Discrete Math Language (DML)

Secure Systems Lab, Stony Brook University

DML is a programming language that has been custom designed for teaching discrete math. It aims for a simple syntax that is familiar to math students, and attempts to match the notations used in discrete math texts. In addition, it offers several operators and constructs that aren't available in most programming languages, e.g., quantification and boolean formula satisfaction checking.

The main goal of DML is to introduce discrete math as a gateway to programming. Initially, student programs will be simple, focusing on *what* needs to be computed, rather than *how*. DML uses brute-force searching as needed to turn these specifications into computations that yield the specified answer. As students gain more experience, they begin to fill in the most critical aspects of *how*, thereby producing code that is more efficient. In order to support this transition, DML tries to preserve the syntax of existing languages (specifically, Python) for constructs and concepts that are commonly supported in existing languages. Here are some examples of DML formulas:

- Generate the set of positive even numbers under 100: `{x for x in 0..100 if x % 2 = 0}`
- Check if a boolean formula is always true: `valid (x->y) || (y->z) || (z->x)`

  Note that `->` is the implication operator.
- Define a function that checks if a given number is prime: `fun prime(x) = forall (y in 2..(x-1)) x % y != 0`

Contents

1 Overview

DML is a *functional language,* adhering to the semantics of mathematical definitions. It is a typed language, where all expressions must have a well-defined type. (Types are inferred automatically, so you need to worry about them only if there is a type error.) Finally, DML is an interpreted language that is designed for interactive use. As such, it comes with a command-editing shell `dmlsh` that provides command completion, history, and editing features.

DML programs consist of *expressions, definitions* and *commands*. These three concepts are interdependent, so we will introduce all of them briefly here, and then proceed to describe them in depth in the next several sections of this document.

The simplest types of expressions are *constants* such as the number `5` or the string `'tomato'`. DML constants can be of four basic types: integers, reals, strings, and booleans. Values of these simple types can be aggregated into collections such as sets, tuples and lists.

Expressions operate over values and variables, combining them using operators and functions to produce new values. For instance, the expression `10-2*3` yields the value `4`.

DML *definitions* can define new variables or functions. A variable definition (aka a binding) consists of a variable name and an expression separated by an equals sign, e.g.,

```
xyz = 7*3 + 10
```

An example of function definition is:

```
fun f(x) = x*x
```

It defines a function `f` that takes one parameter and returns the square of this parameter. The keyword `fun` enables DML to distinguish between function definitions and variable definitions.

A typical program in DML will consist of several variable and function definitions, followed by one or more expressions that produce an output of interest to the programmer.

2 Values and Types

We describe the base types in Section 2.1 and derived types in Section 2.2.

2.1 Base types

DML base types include numbers, strings and booleans. Numeric types supported include integers and real numbers.

2.1.1 Integers

Integers can range from -2^{62} to $2^{62}-1$ and use the familiar decimal representation, e.g., `15` for the number fifteen. Integers may also be specified in binary, octal or hexadecimal form using conventions typically used in C, C++ and Python.

Common arithmetic operations on integers use the same operators as in math. The full set of operations supported on integers is shown in the table below:

DML Symbol	Math equivalent	Explanation
`+`, `-`	$+$, $-$	Addition, subtraction, or unary minus
`*`, `/`, `%`	$\times$, $/$, **mod**	Multiplication, division, and modulo
`^`	superscript	exponentiation[1]
`=`, `==`, `!=`	$=$, $\neq$	Equality and disequality
`>`, `<`, `>=`, `<=`	$>$, $<$, $\geq$, $\leq$	Inequality

2.1.2 Reals

Reals, also called floating point numbers in double precision format, use the same notation as in most other programming languages. They are written out in the usual way, e.g., `1.75`. Real numbers using scientific notation are written out somewhat differently from math, but in a manner consistent with all other programming languages. For instance, 1.53×10^{-27} would be written out as `1.53e-27`.

[1] Note that TeX/Latex, which are most commonly used to typeset math, uses "^" for superscripts. It is convenient to use the same symbol for exponentiation in DML.

Reals in DML support the same set of operations as integers, with the exception of the modulo operation (which is only defined for integers). In addition, there are three functions to convert real numbers into integers:

- `round` rounds its argument, e.g., `round(1.49)` is `1`, while `round(1.5)` is `2`.
- `ceil` computes the closest integer greater than or equal to the argument, e.g., `ceil(1.49)` is `2`.
- `floor` computes the closest integer less than or equal to the argument, e.g., `floor(1.99)` is `1`.

