Named loops 28](#_Toc144326086)

[Step 2: Mixing with the other two reagents 29](#_Toc144326087)

[Injectable parameters 31](#_Toc144326088)

[Splitting drops 32](#_Toc144326089)

[(eventual) variables 32](#_Toc144326090)

[Step 3: Heating the drops 33](#_Toc144326091)

[(maybe) values 34](#_Toc144326092)

[Temperature literals 35](#_Toc144326093)

[Actions at statement level 35](#_Toc144326094)

[Heaters 36](#_Toc144326095)

[Testing properties (is expressions) 37](#_Toc144326096)

[wait expressions 37](#_Toc144326097)

[print expressions 37](#_Toc144326098)

[String literals 38](#_Toc144326099)

[Formatted quantities 38](#_Toc144326100)

[String concatenations 38](#_Toc144326101)

[Parentheses for grouping 38](#_Toc144326102)

[print expressions as injection targets 39](#_Toc144326103)

[Timestamps 40](#_Toc144326104)

[Conditional expressions 41](#_Toc144326105)

[Binary components and states 41](#_Toc144326106)

[Turning heaters on and off 41](#_Toc144326107)

[Step 4: Analyzing the drops 42](#_Toc144326108)

[Extraction ports 43](#_Toc144326109)

[Dealing with deadlock 44](#_Toc144326110)

[prompt expressions 44](#_Toc144326111)

[Local functions 44](#_Toc144326112)

[The board 46](#_Toc144326113)

[Sensors 46](#_Toc144326114)

[Writing readings to files 47](#_Toc144326115)

[local variable declarations 47](#_Toc144326116)

[Sample types 48](#_Toc144326117)

[Conditional statements 49](#_Toc144326118)

[Changing a drop’s reagent 50](#_Toc144326119)

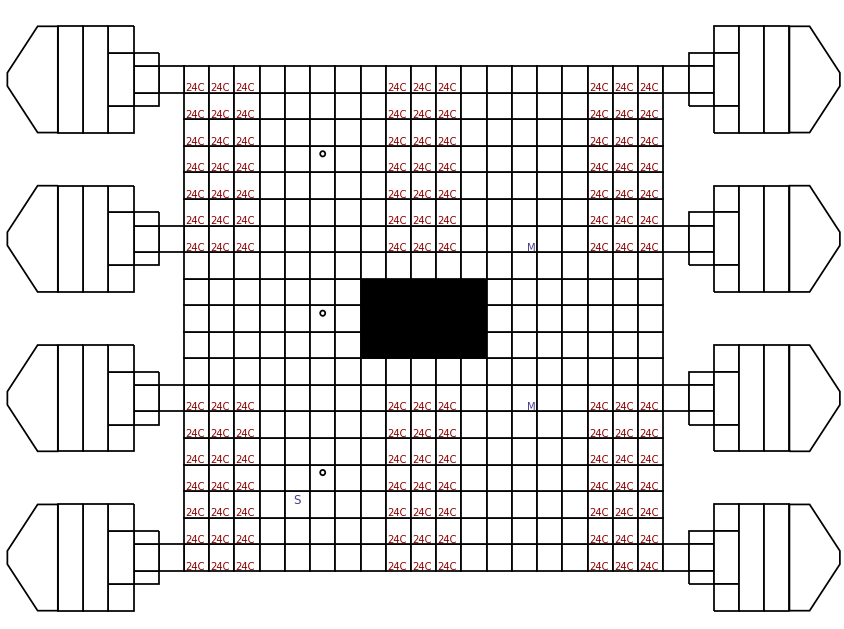
[Entering a well 50](#_Toc144326120)

[Putting it all together 51](#_Toc144326121)

# Introduction

**DMF**, short for “Digital Microfluidics Language” is a language specifically designed for describing experiments and protocols to be run a digital microfluidics boards.

## Digital Microfluidics Context



**Well**

**Heating Zone**

**Well Pad**

**Gate**

**Exit Pad**

**Sensor Target**

**Extraction Port**

**Magnet**

**Dead Zone**

## Guiding Principles

There are a number of principles that I kept in mind when designing DML. Some of these principles occasionally conflicted with one another, forcing me to make tradeoffs, but on the whole, they reinforced one another.

A warning to the reader, though. Most of these principles are designed to make the language *unsurprising*, but unsurprising is not necessarily *simple*, and it certainly isn’t always simple to describe. When I go into detail below about how the language actually works, you may find your eyes glazing over. That’s okay. In most cases—if I did my job right—you should be able to look at the examples and say “Yeah, that makes sense.”

### The customer is not a programmer

This principle can also be stated as “I am not the client,” and it can be a hard one to internalize. I wrote my first program in 1978, drew my first paycheck as a programmer in 1980, and designed my first programming language in about 1985. Programming makes sense to me at a level that it doesn’t to most people, even most computer scientists.

But when designing a language for other people to use, that doesn’t matter. What matters is that it is easy to use for the people who will use it. In this case, that means scientists and engineers, especially those who need to use digital microfluidics boards, and these people cannot be expected to have more than a very basic exposure to writing their own programs in a traditional programming language like C++, Java, or even Python. But they do know what they would like their programs to do, and making the transition from their desired behavior to (correct) DML code simple is the ultimate goal.

### The customer’s informal description is the gold standard

In almost all cases, the customer already has a way of informally writing down (or drawing or talking through) a description of what they want their program to do. In a perfect world, that would be enough. The world we live in is far from perfect, but to the extent that we can come closer to that informal description in a program whose behavior mimics what is desired, we have a better language.

If the programmer wants to talk about “well number 1” on their board, it makes little sense to make them write board.wells()[1], when you can just let them write well #1. If they want to transfer in 0.5 µL of a mixture of reagents A and B in a 2:1 ratio, why not let them write 0.5uL of 2\*A+B?

### Physical quantities are not numbers

Anybody who’s known me for any length of time knows that this is one of my biggest pet peeves. Too often, code that deals with real-word quantities like volume, voltage, or even time has them represented as floating point numbers with — if you’re very lucky — a comment telling the programmer that this functions parameter should represent a voltage in millivolts, but nothing stops the caller from accidentally passing in a value in volts or from a programmer from misspelling a variable and winding up adding a time in milliseconds to a distance in meters.

Every programmer has screwed up multiple times misremembering whether a sleep() function takes its number in seconds or milliseconds. Programs have crashed … and so have spacecraft. People have gotten the wrong radiation therapy dose and died.

This was an embarrassment thirty years ago; it’s inexcusable today. Physical quantities are very important to scientists and engineers, and it’s crucial that DML takes away the ability to make stupid mistakes. It doesn’t matter if you said that t was 20 seconds. It’s just a time, and you can add it to 3 hours or compare it against 0.5 minutes without worry. And you can’t add it to a voltage or ask whether it’s less than 1mL. The compiler will catch those errors when the program is compiled.

### English is better than Computerese

The programmer’s informal description is almost certainly written largely in some form of a natural language, so approximating it will often mean approximating natural language. At this point, we can only go so far, but where we can, we should. Taking English — the modern scientific lingua franca and the language I know best — as a model, this means doing things like allowing multi-word names when possible,[[1]](#footnote-1) so a well can have an associated exit direction rather than an exit\_dir. It means that when asking for a well’s exit direction, instead of saying w.exit\_dir, you can say the well’s exit direction. Often this will mean that a program will contain more characters, but the increase in clarity can more than offset it.

### Don’t worry about mistakes users won’t make

In DML, you can say that a parameter to a function is a drop. You can also say that it is an integer. So how much complexity does the compiler have to point out to the programmer that they accidentally said an drop? Zero. It’s an error that an English speaker is unlikely to make, and if they do make it, the default behavior of not distinguishing between the two is highly unlikely to cause problems. So we don’t worry about it. This sort of thing comes up all the time as a language gets closer to a natural language. What’s important is that you do the right thing with the input you’re likely to get and that you don’t do something wrong if the user makes a simple mistake of the sort that users are likely to make. But errors you’re unlikely to see aren’t worth spending a lot of time worrying about.

### The principle of least astonishment

This rule dates back at least to the 1960s. In language design it recognizes that often what a user expects a given bit of syntax to do in one context may well be different from what they expect the same syntax to do in another context, and that in this case, it’s far more important that the language just do what they expect in each than that it be consistent and surprise them in one. Similarly, if a strict adherence to simple rules means that code that obviously should do something is considered ill-formed, that’s a problem. The language should be changed so that it does what it obviously should, even if that means that a detailed explanation of when it does what may be complex. If the language designer has done their job, the programmer won’t ever notice most of the inconsistencies, because they match their expectations.

### It's easier to learn a few new things that a lot of them

While the customer may not be a programmer, and while we want to be close to their informal description, it’s okay if we introduce notation they’re not familiar with, especially when they’re being asked to describe things at a lower level than they’re used to. But when we do this, it’s important that we don’t do it too often, so when we do introduce something, we should use it as often as possible, so that it becomes second nature.

In DML, it’s common to have a drop go through a sequence of actions. Move it here, merge it with another drop, walk it around a bit, move it over here, split it, enter this well. This isn’t the level a protocol designer thinks at, so DML introduces a notion of *injection* (based on Unix shell pipes) in which an object is passed to a function, and the value is passed to another function, and so on. Once the user has gotten used to the weird (though hopefully evocative) o : X : Y syntax, though, we can exploit the Principle of Least Astonishment to present the user with right-hand-side values that aren’t functions, but whose behavior there is what they expect without thinking about it. If d is a drop, d : east moves the drop one pad to the east. If c is some component that can be turned on and off, c : off turns it off. If we’re in the middle of an injection chain moving a drop and we see print ײWe’re atײ, d’s pad, of course it’s going to print that message. If a target is wait for 10 seconds, that’s what we’ll do.

### There’s more than one way to do it

This is one of the guiding principles espoused by Larry Wall in his design of the programming language Perl in the late 1980s, and it ties in with several of the principles described above. Different people will have different ways of describing the same thing, and not only is that okay, but to the extent possible, you should support many of them.

In DML, a heater has a current temperature attribute. Or, if the user prefers, it can be temperature or temp or current temp. You can describe a variable’s type as (eventual) drop or eventual drop or an (eventual) drop or an eventual drop. You can repeat until a condition is true, and you can repeat while its opposite is true. You can introduce a function definition with define or def or macro or, depending on the definition, function, func, procedure, proc, or action. There are omittable “noise words” all over the place. You can pause the clock or simply pause clock. You can refer to the same direction as up or north, and you rotate a direction right or clockwise. You can ask whether the pad has a drop or doesn’t have a drop or whether the pad’s drop exists or is not missing or …

When coupled with the principle of least astonishment, this can mean that whatever the programmer tries is (hopefully) likely to do what they want. And if it doesn’t, it should in a later version.