2.1.3 Booleans

Booleans are distinct from integers and cannot be intermixed. They arise mainly from comparison operations, but may also be explicitly introduced using the literals `true` and `false`. Variables can also be of boolean type, and they can be operated on by logical operators and functions. Operations on booleans are as shown in the following table. (All of the comparison operators can be applied to booleans, but it is unusual to do so.)

DML Symbol	Math equivalent	Explanation
`<->`	$\leftrightarrow$	Logical equivalence ("if and only if")
`->`	$\rightarrow$	Logical implication
`||`	$\vee$	Disjunction ("or" operator)
`&&`	$\wedge$	Conjunction ("and" operator)
`!`	$\neg$	Logical negation

2.1.4 Strings

String literals can be enclosed in single or double quotes, e.g., `"This is a string"` and `'This is another string'`. Common escape sequences for special characters are supported in a manner compatible with languages such as C and Python. For example, `'a\nb'` is a string that includes a newline character, denoted `\n`. DML will use these escape sequences when it shows the values of string variables and constants, but then such a string is provided as input to the `print` function, it is printed as a newline. Here is a DML session illustrating this.

```
dml> x="a\nb"
x:string = "a\nb"
dml> print(x)
a
b
dml>
```

Strings can be concatenated using the operator `++`, and can be compared with each other using any of the comparison operators discussed above.

```
dml> "Hello" ++ "World"
"HelloWorld"
dml>
```

2.2 Aggregate Types

DML supports four operators to construct new types by combining existing ones. These include: *tuples*, *sets*, *lists*, and *dictionaries*. The last three of these are called *container types* or *collections*. The contents of containers should be homogeneous, i.e., all elements in a collection must have the same type. (However, since integers are also real, it is permissible to mix integers and reals in the same container.) This homogeneity condition does not apply to tuples, whose components can be of different types.

2.2.1 Sets

Sets in DML are no different from sets in mathematics, with one exception: as a programming language, DML sets are limited to be finite. The type `int` is a finite subset of the set $\mathbb{Z}$ of integers, limited to the range of -2^{62} to $2^{62}-1$. Similarly, the set `real` is a finite subset of $\mathbb{R}$ (since its elements are represented with a finite number of bits). As in mathematics, DML sets are also enclosed within braces.

2.2.1.1 Constructing sets by enumerating elements

In mathematics, we often construct sets with arbitrary elements, e.g.,

$$\begin{aligned} A &= \{Alex, Tippy, Shells, Shadow\} \\ B &= \{red, blue, yellow\} \\ C &= \{1, 2, 3, 4\} \end{aligned}$$

The same sets would be specified in DML as follows:

```
A = {'Alex', 'Tippy', 'Shells', 'Shadow'}
B = {'red', 'blue', 'yellow'}
C = {1,2,3,4}
```

While the set consisting of numbers can be specified as in mathematics, for the remaining sets, we enclose the elements in quotes so as to distinguish them from other names in DML that may denote variables or functions. In particular, we are using strings to represent the elements of sets A and B.

In mathematics, we use the ellipsis notation as a shorthand to avoid cumbersome enumeration of larger sets. It is applicable when we are working with sets of contiguous integers. For instance, 1..100 denotes all the integers from 1 through 100. The same exact syntax can be used in DML as well:

```
X = 1..4
```

denotes the set $\{1, 2, 3, 4\}$.

The following table lists the DML operators that are applicable to list. Note that comparison operators are applicable to all types, so all of them can be used on sets as well.

DML Symbol	Math equivalent	Explanation
`in`	$\in$	Membership check
`union`	$\cup$	Set union
`inter`	$\cap$	Set intersection
`subseteq`	$\subseteq$	Subset operators
`-`	$-$	Set difference
`*`	$\times$	Cartesian product

2.2.1.2 Set builder notation

Another common way to construct sets is to start with a known larger set and use a boolean condition to select elements of interest. In mathematics, this is called the *set builder* notation. Here is an example of a specified using this notation in math, with the corresponding DML definition on the right:

$E ::= \{n^2 \mid n \in \mathbb{N} \wedge (n < 100) \wedge (n \textbf{ mod } 2 = 1)\}$

```
E = {n^2 for n in 0..99 if (n % 2 = 1)}
```

Note that in mathematical notation, we can work with infinite sets and hence we can start with the set $\mathbb{N}$ of all integers and then add a condition "$n < 100$". In contrast, DML only supports finite sets, so we cannot begin with the set of all integers. Instead, we must start with a finite subset, so the definition on the right starts out with `0..99`. The last condition in the mathematical definition can be used as is in DML. Note that we have used parentheses for clarity but they are optional.

Another difference between the set builder notation in mathematics and DML uses is that we include additional keywords such as `for` and `if` so that the DML interpreter can easily parse the definition. Also recall that "`%`" is the DML operator that corresponds to **mod**. Here is another example:

$E ::= \{n \in \mathbb{N} \mid (n \leq 100) \wedge (n^2 - 41n - 40 > 0)\}$

```
E = {n for n in 0..100 if (n^2 - 41*n - 40 > 0)}
```

In the context of programming languages, this way of generating new sets is called *set comprehension*. Note that DML uses the same syntax for set comprehensions as Python.

2.2.2 Lists

Lists are ordered collections of elements. They correspond to *sequences* in mathematics. While sets are enclosed by braces in DML, lists are enclosed in square brackets. The order of listing of elements is significant in lists. As such, `[1,2]` $\neq$ `[2,1]`. Elements can be repeated in lists, and hence `[1,2]` $\neq$ `[1,1,2]`. (Repetition as well as order are meaningless in the case of sets, so `{1,2}` = `{2,1}` = `{1,1,2}`.)

Similar to the ellipsis notation for sets, lists support a `range` operator. For instance, `range(1,4)` corresponds to the list `[1,2,3]`. Note that the list ends just before reaching the higher end of the range. This is different from sets, where the set includes the higher end of the range. We have opted for this difference because the meaning of ellipsis in mathematics is to generate a set that includes both the lower and higher end of the range. It is uncommon to use ellipsis (or a similar operator) to denote sequences, so we opt for a semantics that is consistent with other programming languages.

Note that `range` can take an optional third parameter that specifies the step, which must be a positive integer. For instance:

```
range(1,13,3) = [1, 4, 7, 10]
```

Finally, lists can be specified using *list comprehensions*, just as sets are specified using set comprehensions. The syntax is identical, except for the replacement of braces by square brackets. Here are some examples:

```
[ x for x in {1,2,3,2,1} ]  =  [1,2,3]
[ y for y in range(1,50,2) if y % 7 = 0]  =  [7,14,21,28,35,42,49]
[5*z+1 for z in 1..6]  =  [6, 11, 16, 21, 26, 31]
```

Note that the keyword `in` may be followed by either a set or a list. Also note that the `if` clause is optional. Finally, as shown in the third example, the element need not be just a variable but can be an arbitrary expression. All three points apply to sets as well as lists.

2.2.3 Tuples

In mathematics, new sets can be constructed using the *Cartesian (set) product* operation. For instance:

$$\begin{aligned} A &::= \{1,2,3\} \\ B &::= \{a,b\} \\ A \times B &::= \{(1,a),(1,b),(2,a),(2,b),(3,a),(3,b)\} \end{aligned}$$

DML's tuple operation corresponds to this Cartesian product. Just as Cartesian product can operate on multiple sets at a time, tuples in DML can have any number of components. For instance, if you type the following into DML:

```
x = (1, 'w', 2.0)
```

you get the following response:

```
x:(int, string, real) = (1, "w", 2.0)
```

that indicates that `x` is a tuple that consists of three components of types `int`, `string` and `real` respectively.

Components of a Cartesian product are called *ordered tuples*. Consequently, the order of elements in a tuple is significant. This contrasts with sets, which are unordered. (This means that the order in which set elements are listed is irrelevant, e.g., `{1,2,3}` = `{3,2,1}` = `{2,1,3}`.)

Components of a tuple can be accessed using the indexing operator. With `x` as shown above, `x[0]` is `1`, `x[1]` is `"w"`, and `x[2]` is `2.0`. In keeping with the convention of most programming languages, indices start at zero rather than one.