### Easy things should be easy, and hard things should be possible

This is Larry Wall’s other dictum for Perl. In the design of DML, I’ve focused more on the first part, trying to prioritize things so that the things that programmers are going to need to do most of the time should be simple to do, and that it’s okay if the rest is ugly or cumbersome. Indeed it can sometimes be beneficial to say “This thing is dangerous, though sometimes necessary, and I’m going to implement it in such a way that you know you’re doing something dangerous”. (Of course, the next step is sitting back and figuring out *why* the user wants to do that and seeing whether there isn’t a way to support that directly at a higher (and safer) level.)

Where DML falls short at the moment is the second half of the slogan. There are a lot of things that I can see users wanting to do with the language that it simply doesn’t support yet. This is largely due to lack of time, and hopefully it will improve.

### This isn’t the last version of the language

All languages evolve over time. It’s okay if this version of the language doesn’t do everything that I want it to. The next version will be better. The syntax will be cleaner and less surprising. It will allow programmers to more simply write things that are hard today and to write things that are simply impossible today. The compiler will be more efficient, and the code will run faster.

A corollary of this, however, is that as soon as people start writing programs in a language, it becomes more difficult to make changes that will break their code, so you have to start worrying about backwards compatibility.

This, however, should not be overstated, especially at such an early stage of language design. When I was in college, I took a class in programming language design from Al Aho of Bell Labs. He recounted an anecdote about the development of the make utility in 1976, about a decade earlier, and in particular, the syntax of the files that drove it. According to him, Stuart Feldman threw together a little tool for a project he was working on, and other programmers there soon started using it. He realized that the syntax of the files was terrible, but it did the job for what he needed, and he was a busy man. By the time he had a chance to revisit the design some months later, he decided that it was too late to change it: *dozens* of people were using it. And a decade later, tens of thousands of us (at least) were struggling daily with the same awful syntax.

### Trade complexity below for simplicity above

In may systems there’s a tradeoff between the complexity of the interface presented to the user and the complexity of the system that implements the interface. In almost all cases, it’s worth making the invisible implementation more complex in order to make the user-visible portions simple to use.

## Rough Spots and Missing Pieces

While I’m happy with a lot of DML and its implementation, there is still a fair bit that I’m not happy with. In no particular order:

* The error messages from the compiler are *really* bad. Most of this has to do with the ANTLR4 tool used to do the parsing, but some of it is due to my not having come up with a coherent method of reporting errors from within the compiler proper. Code that compiles without error is fine, but if there’s a syntax error, it can be *very* difficult for the user to identify where the problem is.[[2]](#footnote-2) It is apparently possible for ANTLR4 parsers to provide better error messages with support from the compiler-writer (i.e., me), but I haven’t had the chance to put much effort into this.
* It’s unconscionable that there isn’t yet support for things like sequences or mappings and that there’s no way for users to define their own classes, attributes, and types (especially enumerated types).
* The syntax could (and should) be a lot more natural-language-like. My intention was to make things to that all names could have spaces in them and that there would be notions of things like adjectives and prepositions. There are lots of cases where parentheses are still required where they shouldn’t be. (For example, you have to say w : require(2uL of A), where you should simply be able to say w : require 2uL of A.) There are good reasons why I wasn’t able to get around them, but they should go away in the future.
* In particular, it would be very nice to have the ability to use “it” to refer to whatever it is we were just acting on. This would allow for a more verbose but clearer way of doing what we currently do with injection chains.
* I don’t like forcing non-programmers to use braces for blocks. In my initial PhysL design ideas, blocks were done with bullets and indentation, but it wasn’t clear that that would be (1) easily parsable with ANTLR4 and (2) easily typable by the user. This should be revisited.
* There really should be a clearer alternative to the current double-bracket parallel block notation.
* It’s currently impossible to write overloaded functions, and the lack of this ability makes lots of other things uglier in both the implementation and in user-visible portions. (For example, there’s a mechanism in there to support built-in overloaded functions, because they’re actually necessary, but they complicate things a great deal.) I have a partial implementation of this, but I couldn’t get it to work in time.
* There are a lot of places where the underlying Thylacine implementation bleeds through to the user in ways that they should be insulated from.
* Interaction with the user is very primitive. The program can prompt the user, putting up a dialog box, but this is essentially just a pause. There’s no way for the user to provide input that could guide the program.
* There’s no way to read or write files or communicate with remote sites.
* For Python-level Thylacine protocols, it was typically necessary to use things like barriers to do more sophisticated traffic control to prevent drops from deadlocking. This should be extended to DML.
* Similarly, Thylacine has a notion of multi-drop mixing and dilution that needs to be exposed in DML in as simple a way as 2-drop merging and mixing.
* Ideally, the language should take care of motion planning in to pad expressions so that the user doesn’t have to worry about avoiding dead pads and deadlocking with other drops. This may imply changes to the underlying Thylacine system in which drops actually know their targets so that the system can detect deadlock and figure out how to back out of it.
* DML currently can only talk about drops that sit on single pads on the board. The underlying Thylacine system is able to model *blobs* of liquid that may be more than a single drop in size. These should be supported directly in DML.
* It should be possible to do a lot more at a higher level. There’s no reason why the programmer can’t talk about high-level desires (mix this reagent with that one, heat it for this long at that temperature) and have the compiler figure out what to dispense from where and which heater to use. But that may be a higher-level language that compiles down to DML.
* There’s currently no way to write a parameter declaration for parameters that take functions. You can do it for variables by using local, but you can’t actually specify. Similarly, you can’t specify the return type for a function that returns a function.
* There needs to be a way to import one file from another (and from the interpreter), with a check to make sure that it isn’t imported twice and we don’t have a cycle.

# A Tutorial Example

In this chapter, we’ll look at a worked example. Note that this is not necessarily a *realistic* example, but it demonstrates many of the features of a language that a user is likely to need.

In this example, we’ll demonstrate dispensing drops, moving them around simultaneously on the board, merging them together and splitting them apart, heating them, using a sensor to take measurements on them, making decisions based on the measurements, and finally, moving drops into wells or extracting them via (manual or robotic) pipetting through an extraction port.

## The protocol

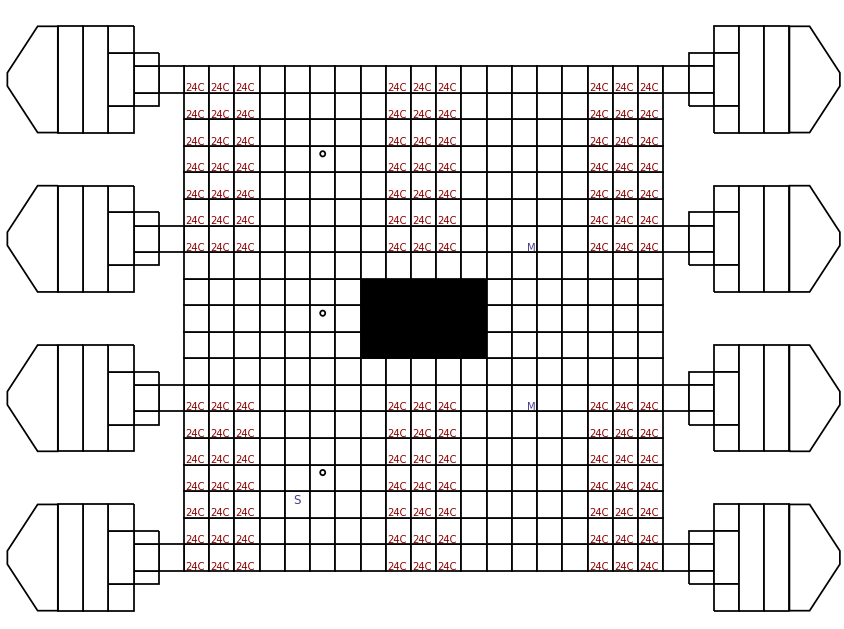
The protocol we’ll implement involves four reagents: A, B, C, and D. It is defined informally as

1. Dispense drops of reagents A and B and merge them together. Walk the merged drop around for 10 seconds to ensure mixing.
2. Split the merged drop in two and merge the resulting drops with a drop of reagents C and D respectively. Mix these drops by “shuttling” (stepping back and forth) 10 times.
3. Heat the combined drops at 80°C for 30 minutes.
4. Measure each heated drop using an ESELog sensor.
   1. If the mean voltage read on the E1D1 channel is greater than 70V, the drop passes and should be preserved for later testing.
   2. Otherwise, the drop fails and is waste.

## The board

The board to be used is the one shown above, repeated more cleanly here:

* The board contains a 19-by-19 array of pads, numbered from pad (1, 1) in the low left-hand corner to pad (19, 19) in the upper right-hand corner.



* There is a 5-by-3-pad dead region in the center of the board. These are pads that cannot be turned on and which, therefore, must be avoided when moving drops.
* The board contains eight wells, numbered 1 to 4 from top to bottom on the left-hand side of the board and 5 to 8 from top to bottom on the right-hand side.
  + We will be using wells 3, 4, 7, and 8 as input wells and well 5 as a waste (output) well.
* The board contains six heating zones, each controlled by a heater, in three columns of two zones each. The left-most column contains heater number 1 on top and 2 on the bottom, the middle column contains heaters number 3 and 4, and the right-most column contains heaters number 5 and 6.
  + We will be using the heating zone on the bottom in the middle, controlled by heater number 4.
* We have available an ESELog sensor, capable of making voltage readings along several channels. The sensor’s target is configured to be pad (5, 3), shown by an “S” on the figure.
* The board has three extraction ports, numbered from 1 to 3 from top to bottom.
  + We will be using the middle port, number 2, to extract successful combinations, potentially for further testing later.
* The board also has two magnets, numbered 1 and 2 from top to bottom.
  + We will not be using them.

## The code

Rather than overwhelm the reader with the entirety of the example at once, we’ll build it up step by step. The entire file, **example.dml**, will contain three informal sections:

1. At the top of the file, we will define the *parameters* of the protocol, things like the reagents to use, the well assignment for each reagent, the temperature to incubate at, and the time to walk while mixing. Putting these together at the top makes them easy to find and modify and means we can avoid using what computer scientists call “magic numbers” in the protocol itself.
2. Next we have the definitions for functions that the protocol will want to make use of.
3. Finally, we have the definition of an action called run\_protocol, which, when run, runs our protocol. The action will be written so that it can easily be run multiple times.

The overall structure of the file will be

In file example.dml:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Our first protocol

\*

\* Author: Evan Kirshenbaum

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Protocol parameters

// Helper functions

define run\_protocol {

// The body of the protocol

}

Comments

At the top of the file, we see the two types of comments available in DML. People familiar with programming languages like Java and C++ will be familiar with both of them:

1. *Block comments* are introduced by /\* (slash star) and contain all characters up to and including the next following \*/ (star slash), even if this runs over multiple lines. Note that these comments do not nest. That is, if you have code that includes block comments and try to “comment it out” by using a block comment, the new comment will end at the first star-slash it sees. For this reason, block comments are typically used for information blocks as shown above.
2. The more common *end-of-line comments* begin with // (slash slash) and run to the end of the line.

In either case, comments are completely ignored by the language.