Instead of accessing tuple components by their index, it is possible to assign names to them. This flavor of tuples are called *records* in DML. For instance, one can define:

```
x={ival=1, sval='w', rval=2.0}
```

This binding creates essentially the same value as the tuple `x` shown earlier, but the components can be accessed using names. For instance, `x.ival` is `1`. In a program, names are more readable than indices and hence it is preferable to use records over tuples. (Internally, DML implements tuples as records that have field names `0`, `1`, `2`, etc.)

Note that Cartesian product requires a minimum of two operands[2] For this reason, tuples with zero components are not permitted. In addition, tuples with a single component are flattened into a simple value. For instance, `(((1)))` becomes the integer `1`. Similarly, `{a={b=7.5}}` becomes a simple real number `7.5`.

[2]The two operands can be the same.

2.2.4 Dictionaries

Dictionaries, also called *associative lists* or *hash tables* in some languages, are a very versatile and efficient data structure frequently used in programs. A dictionary can be viewed as a set of pairs of the form $(key, value)$. Its advantage is that the value associated with a given key can be located efficiently using the index operator. An example dictionary is as follows:

```
x = {'V':'violet', 'I':'indigo', 'B':'blue', 'G':'green',
     'Y':'yellow', 'O':'orange', 'R':'red'}
```

Note that the elements are listed within braces. Within each element, the key precedes the colon operator, while the value follows it. They can both be of arbitrary types. The value associated with a key can be looked up using the index operator, e.g., `x['B']` will return `'blue'`.

While the above notation is convenient for most uses, there can be situations where we want to generate a dictionary from a set of key-value pairs. A predefined function `dict` can do this. For instance, with

```
y = {('V','violet'), ('I','indigo'), ('B','blue'), ('G','green'),
     ('Y','yellow'), ('O','orange'), ('R','red')}
```

`dict(y)` will yield the exact same dictionary as `x`.

In terms of mathematical types, dictionaries correspond to functions. Specifically, a dictionary of type (T_1, T_2) *dict* can represent any function that maps a subset of T_1 to a subset of T_2. For instance, the variable `y` defined above captures a particular function from one set of strings (specifically, `{'V', 'I', 'B', 'G', 'Y', 'O', 'R'}`) to another set of strings (specifically, `{'violet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red'}`).

3 Boolean Formula Satisfiability and Validity

DML supports satisfiability and validity checking on boolean formulas. For instance,

```
valid (bx->by) || (by->bx)
```

returns `true`, indicating that this boolean formula is always true, regardless of whether the variables `bx` and `by` are true or false. (Recall that a boolean formula is *valid* if it evaluates to true for all possible values of the boolean variables occurring in it.) If the formula is not valid, DML will provide an example truth assignment that makes the formula false. For instance,

```
dml> valid ((bx->by) -> (by->bx))
    bx=F, by=T
false
```

You can substitute `bx=false` and `by=true` in the above formula and verify for yourself that the formula `((bx->by) -> (by->bx))` becomes false.

While validity requires the formula to evaluate to true for all values of boolean variables occurring in it, satisfiability addresses a complementary problem: finding at least one truth assignment to the variables that makes the formula true. For instance:

```
dml> satisfiable ((bx->by) -> (by->bx))
    bx=F, by=F
true
```

Once again, by substituting `bx=false` and `by=false`, you can verify for yourself that the formula `((bx->by) -> (by->bx))` becomes true.

If there is no truth assignment that can make the boolean formula true, DML will return the value `false`. In this case, the formula is termed a *contradiction* — a formula that is always false. Note that a contradiction is the opposite of a valid formula. In particular, if the formula F is a contradiction, then $\neg F$ is valid.

DML uses a brute-force search for validity and satisfiability checking: it tries every possible value for each variable occurring in the formula. For relatively small formulas consisting of up to 20 or so variables, this search is fast enough on most computers today.

Note that the argument of `valid` and `satisfiable` may contain some variables that aren't boolean. For those variables, their current values are used during the evaluation of the formula. Any undefined variable is assumed to be boolean, and the procedure iterates through its possible values.

4 Quantified Formulas

DML supports quantified formulas, where the quantification is over finite sets. For instance, the quantified formula

$$\exists a \in \mathbb{N}\ \exists b \in \mathbb{N}\ \exists c \in \mathbb{N}\ \ a^2 + b^2 = c^2$$

representing Pythagorean triples cannot be expressed in DML because it quantifies over an infinite domain. But when the domain is restricted to a finite set, it can be expressed naturally in DML:

```
dml> exists (a in 1..10) exists (b in 1..10) exists (c in 1..10) a^2 + b^2 = c^2
    a=4 b=3 c=5
true
```

Once again, DML uses a brute-force search to check if this formula is true. However, the search follows a random order. This means that if you try the same query again, it will typically return a different example. For instance, `a=3 b=4 c=5`, `a=8 b=6 c=10`, and so on.