Names

The definition of the protocol action:

define run\_protocol {

// The body of the protocol

}

is introduced by the keyword define followed by the name of action being defined, in this case “run\_protocol”. Names in DML can contain letters, digits, and underscores, but they may not begin with a digit and they may not consist solely of a single underscore. In addition, they may end with a single question mark (?). Names in DML are case sensitive, so run\_protocol is different from Run\_Protocol. There are a number of keywords that can’t be used as names.[[3]](#footnote-3) The compiler will complain—likely in a very cryptic way—if you try.

Function definitions

The define keyword is used to define functions (aka “macros”), which may take arguments and may return values when invoked. A function that takes no arguments and returns no value may be referred to as an *action*.

Following the action name, the body of the action is given as a sequence of *statements* between braces { … }. Several types of statements will be shown below. When the action is invoked, the statements will be executed in sequence.

Whitespace

One other thing to note up front is that *whitespace* — spaces, tabs, and line breaks are (almost) completely ignored. The only exceptions (that spring to mind) are (1) spaces and tabs within string literals are preserved (line breaks are illegal there) and (2) at least one whitespace character (or, I guess, comment) is required between two words if not having it would be read another way. As an example of the second, the define run\_protocol in the above example could not have been written as definerun\_protocol, as the compiler would see that as one long name.

### Step 0: Setting things up

The first thing the protocol will want to do is to ensure that the wells contain the appropriate reagents.

In file example.dml:

// Protocol parameters

reagent A = reagent "A";

reagent B = reagent "B";

reagent C = reagent "C";

reagent D = reagent "D";

well A\_well = well #3;

well B\_well = well #4;

well C\_well = well #7;

well D\_well = well #8;

well waste\_well = well #5;

define run\_protocol {

A\_well : require(1 drop of A);

B\_well : require(1 drop of B);

C\_well : require(1 drop of C);

D\_well : require(1 drop of D);

waste\_well’s reagent = waste;

// more to come

}

Variable declarations

In the parameters section, we declare several variables and give them values. Since these variables are declared at the top level of the file, they are considered to be *global variables* and can be referenced from anyplace in the file (unless there is another declaration of a variable with the same name that “shadows” one of them). Variable declarations all have the form

type name = initial-value;

where the initial value (and the preceding equal sign) may be omitted. (If an initial value is not given, you will get an error if you try to use the value of the variable before one has been assigned to it.) Variable declarations, like all statements that don’t end with { … } blocks, end with a semicolon (;).

Reagents

In the case of

reagent A = reagent "A";

the variable being declared is called A and its type is reagent. The reagent type refers to the kind of liquid contained in a drop or well, without respect to its quantity. The initial value expression says that the initial value of A is the reagent whose name is “A”. Note that if you use the expression reagent "A" multiple times, you will get the same reagent each time.

After declaring the reagents we will use, we declare the wells that we will use to store them. In

well A\_well = well #3;

we declare a variable named A\_well whose type is well and which refers to well number 3 on the board. Note that there is no inherent linkage between the reagent variable being A and the well variable being A\_well. It will be up to us to make the linkage explicit later on.

Numbered objects

The well # syntax (and, similarly, with heater #, extraction point #, and so on) is used to identify a particular well. The argument does not have to be a constant, but can be any integer value, so if we had a variable n holding an integer, we could refer to well #n or even well #(n+1). In this case, we are specifying that variable A\_well will refer to the third well from the top on the left-hand side of the board.

Numeric types and literals

Since we’ve seen our first literal number here (3), let’s take a minute to talk about numeric types and literals. Integers are elements of the int type. Integer literals are sequences of digits (0 through 9). Fore readability, they can also contain underscores, so you can write, e.g., 1\_000\_000 as equivalent to 1000000.[[4]](#footnote-4) Negative literals are specified by preceding them with a minus sign (-), as -27. Real numbers are elements of type real (also specifiable as float). An int can always be provided when a real is required.

Real literals include a decimal point (.) and must have at least one digit on each side of the decimal point. Real literals may also end with e (or E) followed by an integer (the *power*), which specifies that the number should be multiplied by ten raised to the power. For example, 2.0e3 is another way of writing 2\_000.0 and 1.5e-2 is equivalent to 0.015. If the power is given, the decimal point can be omitted, so 2.0e3 could just have been written as 2e3, but this is still 2000.0 (a real), not 2000 (an int).

Injections

Within the definition of run\_protocol, we start by saying which reagents are going to be associated with which wells and how much of each we’re going to need:

A\_well : require(1 drop of A);

B\_well : require(1 drop of B);

C\_well : require(1 drop of C);

D\_well : require(1 drop of D);

waste\_well’s reagent = waste;

The first four lines make use of a somewhat unusual bit of DML syntax that you’ll use a lot. Called an *injection*, an expression of the form X : Y (two expressions separated by a colon[[5]](#footnote-5)) are evaluated as follows:

* First, X and Y are evaluated (in that order). The result of evaluating Y must be a function that either takes no arguments or that takes a single argument compatible with the result of evaluating X.
* Next, the result of evaluating Y is called, passing in the result of evaluating X as its single parameter. We can say that X has been *injected into* Y.
* If this returns a result, that result becomes the result of the entire injection, otherwise the result is the result of evaluating X.

That sounds more complicated than it (typically) is, and most of the time, the obvious reading is going to be the correct one. In

A\_well : require(1 drop of A);

require(1 drop of A) evaluates to something into which a well (e.g., A\_well) can be injected.

As we will see in just a bit, injections can be combined into longer *chains*. In X : Y : X, (the result of evaluating) X is injected into (ditto) Y, and the result of *that* is injected into (still ditto) Z. This is most often used to define potentially sequences of actions to be applied to a drop.

The left-hand side of the colon in

A\_well : require(1 drop of A);

is straightforward. It refers to the current value of A\_well, which, unless somebody changes it, will be well number three.

Liquids

In the right-hand side, we see an expression 1 drop of A. This results in a value of type liquid, which represents a specific quantity of a reagent. The of operator takes two values, the first of type volume and the second of type reagent. The volume here, 1 drop demonstrates the way that DML deals with physical quantities. A number (either a constant or a computed[[6]](#footnote-6) value) followed by units represents a quantity of the type specified by the units. drop represents the volume of liquid typically dispensed by a given board, but the units here could equally have been uL, ml, microliters or any of several other possibilities. (Try what seems natural. If it isn’t recognized, it should probably be added.) Note that the units are only used for specifying the quantity. They don’t become a part of the value. So it’s perfectly reasonable to compare or add two volumes that were specified using different units.

Physical quantities

In addition to volume, the system also currently knows about time, frequency, voltage, temperature (that is, absolute temperature), and temperature difference. Values of the same quantity type may be compared against one another (v1 < v2), added to or subtracted from one another, and multiplied or divided by real values. In addition a real value divided by a time becomes a frequency and vice versa.[[7]](#footnote-7)

require expressions

When applied to a well, the expression require(L), for some liquid L, goes through the following steps:

1. First, the system checks to see whether the reagent associated with the well is the same as the one in the liquid required. If it isn’t, the system’s pipettor[[8]](#footnote-8) is asked to empty the well. If a volume is passed in rather than a liquid, this step is skipped, and the well’s reagent is used.
2. Next, the current contents of the well are checked to see if the volume in the well is sufficient. If it isn’t, the pipettor is asked to add liquid to the well to reach the desired level.

If we had added a second parameter, a volume, to the require call, e.g.,

A\_well : require(1 drop of A, 10uL);

whenever we determine that we have less than a drop of reagent A in the well, the pipettor will be asked to fill it so that it has 10 µL. This can be useful if a protocol is going to be run multiple times. The first time, the pipettor will be asked to add 10 µL, but the next time there will be more than one drop left, so nothing further will happen until the well gets sufficiently depleted.

In any case, when the statement is done, well number three will be guaranteed to have at least one drop of reagent A. Note that the first argument to require does not need to be the amount that will be dispensed. If the board you’re using can’t dispense from a well unless there is at least, say, 5 µL of reagent in it, use 5uL as the parameter.

Function calls

To call a function, write the name of the function followed by any arguments the function may take in parentheses and separated by commas, as require(1 drop of A), f(1, 2V, r), or g(). If the function is an *action* (one that takes no arguments and returns no value), the parentheses are optional when it is used as a statement.

Note that injections are another form of function call, and all injection targets are known to take either no arguments or one argument (the value being injected), so the parentheses are not required (or, for that matter allowed). In an injection like

A\_well : require(1 drop of A);

the injection target isn’t require, but rather the result of calling require(1 drop of A), which is itself a function that takes a well. Indeed, that statement could have been written as

require(1 drop of A)(A\_well)

but it wouldn’t have been as easy to read.

And let that be a warning that I won’t always be telling the full truth here. When I said that what comes before the parentheses is a name, that doesn’t tell the whole story. It’s actually any expression that evaluates to a function, and that will, in most cases be a name, but there are a number of functions that take injectable parameters (described below), and when those parameters are omitted in a call, the result is a one-argument function that is intended to be used as an injection target (as with require here), but which can also be used directly.

The waste reagent

The last statement in this block is different:

waste\_well’s reagent = waste;

For the waste well (well number 5), we don’t actually need anything in there, but we do want to say that anything that goes in there is waste. waste (or waste reagent or the waste reagent) is a special predefined reagent value that has the property that whenever its mixed with something else, the result is waste. When drops merge together or when drops enter wells, the system keeps track of the reagent of the resulting mixture. (You can also say things like A+B or even A+2\*B to create combination reagents yourself, but typically you can leave it to the system to keep track of things.) Normally, when a drop goes into a well whose reagent doesn’t match its reagent, you will get a warning. By specifying that the well contains waste, that warning is suppressed, because anything combined with waste is waste.

Attributes

This statement also demonstrates the DML notion of *attributes*. Many objects have values associated with them, and the way you refer to an attribute attr for an object obj is to say obj’s attr.[[9]](#footnote-9) In addition to its reagent, a well also has a number of other attributes including number, gate, exit pad, exit direction, volume, contents (a liquid), capacity, and remaining capacity. As the foregoing (incomplete) list shows, an attribute name may consist of multiple words. There may also be multiple ways to refer to the same attribute. For example, exit direction may also be referred to as exit dir.

Assignments

The final thing the statement demonstrates is *assignment*. An expression v = X assigns (the value of) X to v, which may be either a variable or, as here, an attribute of an object. Note that not all attributes are assignable. For example a well’s number or its capacity are constants, and its remaining capacity is computed at runtime but not directly assignable.

### Step 1: Mixing the first drop

Now that we have the wells set up, we can proceed to step one of the actual protocol:

1. Dispense drops of reagents A and B and merge them together. Walk the merged drop around for 10 seconds to ensure mixing.

We’ll do this in two pieces to make it easier to explain what’s going on. First, we’ll create the merged drop:

In file example.dml:

// Protocol parameters

pad pComb1 = (5, 4);

define run\_protocol {

// stuff we’ve seen already

drop d1;

[[

d1 = A\_well : dispense drop

: to pComb1

: accept merge from south

;

B\_well : dispense drop

: to pComb1 + 2 south

: merge into north

;

]]

// more to come

}

Pads

pComb1 (the first combination pad) is a pad which represents where this first merge will take place. It’s defined as (5,4), the pad whose x coordinate (or column) is 5 and whose y coordinate (or row) is 4, remembering that the pad in the lower left corner is (1,1). The syntax (x,y) specifies a pad at those coordinates, which may, of course, be complicated expressions.

Drops

Now let’s look at what’s going on in the protocol itself. First we declare a drop d1, but we don’t give it a value yet. A drop represents a liquid (that is to say a volume of a reagent) at a particular pad. Note that the drop maintains its identity as it moves around the board, although it may lose it when it merges with other drops.

Now on to the fun part. For now, we’ll ignore the funny [[ … ]] syntax and look at the two statements between the funny brackets. The first

d1 = A\_well : dispense drop

: to pComb1

: accept merge from south

;

dispenses a drop from A\_well, walks it to the pad specified by pComb1 (that is, (5,4)), and waits for a drop to merge with it. The result of that whole injection chain, the drop resulting from the merge, is assigned to d1.

Dispensing drops

Looking more closely, we first see A\_well : dispense drop. This injects well number three into dispense drop, the result of which will be a drop containing 1 drop (whatever that means on this board) of reagent A situated at well number three’s exit pad, (1,7).

Moving drops

Next, this drop is told to walk to pComb1 which is the pad at column 5, row 4. It does this by first walking to row 4 and then to column 5. If, for some reason, it was important to do the motion in the other order, we could be more specific and say

: to column pComb1’s column

: to row pComb1’s row

The result of this is the same drop that was dispensed, and it is injected into accept merge from south. There are a couple of things to mention here.

Directions

The first is that directions are a first-class type in DML (called direction or simply dir). There are two equivalent sets of direction constants: north, south, east, west and up, down, right, left. Use whichever seems more natural and don’t worry about being consistent. Given a direction d, you can also compute d turned around, d turned right (or clockwise) and d turned left (or counterclockwise). Each of these results in a new direction.

Merging drops

Now that we know what south means (although it was probably already clear), let’s look at what accept merge from south means.

Basically, it means that it’s going to wait for a drop on the pad two pads to the south of it (i.e., in our case, a pad at (5,2) to indicate that it desires to merge into it. The other drop will indicate this by moving to the drop and injecting the drop into merge into north. Why *two* pads to the south? Well, to prevent drops from running into one another and accidentally merging, DML assumes that the underlying system requires that drops maintain an empty pad on all sides (including diagonally). This allows the system to provide at least a primitive traffic control so that in most cases drops can move around the board without worrying about one another, yielding when necessary.[[10]](#footnote-10)

In any case, when both drops are in position and have indicated their desire to merge, the one invoking merge into is absorbed by the one invoking accept merge from, and the resulting larger drop will be at the pad of the one invoking accept merge from (and will, in fact, be the identical drop).

So we’ve got a drop of A in position. So all we need is a drop of B to get in position and be injected into merge into north. This is exactly what the second statement does:

B\_well : dispense drop

: to pComb1 + 2 south

: merge into north

;

The only thing that’s new here is that instead of moving to pComb1, this drop moves to pComb1 + 2 south. As is, hopefully, obvious, this is the pad two rows below pComb1. The value of 2 south is of type delta, which represents a number of steps in a direction. This could also have been written as south 2[[11]](#footnote-11) If the direction isn’t a constant, you can use the slightly more awkward 2 in direction d (or in dir). Note that any time you need a delta, you can pass in a direction. south is equivalent to 1 south.

One other thing to note is that the result of merge into south is *also* the merged drop. That is, both accept merge from and merge into get the same drop as the value. Because of this, you can extend the injection chain of either one to encompass more actions. **But!** It is almost certainly a really bad idea to do this for both of them. Which one you continue with (if either) will depend on what makes sense in your protocol (that is, does it make more sense to accept the merge and then keep moving or to merge into the other drop and keep moving). But don’t do both.

So we dispense a drop of A, move it into position. Then we dispense a drop of B and move it into position. Then we merge the two.

Simultaneous execution

Well, almost. Here’s where we address the mysterious [[ … ]] syntax. The actual code is

[[

d1 = A\_well : dispense drop

: to pComb1

: accept merge from south

;

B\_well : dispense drop

: to pComb1 + 2 south

: merge into north

;

]]

The [[ … ]] is called a *parallel block*, and it says that all of the statements within it are to be executed *at the same time*. So both drops are dispensed, both walk into position, and each informs the other that its ready to merge. The parallel block as a whole ends when all of its statements are done.

In my explanation of accept merge above, I said that the drop waits for another drop to get into position and call merge into. That’s not quite true. Because we expect that, as in this example, the drops will be moving at the same time, it doesn’t matter which of the two drops gets into position first. It will work equally well if the merge into drop gets there before the accept merge from drop. Whichever gets there first will wait for the other.

Now that we have the merged drop (which will contain 2 drops of A+B) in d1, we need to handle the second part of the step: “Walk the merged drop around for 10 seconds to ensure mixing.”

A caveat here. The method we’ll use is perhaps a bit more verbose that what I would probably use, but it allows me to explain some more features of DML:

In file example.dml:

// Protocol parameters

time walk\_time = 10s;

define square(int len) -> path {

return left len

: down len

: right len

: up len

;

}

define run\_protocol {

// stuff we’ve seen already

path walk\_to\_mix = square(3);

repeat for walk\_time {

d1 : walk\_to\_mix;

}

// more to come

}

Time

First, we give ourselves a variable, walk\_time that will hold the amount of time to walk. Making it a variable at the top of the file makes it easy to find if we find ourselves wanting to change it later. The variable is of type time and has a value of 10s. As per usual, the s could have been sec or secs or seconds, and we could have specified milliseconds, minutes or hours.

The next thing is to decide what we mean by “walk around”. What we’ll do here is to walk the drop along a square path, and just to be unnecessarily flexible, we’ll write a *helper function* to allow us to specify how big that square should be:

define square(int len) -> path {

return left len

: down len

: right len

: up len

;

}

Declaring function parameters

This defines a function named square that takes a single parameter, an integer (int) named len. As in most programming languages, parameters are specified as a comma-separated list between parentheses. If there are no parameters, the parentheses are optional.

The definition of square shows one common way of declaring a parameter by writing a type (int) followed by the name of the parameter (len). This declares a new variable whose value will be the actual value passed in when the function is called. If there is already a variable or function named len in the enclosing scope (typically at global level), this one will shadow it, which means that within the function the outer one will not be available, and modifications to the parameter will not affect the outer one. Note that a shadowing parameter—or, indeed, any shadowing declaration—does not need to be the same type as the name that is shadowed.

Unnamed parameters

Alternatively, the parameter can be declared without introducing a name. This can be done in two ways. First, you can give a number rather than a name following the type. If a parameter is declared as, e.g., pad 1 then uses within the body must also be pad 1. This syntax is sometimes clearer when there are multiple parameters of a given type and there isn’t really a good name to give them. The other alternative, when there’s only one such parameter is to simply declare the parameter as a drop (or just drop). In this case, uses of the parameter will be the drop or drop. We will see an example of this below.

Declaring function return types

Return statements

If, as with square, a function returns a value, the type of that value must be specified after the (optional) parameter block following an dash-greater (->) arrow. If there is no return value, this is omitted. If there is a return type, all control paths in the function must include a return statement specifying a value of that type. If there is no return type, return statements are optional and may not specify a value.

Paths

In the case of square, the function returns a path (also called a motion). This is essentially anything that a drop could be injected into where the result of the injection would be a drop.

Following the header, the body of the function is specified as a sequence of statements within braces ({ … }). In this case, the body contains a single return statement that returns a path that says to go left then down then right then up, each time traveling len steps, so we wind up right back where we started:

return left len

: down len

: right len

: up len

;

Or does it? If I’ve done my job, hopefully you’re saying “Of course it does.” But based on how I described how injections work above, it shouldn’t. After all left len is a delta, and injecting that delta into down len doesn’t make a lot of sense.

Compound injections

Except it does. (At least to me.) So what actually happens is that if you write an injection where the second part can take the first but *can* take the result of something injected into the first, what you get is a new injection target that passes the injected object into the first and then passes the result into the second. And that’s what’s going on here. We don’t actually do any injecting, but rather we create a compound injection that can be used later.

Now that we have square defined, let’s use it within run\_protocol. First we call square, passing in a parameter telling how big a square we want. We store the resulting path in a local variable:

path walk\_to\_mix = square(3);

Loops

Next we use that path to walk the drop d1 for the given amount of time (walk\_time):

repeat for walk\_time {

d1 : walk\_to\_mix;

}

This type of statement is called a *loop*, and it is used to perform a sequence of actions potentially more than once. The loop has two parts: its *header*, introduced by the repeat keyword and its *body*, signified by a block of statements enclosed in braces ({ … }) or double brackets ([[ … ]]). In this example, the body of the loop consists of a single statement, an injection in which d1 is passed to the path we got from calling square(3), causing the drop to be walked around a 3-by-3 square.

The loops header specifies when the loop should stop repeating. In detail, what happens when a repeat loop is executed is as follows:

1. The header is checked to see whether the body should be executed. It’s possible that the answer may be no even on the first check. If the answer is no, the loop is done.
2. The statements in the body are executed sequentially (for a body specified with braces) or simultaneously (for a body specified with double brackets).
3. Then we go back to step 1 to check to see whether we should execute the body again.