For a formula that only contains existential quantifications, DML will provide an example assignment to each variable such that the formula is satisfied. If no such examples exist, it will stop after printing `false`. Similarly, for formulas that only use universal quantification, DML will produce examples that violate the formula, or print `true`, indicating that the formula is true for all possible values of the quantified variables.

For formulas that contain a combination of universal and existential quantification, it becomes more difficult to precisely characterize when examples will be produced. However, the results produced should be self-explanatory. For instance, consider the following quantified formula that asserts that the square of an integer is also an integer:

$$\forall x \in \mathbb{N}\ \exists y \in \mathbb{N}\ \ x^2 = y$$

If we simply place an upper bound on x and y in order to produce a DML formula, we get:

```
forall (x in 1..100) exists (y in 1..100) x^2 = y
```

If we feed this to the DML interpreter, we get:

```
dml> forall (x in 1..100) exists (y in 1..100) x^2 = y
    x=76
false
```

The formula is false! On further reflection, it becomes clear that we were too hasty in constructing a finite-domain version. If x can be as large as 100, then x^2 can be as large as 10000. If we correct the formula to account for this, we get:

```
dml> forall (x in 1..100) exists (y in 1..10000) x^2 = y
    true
```

If we change the order of quantification, i.e., we consider the formula $\exists y \in \mathbb{N}\ \forall x \in \mathbb{N}\ \ x^2 = y$, we get:

```
dml> exists (y in 1..10000) forall (x in 1..100) x^2 = y
    false
```

Finally, note that quantified formulas can be used within functions:

```
dml> fun is_prime(x) = forall (y in 2..sqrt(x)) x % y != 0
dml> is_prime(7)
    true
dml> is_prime(8)
    y=2
false
```

5 Expressions

DML expressions are constructed from variables and values using various operators, built-in functions and user-defined functions. We describe the operators and built-in functions below, while user-defined functions are covered in a subsequent section.

Complex expressions can be broken into a sequence of steps using the `let` expression, e.g.,

```
let x = 3
    y = 4
in
    x*y
```

will print the result `12`.

5.1 Operators

The operators supported in DML are listed below in increasing order of precedence. (Precedence dictates the order of evaluation of operators. Specifically, in the absence of explicit parenthesization to control the order of evaluation, an operator of higher precedence will be evaluated before another operator of a lower precedence. For instance, `2+3*5` will be interpreted as `2+(3*5)` since "`*`" appears lower down in the list and hence has a higher precedence than the "`+`" operator.)

Symbol	Explanation
`,`	Used as separator in various lists
`;`	Terminates statements
`{, }`	Used to construct sets and dictionaries.
`(,)`	Parenthesis, and for constructing tuples
`=`	Operator used in variable and function definitions
`<->`	Logical equivalence ("if and only if")
`->`	Logical implication
`||`	Disjunction ("or" operator)
`&&`	Conjunction ("and" operator)
`!`	Logical negation
`=, ==, !=`	Equality and disequality on all types
`>, <, >=, <=`	Inequality operators on all types
`in, inlist, subseteq`	Membership check and subset operators
`union, ++`	Set union, list/string concatenation
`inter`	Set intersection
`:`	Dictionary element construction
`range, ..`	Range construction
`+, -`	Addition, subtraction/set difference
`*, /, %`	Multiplication/Set cartesian product, division, and modulo
`^`	exponentiation
`-`	Unary minus
`[, ]`	Used to construct lists, and for indexing/selection

5.2 Built-in Functions

Built-in functions are:

- `abs`: Returns the absolute value for a given number (e.g. `abs(-10)` = 10).
- `avg`: Returns the average of elements of a List or Set with numeric members.
- `ceil`: Returns the closest integer that is larger than the given real number (e.g. `ceil(1.2)` = 2).
- `concat`: Variable-arity function that concatenates the input Lists or Sets into a single List or Set
- `count` or `len`: returns the length of a given collection or string.
- `dict`: If given no arguments, returns an empty Dictionary. If given a List or Set of Records with two fields (e.g. `{key="key", val="value"}`), returns a Dictionary with the key having the type of the first field and the

value having the type of the second. If there are any repeated key values, only one of the values will be present in the Dictionary returned with no guaranteed value based on order.