There are several different types of loop headers available, covering commonly needed scenarios:

* repeat for 10 minutes stops looping after ten minutes have passed. Note that the length of time is computed once when the loop is entered, so if you write repeat for t and t is 10 seconds, the loop will end after ten seconds even if t is changed to be 1hr before that time is up. If the amount of time is zero (or negative), the loop is never entered. Note also that header is only checked once the body is done executing, so if the body of a repeat for 5 seconds loop takes fifteen seconds to execute, it won’t be cut off when the five seconds is up.
* repeat 5 times runs the body the given number of times, which must be an integer (int). If the number is not positive, the body is never run. As with time-based loops, the number is evaluated once and stored, so if you say repeat n times, and n is ten when the loop is entered, the body will be executed ten times even if n is changed to 100 before it’s done.
* repeat until end\_ts runs the loop body until the *wall-clock* *time* gets to (or passes) end\_ts, which must be a value of type timestamp. Values of this type are typically acquired by evaluating time now (or current time) and possibly adding or subtracting values of type time (e.g., time now + 1 hour).
* repeat until n\_found >= 10 continues looping until the expression n\_found >= 10 evaluates to true. Unlike the repeat for and repeat … times forms, here the expression is evaluated before each entry to the loop (including the first), so changes to n\_found will be noticed. The provided expression must return a value of type bool (Boolean), either true or false.**[[12]](#footnote-12)**
* repeat while n\_found < 10 is identical to the prior example. Whereas a repeat until loop executes its body until the condition is true, a repeat while loop executes its body until the condition is false.
* repeat with int n = 1 to n\_iters creates a new variable n and initializes it to one (or whatever value follows the equals sign). This variable will be available within the loop body. It then evaluates and stores the final value (n\_iters in this case). When the header is checked, the body will be executed unless the value of the variable is greater than the final value. (Note that it is acceptable for the loop body to assign a new value to the variable. This will be reflected in the next test.) After the body is executed, the variable is incremented by its *step size*, which will be one unless another value is specified with a by clause. For example, repeat with real x = 0 to 1 by 0.2 will repeat the loop with x taking values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.
  + If the to in a repeat with header is replaced by down to, the variable is decremented (rather than incremented) by the step size, and the body executes unless the value of the variable is less than (rather than greater than) the final value.
  + If no type is specified, instead of creating a new variable, a variable that already exists in the surrounding scope is used. In this case the equals sign and initial value may be omitted and the variable will start from its current value.
  + For clarity, especially when no initial value is specified, the keyword to may be replaced by up to.

exit statements

Within a loop body, an exit statement tells the system that the loop is finished and should be exited immediately without checking whether to go around again:

repeat for 1 hour {

// Do stuff

if saw\_an\_error {

exit;

}

// Do more stuff

}

Named loops

If the loop is nested within another loop

repeat n times {

repeat for 10 minutes {

// Do stuff

If saw\_an\_error {

exit;

}

// Do more stuff

}

}

only the inner loop is exited and control passes to the following statement. If you want to exit out of multiple loops, you can add a *loop tag* and specify it in the exit statement. A loop tag is a name enclosed in brackets ([ … ]) that precedes the repeat keyword, as in

[outer\_loop]

repeat n times {

repeat for 10 minutes {

// Do stuff

If saw\_an\_error {

exit outer\_loop;

}

// Do more stuff

}

}

### Step 2: Mixing with the other two reagents

At this point, d1 contains a double-sized drop of a mixture of reagents A and B (that is, 2 drops of A+B) and is sitting on pad (5,4). We’re ready to move on to step two of the protocol:

1. Split the merged drop in two and merge the resulting drops with a drop of reagents C and D respectively. Mix these drops by “shuttling” (stepping back and forth) 10 times.

Here’s the code to do this:

In file example.dml:

// Protocol parameters

pad pComb2 = (10, 6);

pad pComb3 = (10, 2);

int n\_shuttles = 10;

define shuttle(an injectable drop, int n) {

repeat n times {

the drop : east : west;

}

}

define run\_protocol {

// stuff we’ve seen already

(eventual) drop d2;

[[

d1 : split to south as d2

: to pComb2

: accept merge from right

: shuttle(n\_shuttles)

;

d2 : to pComb3

: accept merge from right

: shuttle(n\_shuttles)

;

C\_well : dispense drop

: to pComb2 + 2 right

: merge into left

;

D\_well : dispense drop

: to pComb3 + 2 right

: merge into left

;

]]

// more to come

}

We start out by defining parameter variables for the places that the two mixtures take place:

pad pComb2 = (10, 6);

pad pComb3 = (10, 2);

These are in the lower middle heating zone (zone number four), so the drops will be in position to be heated in the next step. We also create a parameter variable for the number of times we want to shuttle the drops:

int n\_shuttles = 10;

To define what it means to shuttle a drop, this time we’ll write a function that does it (rather than, as we did with square, a function that returns a path that does it):

define shuttle(an injectable drop, int n) {

repeat n times {

the drop : east : west;

}

}

Most of this should already be familiar. We’re defining a function named shuttle that takes two arguments, a drop to shuttle and the number of times to repeat (n, an int), and returns no value. Within its body it performs a loop, repeating the loop body n times, and within the loop body, we have an injection chain that takes its drop parameter and steps it once to the east (that is, right) and once to the west (left).

There are, however two new bits of DML syntax used in this function definition.

First, the function is so simple that there’s really no reason to give its parameter a name. All that’s important is that it takes a drop, and so, as mentioned above, we can simply say that its first parameter is a drop. (We could also have simply said drop.) Within the body this parameter can be referred to as the drop (as is done here) or just drop.

Injectable parameters

Second, the drop parameter is specified not merely as a drop, but more specifically an injectable drop. We want to be able to use shuttle as an injection target, saying things like

d : shuttle(5);

for some drop d. By adding the injectable keyword preceding a parameter’s type, we’re saying that if all of the other parameters are specified, what we get back is something that a value of that type can be injected into. Without the injectable keyword, we have a function that takes two arguments, and we can say

shuttle(d, 5);

but we can’t use this in as the target of an injection. With the injectable keyword, shuttle(d,5) will still work, but we can also say simply shuttle(5) to get something that a drop can be passed to.

There can only be one injectable parameter, but it doesn’t need to be the first one. Note that if the function only takes one parameter, that parameter is automatically injectable.

Given that, let’s take a look at this part of run\_protocol:

(eventual) drop d2;

[[

d1 : split to south as d2

: to pComb2

: accept merge from right

: shuttle(n\_shuttles)

;

d2 : to pComb3

: accept merge from right

: shuttle(n\_shuttles)

;

C\_well : dispense drop

: to pComb2 + 2 right

: merge into left

;

D\_well : dispense drop

: to pComb3 + 2 right

: merge into left

;

]]

Again, most of this should be familiar. We have a parallel block that specifies motion for four drops that will all take place at the same time. One of the drops is drop d1, which we created previously, two are dispensed from C\_well and D\_well respectively, and the fourth, d2, we’ll get to in a bit. Each of these drops moves into position to merge, two of them calling accept merge and the other two calling merge into. The targets of the two merges go on to call shuttle passing in the value of parameter variable n\_shuttles to tell it how many times to walk back and forth. The block as a whole ends when all four movement chains have finished.

Splitting drops

If you look at the movement of drop d1, you’ll see that it has an extra step. Recall that at this point d1 is a double-sized drop, being the combination of a drop of A and a drop of B. We want to split this into two normal-sized drops (of the same mixture), and we do this by saying split to south. When this is done, instead of having a double-sized drop at (5,4), we now have two normal-sized drops, one at (5,4) and the other two pads below it at (5,2).

(eventual) variables

Now let’s look at the tricky bit. Variable d2 is declared to not hold a drop, but rather an (eventual) drop:[[13]](#footnote-13)

(eventual) drop d2;

When you try to use the value of a normal variable and that variable hasn’t yet been assigned a value, the system will complain and the program will stop. For an “eventual” variable, on the other hand, if you try to use it before it is set, that thread of control will simply pause until some other thread of control assigns it a value. So within the parallel block, when we say

d2 : to pComb3

: accept merge from right

: shuttle(n\_shuttles)

;

nothing will happen until and unless someone running in parallel gives d2 a value. This can be done using an assignment statement such as

d2 = the pad’s drop;

or it can be done, as here, by adding an as clause to a split to expression:

d1 : split to south as d2

: to pComb2

: accept merge from right

: shuttle(n\_shuttles)

;

As noted above, this split to expression will result in a new drop being formed at (5,2), but by adding as d2, it says that that drop should be assigned to the variable d2. As soon as this happens, the injection of d2 into to pComb3 is able to happen and does.

When all of this is done, there will be two drops on the board. d1 will hold 1 drop of A+B+2\*C**[[14]](#footnote-14)** and d2 will hold 1 drop of A+B+2\*D.

### Step 3: Heating the drops

Now that we have the two drops, we’re ready to move on to step 3:

1. Heat the combined drops at 80°C for 30 minutes.

To be able to show off … er, “explain” … more features of DML, we’ll do this in an elaborate manner that gives the user periodic feedback about how much time is remaining.

In file example.dml:

// Protocol parameters

heater incubator = pComb2's heater;

temperature incubation\_temp = 80C;

time incubation\_time = 30 minutes;

define incubate {

incubator's target = incubation\_temp;

wait until incubator is ready;

print "Incubating at", incubation\_temp,

"for", (incubation\_time : in H:M:S)+".";

timestamp end\_at = time now + incubation\_time;

time pause\_length = 1 minute;

repeat until end\_at {

time remaining = time until(end\_at);

print time now+":", (remaining : in M:S), "left.";

time t = remaining if remaining < pause\_length

else pause\_length;

wait for t;

}

incubator : off;

}

define run\_protocol {

// stuff we’ve seen already

incubate;

// more to come

}

As usual, we start out by declaring variables to hold the values that parameterize the protocol:

heater incubator = pComb2's heater;

temperature incubation\_temp = 80C;

time incubation\_time = 30 minutes;

We could specify the value of heater as heater #4, but instead we’ll use the heater associated with the pad we plan on sitting on, pComb2. If for some reason there is no heater associated with this pad, you’ll get an error at run time.

(maybe) values

The heater (or heating zone) attribute associated with a pad actually returns an object of type (maybe) heater rather than of type heater. (The parentheses are optional.) This means that the actual value is either a heater or the special value missing (or none). As noted above, if you try to use a (maybe) heater as a heater—the most common use case—and the actual value is missing, you’ll get an error. If you want to check whether a pad p has a heater, there are several ways to ask the question. First, you can ask directly about the attribute:

if p has a heater { … }

if p does not have a heater { … }

if p doesn’t have a heater { … }

Alternatively, you can ask about the value itself:

if p’s heater exists { … }

if p’s heater does not exist { … }

if p’s heater doesn’t exist { … }

if p’s heater is missing { … }

if p’s heater is not missing { … }

if p’s heater isn’t missing { … }

The latter set are valid any time you have a (maybe) value, which can also be used for function parameters and variable declarations. For example, you can say

define use\_if\_real(maybe heater h) {

if h doesn’t exist {

print ײNot a real heaterײ;

return;

// Do the work

}

and pass in p’s heater without worrying.

Temperature literals

For the incubation temperature, we set it equal to 80C. Temperatures (both absolute and relative) can be specified with a trailing C for Celsius or F for Fahrenheit. The system figures out from context whether it’s a temperature (or temp or temperature point or temp point) or a temperature difference (or temperature diff or temperature delta or temp difference or …).[[15]](#footnote-15) This even works with things like

temperature t = 80C + 5C;

Here, the 80C will be taken to be a temperature and the 5C will be taken to be a temperature difference, and the result will be 85C, a temperature. Of course, you can freely mix temperature systems. 50F + 10C will either be 20C as a temperature or about 37.8C as a temperature difference. (Note that if you add two temperature constants directly like that, the second one will always be taken to be a temperature difference. Also, if you directly compare two temperature constants, they will be taken to be temperatures.)

Actions at statement level

Inside run\_protocol, this step is trivial:

incubate;

but there is one subtlety that may not be apparent—and hopefully isn’t. As in most programming languages, a function is invoked by following it with a list of values in parentheses, e.g., square(3) above. But here, we’re not saying incubate() (although we could), but rather simply incubate. Recall that earlier we noted that a function that takes no arguments and returns no result, like incubate, is called an *action*. When the value of an expression used as a statement is an action, the action is immediately evaluated.

With that out of the way, all the work happens within the new action incubate:

define incubate {

incubator's target = incubation\_temp;

wait until incubator is ready;

print "Incubating at", incubation\_temp,

"for", (incubation\_time : in H:M:S), ".";

timestamp end\_at = time now + incubation\_time;

time pause\_length = 1 minute;

repeat until end\_at {

time remaining = time until(end\_at);

print time now+":", (remaining : in M:S), "left.";

time t = remaining if remaining < pause\_length

else pause\_length;

wait for t;

}

incubator : off;

}

Heaters

We’ll walk through this step by step. First, we tell the heater what temperature we want it to go to:

incubator's target = incubation\_temp;

Heaters are characterized by a number of attributes: current temperature (or simply temperature), maximum temperature, and minimum temperature**[[16]](#footnote-16)** as well as their target temperature (or simply target), which is the temperature the user has requested the heater go to. This last is the only one a program can modify. The target is actually a (maybe) temperature. If the value is missing, then the heater is to go to the ambient temperature.

Note that when we set a heater’s target temperature, this doesn’t necessarily mean that it will need to heat up. If the heater’s current temperature is already above the desired target, the heater will turn off until it cools off to the desired point.

Testing properties (is expressions)

Once we tell the heater to set its temperature, we wait until it gets there:

wait until incubator is ready;

An expression of the form X is Y requires that Y is a *property*, a function that can take X and return a Boolean (bool) value. You can use X is not Y or X isn’t Y (or, of course, not X is Y) to ask the opposite question.

Heaters have two defined properties: ready is true if the heater is maintaining its temperature at or near its designated target, which is not missing. ambient is true if the heater’s target is missing and its current temperature is below some (possibly dynamic) threshold set by the board.

wait expressions

A statement of the form wait until condition; pauses the current thread of control until condition is true. The condition is re-checked after each tick of the board’s clock.[[17]](#footnote-17)

print expressions

Now that the heater is at the temperature we want, we’ll notify the user that we’ll probably be here for a while:

print "Incubating at", incubation\_temp,

"for", (incubation\_time : in H:M:S)+".";

When used as a statement, a print expression takes one or more values, separated by commas, and prints them, separated by spaces. The values can be of any type. In this example, we see four values.

String literals

The first and third arguments here are string literals (that is, literals of the string type), which are written as sequences of characters between double quotes (" … ")[[18]](#footnote-18) The characters that make up a string can be any character except double quotes or line break characters (carriage returns or newlines). Within the string literal, you can also include various *escape sequences*, each of which begins with a backslash (\): \t represents a tab character, \n represents a newline, \r represents a carriage return, and \" represents a double quote. In addition, \u followed by exactly four hexadecimal characters represents the Unicode character with that code. For example, \u05D0 represents the Hebrew letter aleph (א).

Formatted quantities

For the final argument, we want to print a time value, but instead of printing it as a single number followed by units, say 30 min or 1,800 sec, we want to format it as H:M:S. To do this, we can use in H:M:S, either as a function (in H:M:S(incubation\_time)) or, as we do here, as an injection target. This function returns a string, so the result will be "0:30:00". Other available formats are in H:M and in M:S. If we had wanted to get it formatted as a string in units of our choice, we could say, e.g., incubation\_time as a string in hours, which would return something like "0.5 hr". If we just want it as a number in the units of our choice we could say, e.g., incubation\_time’s magnitude in seconds, which would return 1,800.0 as a real. Both as a string in and ’s magnitude in work for any physical quantity, as long as the units match the type.

String concatenations

We also want to end the line with a period. If we were to simply add "." as a fifth argument, the print would print a space between the time and the period, so instead, we concatenate the period to the time representation. When the plus (+) operator is used between a string and any value (in either order), the result is the concatenation of the two values, converted to strings if necessary.

Parentheses for grouping

Somewhat surprisingly, it took this long to get an opportunity to point out that in DML, as in pretty much the rest of the world, parentheses (( … )) are used to make sure that things group together correctly. In

print "Incubating at", incubation\_temp,

"for", (incubation\_time : in H:M:S)+".";

the parentheses around incubation\_time : in H:M:S are needed, because if they had been omitted, the statement would bind try to do the concatenation before the injection, as if it had been written as

print "Incubating at", incubation\_temp,

"for", incubation\_time : (in H:M:S + ".");

which would not have worked at all, but which would luckily have been caught at compile time.

print expressions as injection targets

The above description of what print does, while correct when it is used as a statement as we have here, is a bit of a fib. What actually happens is that when it is evaluated, it returns an action that, when invoked, prints the values. Since an action returned as the value of an expression used as a statement is immediately invoked, the two characterizations are the same. Doing it in this two-step process, however, means that a print expression can be used as part of an injection chain, as in

d : east 2

: print "The drop is now at", d’s pad

: north 2;

which will print the d’s location after it has moved east and before it has moved north. (That is, both the printing takes place at that time and the value printed is the drop’s location at that time.)

Now we wait:

timestamp end\_at = time now + incubation\_time;

time pause\_length = 1 minute;

repeat until end\_at {

time remaining = time until(end\_at);

print time now+":", (remaining : in M:S), "left.";

time t = remaining if remaining < pause\_length

else pause\_length;

wait for t;

}

We could do this simply by saying

wait for incubation\_time;

which would pause the current computation for the desired amount of time, but since this is going to take a while, we’ll instead give the user periodic feedback of how much time is remaining.

Timestamps

We start by figuring out at what time the pause will end:

timestamp end\_at = time now + incubation\_time;

time now returns the current wall-clock time as a timestamp, and when we add incubation\_time to it, we get a new timestamp that far in the future.

Next, we specify how often we’ll print our message or, equivalently, how long we’ll pause after each print statement:

time pause\_length = 1 minute;

Finally we go into a loop that will end at the specified timestamp:

repeat until end\_at {

time remaining = time until(end\_at);

print time now+":", (remaining : in M:S), "left.";

time t = remaining if remaining < pause\_length

else pause\_length;

wait for t;

}

Recall that there are two meanings to repeat until. If the value given is an expression of type bool (e.g., n < 5), the expression will be evaluated each time around the loop and the loop will stop when the value is true. If, on the other hand, the value is an expression of type timestamp, it is evaluated once to determine the wall-clock time after which the loop should stop. We use the second meaning here.

Inside the loop, we determine how much time we have left to wait:

time remaining = time until(end\_at);

time until(ts) gives us the amount of time until the given timestamp. If the timestamp is in the past, this will be negative. Similarly, time since(ts) gives us the time since the timestamp, which, if the timestamp is in the future, will be negative.

We print the remaining time:

print time now+":", (remaining : in M:S), "left.";

this time using in M:S to format it as minutes and seconds.

Conditional expressions

Next we calculate the time we will wait until the next time we print. We’ve specified that pause\_length is to be 1 minute, but we don’t want to wait that long if that’s more time than we have left, so we use a conditional expression to determine the actual amount of time to wait:

time t = remaining if remaining < pause\_length

else pause\_length;

An expression of the form X if C else Y first evaluates C and then evaluates either X or Y (but not both) depending on the value of C.

Once we know how long we want to wait, we wait for that amount of time:

wait for t;

Binary components and states

When the loop exits (i.e., after time now reaches or passes end\_at), we’re done incubating, and we can turn off the heater:

incubator : off;

Here we see another pair of useful concepts. The class binary component (or binary cpt or simply binary) encompasses all classes of thing that can be turned on and off. This includes heater, as well as pad, magnet, chiller, fan, power supply, well pad, and well gate. It also includes the unnamable class returned by the clock. All binary components have a state attribute whose value is of type binary state (or simply state), which has constants on and off. You can set this attribute directly (e.g., incubator’s state = off), but it’s more common to use on and off (or turn on and turn off) as injection targets. You can also say

incubator : toggle

(or toggle state) to flip from off to on or vice versa.

Turning heaters on and off

For a heater specifically, setting its state to off (by any of these means) is equivalent to setting its target temperature to none. The heater turns off and is allowed to return to ambient temperature. Setting the state to on, on the other hand, is slightly different. This is interpreted as a request to set the heater’s target to whatever it was before the last time it was turned off. So, once you’ve established a target, you can switch between states without having to reassert the target temperature.

### Step 4: Analyzing the drops

Finally, the drops are ready to be analyzed.

1. Measure each heated drop using an ESELog sensor.
   1. If the mean voltage read on the E1D1 channel is greater than 70V, the drop passes and should be preserved for later testing.
   2. Otherwise, the drop fails and is waste.

Here’s how we’ll do that:

In file example.dml:

// Protocol parameters

voltage reading\_target = 70 V;

extraction port product\_port = extraction port #2;

define run\_protocol {

// stuff we’ve seen already

define finish\_up(injectable drop d) {

eselog e = the board's eselog;

d : to e's target;

local r = e : take a reading

: write to csv file;

voltage v = r's e1d1\_on;

if v < reading\_target {

print d's reagent, "is good!", "("+v+")";

d : to product\_port : transfer out;

} else {

print d's reagent, "is bad.", "("+v+")";

d : become waste : to waste\_well : enter well;

}

}

[[

d2 : finish\_up;

d1 : finish\_up;

]]

prompt "Finished at", time now;

}

There are two parameters relative to this phase. The first is the voltage cutoff that we’ll use to distinguish good reagents from bad ones:

voltage reading\_target = 70 V;

Extraction ports

The second is the extraction port we’ll use to extract the good drops for later testing:

extraction port product\_port = extraction port #2;

An extraction port (aka, extraction point, extraction hole, or simply hole) is a hole in the board’s lid that can be used to add liquid to or remove liquid from the pad below. In our case, we’ll be using the port number 2, above pad (7,10). To transfer liquid in through an extraction port, you can say

ep : transfer in 2uL of r;

To transfer out, move the drop to the extraction port and call transfer out:

d : to ep : transfer out;

This will remove the whole of the drop, and the drop will no longer be considered to be on the board. If you only want part of the drop removed, you can specify the amount to remove:

d : to ep : transfer out(0.5uL);

This will ask the pipettor to remove the specified volume, and if it is less than the drop’s volume, the drop will shrink by that much.

The last step of the protocol is implemented by the finish\_up action, which processes one drop and which we’ll get to in a minute. With it, we can simply say

[[

d2 : finish\_up;

d1 : finish\_up;

]]

to run each of our two drops through finish\_up.

Note that finish\_up will require walking the drop to the pad the sensor uses to take a reading and then sitting there while the reading is taken before moving away. But because of the traffic control of the underlying system, we can simply have both drops execute this action at the same time. One of them will get there first, and the other will wait, two pads away, until the first drop leaves.

Dealing with deadlock

It is, of course, possible that you’ll get unlucky and the default paths used to walk to the target pad cause the two drops to “deadlock” by getting into a state in which each is waiting for the other to get out of the way. It is also possible that the second, waiting, drop may block the first drop’s default path to get off of the sensor’s pad. In either of these cases, you may have to be more specific about the paths taken, e.g., by adding a to column or to row component to the movement to ensure that they can’t get in each other’s way. Because of this possibility of needing to tweak paths, it’s usually a good idea to run you protocol in simulation before running it with a physical board and, especially, before running it using expensive and/or irreplaceable reagents.