- `floor`: Returns the closest integer that is smaller than the given real number (e.g. `floor(1.2)` = 1).
- `if`: Takes three arguments. The first argument is of boolean type, and is always evaluated. If it evaluates to `true`, then the second argument is evaluated and returned. Otherwise, the third argument is evaluated and returned. (Note that `if` is special in that it does not evaluate all arguments.)
- `insert`: The first argument to this function is a collection. It takes one or two additional arguments, depending on the collection type:
  - If the collection is a set, then the second argument is the value to be inserted into the set.
  - If the collection is a list, this function can take one or two additional arguments. If there is only one additional argument, that is taken as the element to be inserted *at the end* of the list. If two additional arguments are provided, the first of them should specify the insertion position and the second one the value to be inserted.
  - If the collection is a dictionary, two additional arguments are expected: the key and the value to be inserted.
- `join`: If the argument is a single list of strings, returns all the strings joined together. If two arguments `(strlist, separator)` are provided, it returns the strings joined together with the separator string in between.
- `log`: If the argument is a single number, returns the logarithm of that number to the base 10. If two numbers `(n, b)` are provided, it returns the logarithm of `n` to the base `b`.
- `ln`: Returns the natural logarithm of the number (i.e. to the base $e \approx 2.7182$)
- `min`: Returns the smallest element of a List or Set with numeric members.
- `max`: Returns the largest element of a List or Set with numeric members.
- `pow`: This function can take a single set-valued argument and returns the power set of the given set.
- `print`: prints the given input. It can take any number of arguments and prints all of them. Spaces are inserted between these values, and a newline inserted after printing all of the arguments.
- `printr`: is a "print raw" function that prints exactly what is asked for, i.e., it does not add any spaces or newlines on its own.
- `rand`: Returns a randomly generated integer. Can be seeded using `seed`
- `remove`: Removes an item from a set or a dictionary.
- `removeat`: Removes an item at a particular index in a list.
- `round`: Rounds the real number to the nearest integer.
- `seed`: A function that returns no output but "seeds" the pseudo-random number generator with the input integer in order to make the outputs replicable.
- `sortaz`: Sorts the items in an orderable collection in ascending order.
- `sortza`: Sorts the items in an orderable collection in descending order.
- `split`: If the argument is a single string, returns a list of strings which is the parts of the string broken up by whitespace. If two arguments `(string, separator)` are provided, it returns the list of strings split at the separator string instead.
- `sqrt`: Returns the square root of the given number.
- `sum`: Returns the sum of elements of a list or set of numbers.
- `time`: Takes one argument, which can be any DML expression. It is used to measure the time it takes to evaluate this expression. The output is a record with four fields, `elapsed`, `user`, `system` and `val`. The last field is the value of the argument expression, while the the other three corresponds to the three standard measurements you get with UNIX time commands. (Search "man time" for a background on UNIX time measurement.)
- `tojson`: Converts the input number, boolean, list, set, or dict into a valid JSON representation of that type (e.g., sets are converted into lists) and returns that as a string.
- `toset`: Converts the collection to a set.
- `tostring`: Converts the input number, boolean, list, set, or dict into a string representation and returns that string instead of printing it.

- Trigonometric and Hyperbolic functions: DML implements:
  - Trigonometric functions `sin`, `cos` and `tan`,
  - Inverse Trigonometric functions `asin` ($\sin^{-1}$), `acos` ($\cos^{-1}$) and `atan` ($\tan^{-1}$),
  - Hyperbolic functions `sinh`, `cosh` and `tanh`, and
  - Inverse hyperbolic functions `asinh`, `acosh` and `atanh`

  that each take in a single number and return a real number output in SI units like radians.

5.3 Top-level commands

All of predefined functions described above take a comma-separated list of arguments that are enclosed in parenthesis. In contrast, there are two top-level operations use the format *cmd args* where *cmd* names the command to be executed, and *args* is a space-separated list of expressions. We call them as "top-level" operations because they cannot be nested inside any expression.