Finally, we let the user know we’re done and when we finished.

prompt "Finished at", time now;

prompt expressions

In this case, I’ve chosen to use a prompt statement rather than a print statement, both in order to introduce it and because it is more eye-catching. prompt is just like print, but instead of writing the message to the console, it pops up a dialog box and requires the user to acknowledge it.

Like print, prompt can be used in injection chains, and this can be useful when you require the user to do something before you can proceed:

d : move\_into\_position

: prompt “Ensure the filter is in place.”

: do\_the\_thing\_that\_requires\_the\_filter

;

Unlike print, prompt can be used without any expression, in which case the dialog box contains a default message. Because one of its functions is waiting for the user to do something, prompt can also be written as pause for user or wait for user.

Local functions

The first thing to notice about the finish\_up action, which does the actual work, is that we’ve put its definition within the definition of run\_protocol. This has a couple of consequences. First, it means that finish\_up can only be called from within run\_protocol. Outside an invocation of run\_protocol, the name is simply not defined (or it may be defined to be another function or variable). Second, it means that finish\_up has access to any variables within run\_protocol (including any parameters it may have been called with) that were declared before it in the body of run\_protocol.

Defining local functions like this for things like protocol phases, which nobody outside of the protocol should want to call, can be a useful technique. It also means that several local functions or actions can all see the parameters to the enclosing function and any variables declared at the beginning of it. This means that these variables can be used to store state between invocations of the local actions:

define process(a drop) {

int n = 0;

define trace {

n = n+1;

print the drop+”:”, “Times called:”, n;

}

repeat for 10 minutes {

the drop : do\_the\_thing;

trace;

}

}

Here, n is used to keep track of the number of times we’ve gone around the loop, and trace both refers to and increments it, as well as referring to the drop, a parameter to process.

Another thing to note about that example is that each time process is invoked, a new environment containing the drop and n is created, and a new trace is created that refers to that environment. This means that even if we call process in parallel:

[[

d1 : process;

d2 : process;

]]

each one will have its own drop, and each one’s trace will modify its own n.

Now, let’s look at what’s going on inside of finish\_up.

define finish\_up(injectable drop d) {

eselog e = the board's eselog;

d : to e's target;

local r = e : take a reading

: write to csv file;

voltage v = r's e1d1\_on;

if v < reading\_target {

print d's reagent, "is good!", "("+v+")";

d : to product\_port : transfer out;

} else {

print d's reagent, "is bad.", "("+v+")";

d : become waste : to waste\_well : enter well;

}

}

The board

In the first step

eselog e = the board’s eselog;

we use the board to get access to the sensor we need. This form (or just board) returns a value representing the board as a whole, which is mostly useful for its attributes, which can be used to get various components that may be present in the system. Besides eselog, these attributes include power supply, fan, and clock. Aside from clock, these all return (maybe) values, so you can say things like if the board has a power supply or if the board’s fan exists.

Sensors

The value of e represents the board’s ESELog. It’s type, eselog, is a subtype of sensor and shares with it all of the things that are common to all sensors, several of which we’ll use below. All sensor objects have a target attribute, which is a (maybe) pad that gives the pad, if any, the sensor will take its reading from. This attribute can be set to a new value by assignment, but note that this should be done in response to the hardware changing, not in an attempt to change the hardware, as it will not do so. If you wish to engage the hardware to ensure that it is looking where it should be, you can say s : aim (or s : aim(p) for some pad p). For an ESELog, this will turn on the sensor’s targeting laser and prompt you to confirm when it’s in the right place. Note that if an argument is given to aim, this does not change the sensor’s target.

The main thing you do with sensors is to use them to take readings:

s : take a reading

(or take reading or take readings). This can take two extra parameters, both optional. The first is an integer giving the number of samples to take. If it is not given, it defaults to the value of the sensor’s n samples attribute. The second is either a time or a frequency giving an indication of the rate at which the samples should be taken, measured from the start of one to the start of the next. If not given, this defaults to the sensor’s sampling rate (or sampling rate), a frequency, or its sampling interval (or sample interval), a time. Both of these are views of the same value, so changing one changes the other. “test”

Once we have the sensor, we walk the drop to its target pad:

d : to e's target;

and to take the reading:

local r = e : take a reading

: write to csv file;

Writing readings to files

The value that comes back from take a reading is an eselog reading, which is a special case of sensor reading. One of the things that you can do with an eselog reading is to write it to a CSV file, which can be done with write to csv file (write csv file, write to file, write file). This can take two optional parameters. The first is a string specifying the name of the file to use, and the second is a timestamp, which defaults to the timestamp associated with the first sample in the sensor reading. The first parameter (or its default) is actually a template for the file name to use and may include indications of where to put in parts of the timestamp. These substitution specifications are based on those used by strftime. In Thylacine, the default file name template is “eselog-%Y-%m-%d\_%H\_%M\_%S.%f.csv”, which substitutes in the four-digit year (%Y), two-digit month (%m), two-digit day of the month (%d), two-digit 24-hour hour (%H), two-digit minute (%M), two-digit second (%S) and six-digit fraction of a second (%f). More formats can be found at <https://strftime.org/>.

local variable declarations

Looking at the declaration of r, you’ll see that while the initial value is of type eselog reading, we don’t say eselog reading r, but rather local r. In fact, the type of r will be eselog reading. When an initial value is given, as a shorthand, you can give the variable’s type as local to mean “whatever the type of the initialization expression is”. This can be convenient, especially for variables that have complicated (or even inexpressible) types, but it should be used sparingly, especially if the variable is going to have a new value assigned later. For instance, if you say

local multiplier = 1;

and then later say

multiplier = 2.5;

the compiler will complain, because the type of multiplier is the type of 1, which is integer, not real.

Sample types

Now we need to check whether the drop passes or not.

voltage v = r's e1d1\_on;

Within the eselog reading are a number of *samples* which can be thought of as like the columns in a spreadsheet table, where the values for each sample the sensor took are like the values for that column in each row.

As an eselog reading, r has attributes e1d1\_on, e1d1\_off, e2d2\_on, e2d2\_off, e1d2\_on, and e1d2\_off, all of which are of type voltage sample and which represent the voltage read long each channel. In addition r’s temperature is a temperature sample representing the temperature when the reading was taken, r’s ticket is an integer sample representing a serial number given to each sample by the device. And, like all sensor readings, r’s timestamp represents the times the samples were taken.

So if a sample represents a column of values, what can you do with it? Basically, you can use its attributes to find out things about the values. It’s attributes include mean (or arithmetic mean), std dev (or std deviation or standard dev or standard deviation), median, count, min (or minimum or min value or minimum value), max (or maximum or max value or maximum value), range (max–min), first (or first value), last (or last value), harmonic mean, and geometric mean.

If you try to convert it to its element type, as we do here, using a voltage sample as a voltage, the value that’s used is the mean, so what we get is the mean of the readings along that channel.

You can create your own sample objects. For instance, if you want to compute performance statistics on running a function, you can do something like

time sample s = an empty time sample;

repeat n times {

timestamp start = time now

do\_the\_thing;

s : add(time since(start));

}

print “It took”, s’s mean, “+-“, s’s standard deviation;

an empty time sample gets you an empty time sample and add(t) adds t to it. Samples can be created for any physical quantity type as well as temperature point, timestamp, integer, and real. Note that in some cases the attribute values may not be of the element type. For instance, an int sample’s mean will be a real number rather than an integer, and a timestamp sample’s standard deviation will be a time rather than a timestamp.

Now that we have the value we want to test in v, the only thing left to do is to compare it and take the appropriate action:

if v < reading\_target {

print d's reagent, "is good!", "("+v+")";

d : to product\_port : transfer out;

} else {

print d's reagent, "is bad.", "("+v+")";

d : become waste : to waste\_well : enter well;

}

Conditional statements

Here we see an example of a *conditional* statement. Such a statement is of the form

if condition\_1 {

action\_1;

} else if condition\_2 {

action\_2;

} else if condition\_3 {

action\_3;

} else {

default\_action;

}

where there may be any number of else if clauses (including zero) and where the else clause is optional.[[19]](#footnote-19) The logic is that first condition\_1 is evaluated. If it evaluates to true, action\_1 (and all other statements in its block) are executed and the statement is done. If condition\_1 evaluated to false, condition\_2 is evaluated, and if it is true, all of the statements in the second block are executed and the statement is done. This continues through all of the else if conditions. If none of them evaluate to true and there is an else clause, its block is executed.

In our example, we have a single condition, v < reading\_target, and no else if clauses, but there is an else clause, so one of the blocks will necessarily be evaluated.

If the test is satisfied, the drop is good, so we say so:

print d's reagent, "is good!", "("+v+")";

We then walk the drop to the extraction port and instruct the pipettor to remove it:

d : to product\_port : transfer out;

If the test wasn’t satisfied, we enter the else block and let the user know”

print d's reagent, "is bad.", "("+v+")";

and then we walk the drop into the waste well:

d : become waste : to waste\_well : enter well;

Changing a drop’s reagent

As noted above, a drop has a reagent attribute, and you can change the reagent by setting that value:

d’s reagent = r;

become r, as an injection target, is a shorthand for this. Changing the reagent is useful in a couple of situations. First, as in our example, when we’ve decided that the drop is waste and we’re in the process of walking it to a waste well. Strictly speaking, this isn’t necessary, but when there are a large number of drops on a board and there is a GUI showing a representation of what is where, it can be useful to be able to distinguish at a glance between waste drops and drops that are still of interest. Second, when you are building up some product by merging, heating, and otherwise munging drops, there may be a point at which you can say “Okay, we’ve now got what we were trying to make.” In the Thylacine GUI up to that point, the drop will have been represented by a pie chart of the reagents that make up its components. Changing the reagent to something simpler allows it to be more simply represented.

Entering a well

Finally, the drop calls to waste\_well to walk to the well (or, rather, the well’s exit pad) and then calls enter well to go in. It goes without saying that if you inject a drop into enter well and it isn’t on the exit pad for some well, an error will be raised.