- `quit` quits the command shell. (Pressing Ctrl+D has the same effect.) It does not take any arguments.
- `load` *file*, which has the same effect as manually typing the contents of *file* at the command prompt. For complex programs, it is easier to write them using your favorite editor and save them in a file. Such a program may be run by loading it from its file.
- Ctrl+C interrupts the current operation and returns to the top level. Sometimes your program goes into an infinite loop, or simply takes too long to complete. By pressing Ctrl+C, this computation is aborted, and DML returns to the top level to accept the next command. In some instances, it may take a second or two to get a response, but anything longer should be considered a software bug.

6 Variable and Function Definitions

Variable definitions consist of a variable name, followed by the "`=`" symbol and then an expression. Variable names (and all other names) in DML must begin with an alphabetic character or an underscore, followed by zero or more alphanumeric characters or underscores.

The effect of a variable definition is to evaluate the expression on the right-hand side and to bind that value to the variable on the left-hand side of the equality operator. If the same variable has been bound to something previously, that old binding is lost and replaced by that of the expression just evaluated.

Function definitions consist of a function name and parameters (also called arguments) on the left-hand side of the "`=`" operator, and an expression on the right-hand side. For instance, the following function will compute the square of its argument:

```
fun f(x) = x*x
```

It can be used in an expression as follows:

```
y = f(3)
```

This statement will cause `y` to be bound to the value of `9`.

An important point about functions is that they must be entirely self-contained: the expression defining a function cannot use variables other than the parameters and any new variables introduced within the function definition. For instance, the following function definition is invalid:

```
x = 3
fun g(y) = x*y
```

While DML does not allow the use of non-local variables in a function body, it does allow the use of previously defined functions.

7 Command completion and editing with `dmlsh`

DML is meant for interactive use. Frequently, there are typographical and conceptual errors in commands typed interactively. Moreover, there is a tendency to rerun commands with small variations. This may reflect a gradual refinement of our approach for solving a problem, or a simple repetition of the same task on multiple data sets. In all of these cases, we need to make small changes to previously typed commands, and it would be very cumbersome

if we had to retype everything all over. `dmlsh` is a command shell that runs on top of DML to help with this task. It provides the following functions:

- *Command history:* Up and down arrow keys can be used to cycle through past commands.
- *Command editing:* After selecting one of the previous commands, forward and backward arrows can be used to access parts of the selected command and edit it using backspace and/or delete keys. (This kind of editing is also applicable to a new command that is currently being entered.)
- *Command completion:* At any point, the tab key may be pressed. This causes `dmlsh` to complete the current word. If there is no unique completion, then a single press of the tab key does not do anything, but two successive presses will list all available completions.

  When function names are completed, `dmlsh` prints parenthesis and commas indicating the position and the expected number of arguments to the function. For functions that take a variable number of arguments, the number of commas printed corresponds to the minimum number of expected arguments.
- *File completion:* Some commands take a file name as an argument, and in that case, completion will look up in the file system to expand the current word. As with command completion, if there isn't a unique completion, the word is left unchanged, or is expanded to the maximal common prefix among all possible completions of the current word; and a second press of the tab key will list all of them. (File completions does not work correctly with file names containing space.)
- *Loading code:* DML code stored in text files can be loaded using the `load` command. Currently, file completion is applicable just to this command.

Here is an example of loading a file `t02` in a subfolder called `test`. Note that `dmlsh` prints the entire file contents first, and then feeds the command one line at a time to DML.

```
dml> load tests/t02
0 . # Implication, equivalence, validity and satisfiability
2 . valid !(x || y) <-> !x && !y
3 . valid (x->y) || (y->z) || (z->x)
4 . (satisfiable (x->y) && (y->x)) && !(valid (x->y) && (y->x))
5 . valid x=y <-> (x<->y)
6 . valid (x->y) -> (x<->y)
7 . valid (x<->y) -> (x->y)
8 . valid (x=x) <-> satisfiable (y<->y) # Produces an error
9 . (valid (x=x)) <-> satisfiable (y<->y) # Works
    Processing commands from 'tests/t02'
    ------------------------------------------------------------------------

dml> # Implication, equivalence, validity and satisfiability
    ------------------------------------------------------------------------

dml> valid !(x || y) <-> !x && !y
    true
    ------------------------------------------------------------------------

dml> valid (x->y) || (y->z) || (z->x)
    true
    ------------------------------------------------------------------------

dml> (satisfiable (x->y) && (y->x)) && !(valid (x->y) && (y->x))
    x=F, y=F
x=T, y=F
true
    ------------------------------------------------------------------------

dml> valid x=y <-> (x<->y)
    true
    ------------------------------------------------------------------------
```

```
dml> valid (x->y) -> (x<->y)
    x=F, y=T
false
    ------------------------------------------------------------------------

dml> valid (x<->y) -> (x->y)
    true
    ------------------------------------------------------------------------

dml> valid (x=x) <-> satisfiable (y<->y) # Produces an error
    Near tokens ‘<->’ and ‘satisfiable’:Error: "Valid" and
    "satisfiable" operators cannot be nested. Use explicit
    quantification instead
    ------------------------------------------------------------------------

dml> (valid (x=x)) <-> satisfiable (y<->y) # Works
    y=F
true
    ------------------------------------------------------------------------

    Done processing commands from ’tests/t02’
    ----------------
```