## Putting it all together

The foregoing sections have built up the protocol piecemeal over quite a number of pages, so it may seem as though there’s a lot of it, but in fact the whole example.dml file, comments and all, is only 171 lines long:

File example.dml:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Our first protocol

\*

\* Author: Evan Kirshenbaum

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Protocol parameters

reagent A = reagent "A";

reagent B = reagent "B";

reagent C = reagent "C";

reagent D = reagent "D";

well A\_well = well #3;

well B\_well = well #4;

well C\_well = well #7;

well D\_well = well #8;

well waste\_well = well #5;

pad pComb1 = (5, 4);

pad pComb2 = (10, 6);

pad pComb3 = (10, 2);

time walk\_time = 10s;

int n\_shuttles = 10;

heater incubator = pComb2's heater;

temperature incubation\_temp = 80C;

time incubation\_time = 30 minutes;

voltage reading\_target = 70 V;

extraction port product\_port = extraction port #2;

// For testing

define speed\_up {

incubation\_time = 10 seconds;

}

// Helper functions

define square(int len) -> path {

return left len

: down len

: right len

: up len

;

}

define shuttle(an injectable drop, int n) {

repeat n times {

the drop : east : west;

}

}

define incubate {

incubator’s target = incubation\_temp;

wait until incubator is ready;

print "Incubating at", incubation\_temp,

"for", (incubation\_time : in H:M:S)+".";

timestamp end\_at = time now + incubation\_time;

time pause\_length = 1 minute;

repeat until end\_at {

time remaining = time until(end\_at);

print time now+":", (remaining : in M:S), "left.";

time t = remaining if remaining < pause\_length

else pause\_length;

wait for t;

}

incubator : off;

}

// The protocol

define run\_protocol {

// Step 0:

A\_well : require(1 drop of A);

B\_well : require(1 drop of B);

C\_well : require(1 drop of C);

D\_well : require(1 drop of D);

waste\_well's reagent = waste;

// Step 1. Dispense drops of reagents A and B and merge them

// together. Walk the merged drop around for 10

// seconds to ensure mixing.

drop d1;

[[

d1 = A\_well : dispense drop

: to pComb1

: accept merge from south

;

B\_well : dispense drop

: to pComb1 + 2 south

: merge into north

;

]]

path walk\_to\_mix = square(3);

repeat for walk\_time {

d1 : walk\_to\_mix;

}

// Step 2. Split the merged drop in two and merge the

// resulting drops with a drop of reagents C and D

// respectively. Mix these drops by “shuttling”

// (stepping back and forth) 10 times.

(eventual) drop d2;

[[

d1 : split to south as d2

: to pComb2

: accept merge from right

: shuttle(n\_shuttles)

;

d2 : to pComb3

: accept merge from right

: shuttle(n\_shuttles)

;

C\_well : dispense drop

: to pComb2 + 2 right

: merge into left

;

D\_well : dispense drop

: to pComb3 + 2 right

: merge into left

;

]]

// Step 3. Heat the combined drops at 80°C for 30 minutes.

incubate;

// Step 4. Measure each heated drop using an ESELog sensor.

// If the mean voltage read on the E1D1 channel is

// greater than 70V, the drop passes and should be

// preserved for later testing. Otherwise, the drop

// fails and is waste.

define finish\_up(injectable drop d) {

eselog e = the board's eselog;

d : to e's target;

local r = e : take a reading

: write to csv file;

voltage v = r's e1d1\_on;

if v < reading\_target {

print d's reagent, "is good!", "("+v+")";

d : to product\_port : transfer out;

} else {

print d's reagent, "is bad.", "("+v+")";

d : become waste : to waste\_well : enter well;

}

}

[[

d2 : finish\_up;

d1 : finish\_up;

]]

prompt "Finished at", time now;

}

# Things Not Covered

While I tried to cover as much of DML as I could in a single program, there are still some topics that didn’t make it in, and I’ll try to cover them here. Since my time at the moment is *extremely* tight, these won’t necessarily be in any order (and may be somewhat brief).

### Functions

Function definition keywords

In the example, I showed several examples of defining functions, both with and without parameters and with and without return types, all using the define keyword. In all cases, define can be replaced by def (as in Python) or macro (as it was in an early version of DML, back when it was simply called “the macro language”).

In addition, if the function has no parameters and doesn’t return a value, you can replace define by action, as in

action complain {

print “It didn’t work!”;

}

If the function doesn’t return a value (whether or not it has parameters), you can use procedure or proc:

procedure print\_reagent(a well) {

print the well+”’s” reagent is the well’s reagent;

}

On the other hand, if it *does* return a value, you can use function or func:

function double(int n) -> int {

return 2\*n;

}

All of these are a matter of personal preference. define, def, and even macro will work in all cases.

Parallel-block functions

All of the examples of function given so far define their bodies using curly-brace blocks ({ … }), which evaluate their statements in order. It is also possible to use double-bracket blocks ([[ … ]]), which evaluate their statements in parallel:

define merge\_drops(drop 1, drop 2, dir to\_d2) [[

drop 1 : accept merge from to\_d2;

drop 2 : merge into (to\_d2 turned around);

]]

Anonymous function values

All of the examples so far include a name following the keyword and occur at statement level. This had the effect of defining the function with that name. But if you omit the name, you can still create a function. It just won’t have a name. This can be useful when you want to create a simple function to use as an injection target:[[20]](#footnote-20)

d : … : def (drop d) { print “d is at”, d’s pad; } : …

(Yes, this is a bit ugly. Wait until the next section.)

In fact a definition like

function double(int n) -> int {

return 2\*n;

}

is actual equivalent (mostly) to

local double = function(int n) -> int {

return 2\*n;

}

When there is no name for a function value, the keyword lambda is another option.

Expression-bodied functions

As an alternative to providing a (sequential or parallel) block as the function body, you can instead use colon (:) followed by an expression, so simple functions can be written as, e.g.,

define double(real x) : 2\*x;

When the body is an expression, specifying the return type is optional. If it is omitted, it will be inferred from the value of the expression.

Expression-bodied functions may also be anonymous, and so the earlier example could have been written as

d : … : (lambda(drop d): print “d is at”, d’s pad) : …;

and if you want to change from a drop to its reagent in the middle of a chain, you can simply say

d : … : (lambda(drop d): d’s reagent) : …;

Note that in these examples, the parentheses are necessary to ensure that the following colon isn’t considered to be part of the function’s body.

Recursive functions

The name of a function (if there is one) is visible within the body of the function, so the function can call itself, as in this definition of the factorial functions:

define factorial(int n) -> int {

return 1 if n < 2 else n\*factorial(n-1);

}

Unfortunately (at least in the current implementation), to define a recursive function using an expression body, the return type specification is mandatory:[[21]](#footnote-21)

define factorial(int n) -> int : 1 if n < 2 else n\*factorial(n-1);

Without the return type, the compiler can’t figure out the type of the recursive call, which is needed to figure out the return type.

Mutually recursive functions

This doesn’t work yet. I’m not sure why. The compiler doesn’t accept the forward declaration. I think I know how to fix it, but it’s not completely trivial.

Sometimes you will have two functions that need to call one another. In this case, you need to add a *forward declaration* for the one that is defined later so that the compiler knows to expect it and what its type will be. This is simply the function header without any body.

For example, the following two functions ping-pong back and forth between one another:

define f2(int n);

define f1(int n) {

print "First:", n;

if n > 0 {

f2(n-1);

}

}

define f2(int n) {

print "Second:", n;

if n > 0 {

f1(n-1);

}

}

Without the forward declaration of f2, the compiler wouldn’t know what to do when it saw the f2(n-1) call inside of f1.[[22]](#footnote-22)

1. It hasn’t been to the extent I’d like, but I’m working on it. [↑](#footnote-ref-1)
2. As a clue, if you see an error message that starts with something like 45:12, look at line 45, character 12. But if there are multiple such errors, the real problem will often be on the second or third of them. In any case, the actual message is probably not going to give you a lot of guidance. You’re probably better off just looking at the line in question and seeing if you can figure out what might be wrong. [↑](#footnote-ref-2)
3. Although there are also a lot of keywords that can be used as names, because the compiler can tell that they aren’t being used as keywords. [↑](#footnote-ref-3)
4. Unfortunately, there are too many situations in which two literal integers could be separated by a comma to allow commas to be used as a separator. [↑](#footnote-ref-4)
5. Or a vertical bar (|). The vertical bar is more reminiscent of the Unix shell’s “pipe” operator, and I had planned to deprecate the colon in favor of it, since the colon is probably more useful elsewhere, but the colon just looks better to me in this context, so I’m leaving it for now. [↑](#footnote-ref-5)
6. Note that in programming terms, the units “bind tighter” than things like addition, so an expression like x+1 uL will try to add 1uL to x, which is fine if x is a volume, but if x is a number, you will want to say (x+1)uL. [↑](#footnote-ref-6)
7. The exception is that temperature values do not allow normal arithmetic. You can subtract one temperature from another, yielding a temperature difference, and you can add or subtract a temperature difference from a temperature, yielding another temperature. [↑](#footnote-ref-7)
8. If the system doesn’t have an automated pipettor, a dialog box will probably pop up asking the user to do the transfer manually. [↑](#footnote-ref-8)
9. You can also say obj.attr, but I find the possessive notation more intuitive. After having the compiler complain enough times when copying and pasting from this document, you can also say obj’s attr, with a curved apostrophe. [↑](#footnote-ref-9)
10. It doesn’t always work, and sometimes it’s necessary to use other mechanisms to avoid drops entering into a state called “deadlock”, where two drops are both attempting to yield to one another in a way that will never end. These mechanisms are beyond the scope of this example. Note that DML provides a way to specify that you really want one drop to move right next to another, but it’s (intentionally) ugly and cumbersome and, again, beyond the scope of this example. [↑](#footnote-ref-10)
11. The two syntaxes aren’t quite identical, but they hopefully both do what you expect in the contexts you are likely to use them in, so you won’t notice. In particular, x+1 south is equivalent to x+(1 south), while south x+1 is equivalent to south (x+1). [↑](#footnote-ref-11)
12. DML also allows True, TRUE, yes, Yes, and YES for true and False, FALSE, no, No, and NO for false. [↑](#footnote-ref-12)
13. This could have also been written as eventual drop, (future) drop or future drop. [↑](#footnote-ref-13)
14. The result of mixing equal parts A+B and C. Getting equal parts A, B, and C is trickier but will be made straightforward in a future version of DML. [↑](#footnote-ref-14)
15. It actually allows it to be ambiguous between the two readings—even if you do math on it—until you try to do something like assign it to a variable or pass it to a function. [↑](#footnote-ref-15)
16. Where temperature can be replaced by temp and minimum and maximum can be simply min and max. [↑](#footnote-ref-16)
17. Note that this means that if you pause the clock, e.g., by saying pause the clock or clock : off, this thread of control will be blocked until the clock is resumed (resume the clock or clock : on). [↑](#footnote-ref-17)
18. For ease of cutting and pasting and using editors that put them in by default, you can also use smart quotes (“ … ”). They need not be used appropriately. [↑](#footnote-ref-18)
19. Programmers familiar with C or C++ should note that the body of each of these clauses is a block, not a statement. The braces (or double brackets) are required. [↑](#footnote-ref-19)
20. It will become more useful when functions can be used as arguments and return values. [↑](#footnote-ref-20)
21. Okay, I just tried that and it didn’t work. It works if you put the recursive call in parentheses as (factorial(n-1)), and it works if you put the multiplication in parentheses (n\*factorial(n-1)). Oh, well. One more for the issues list. [↑](#footnote-ref-21)
22. That’s not quite true. A smarter compiler could look ahead and see what was going to be defined within the block. We don’t have a smarter compiler at the moment. [↑](#footnote-ref-22)