Currently, file completion applies to just single command that takes a file argument, namely, the `load` command.

There can be instances when `dmlsh` gets into a bad internal state, and behaves erratically. If this happens, exit the shell and restart it. Since your command history is saved into a file, this restart is usually painless. You can also file a bug report.

8 Example Programs

8.1 Illustrative operations on sets

```
# Basic set operations
A = 1..3
B = 4..5
A1 = {2,3,4,4,6,6,6}
A2 = A1 union A
A3 = A1 union B
A1 inter A
A1 inter B
A2 inter A3
A2 - A1
A3 - A
A1 == (A1 - A) union (A1 - B) union ((A inter A1) inter B)
A subseteq A1

# Cartesian product
C = {'T','F'}
D = A*B

# Note that elements of A*B*C are not of the form (a, b, c), but are instead of the form ((a, b), c),
# since A*B*C is parsed as (A*B)*C. A*B*C being interpreted as a set of triples of the form (a, b, c)
# is simply a convention in mathematics.
E = A*B*C
F = (A*B)*(A*C)
print(A.len(), '*', B.len(), '=', D.len())
print(A.len(), '*', B.len(), '*', len(C), '=', E.len())
print('square(', A.len(), '*', B.len(), ')␣=', F.len())
```

```
# Power set

PA = pow(A)
PE = pow({})
PPE = pow(pow({}))
PBC = pow(B*C)
len(PBC)
```

8.2 Boolean formula satisfaction and validity

```
# Implication, equivalence, validity and satisfiability

valid !(x || y) <-> !x && !y
valid (x->y) || (y->z) || (z->x)
(satisfiable (x->y) && (y->x)) && !(valid (x->y) && (y->x))
valid x=y <-> (x<->y)
valid (x->y) -> (x<->y)
valid (x<->y) -> (x->y)
valid (x=x) <-> satisfiable (y<->y) # Produces an error
(valid (x=x)) <-> satisfiable (y<->y) # Works
```

8.3 Quantified formulas

```
# Pythagorean triple

fun pyth_triple(x, y, z) = x*x + y*y = z*z

# Some pythagorean triples exist

exists (x in 1..10) exists (y in x..10) exists (z in y+1..10) pyth_triple(x,y,z)

# Return some pythagorean triples under 100

fun pyth_exists(N) =
    exists (x in 1..N-1) exists (y in x..N-1) exists (z in y+1..N)
           pyth_triple(x,y,z)

pyth_exists(60)
pyth_exists(60)
pyth_exists(60)
pyth_exists(60)

# Return all pythagorean triples under 100

pt = {(x, y, z) for x in 1..60 for y in x..60 for z in y+1..60
     if pyth_triple(x,y,z)}

# Just return the hypotenuse values

zs = sortaz({z for x in 1..60 for y in x..60 for z in y+1..60
                   if pyth_triple(x,y,z)})

print("On␣average,␣each␣hypotenuse␣corresponds␣to", len(pt)/(1e-10+len(zs)),
          "␣distinct␣combinations␣of␣sides")


# Quadruple square

fun pyth_quad(w, x, y, z) = w*w + x*x + y*y = z*z

exists (w in 1..20) exists (x in w..20) exists (y in x..20) \
```

```
          exists (z in y..20) pyth_quad(w,x,y,z)

ptq = {(w, x, y, z) for w in 1..20 for x in w..20 for y in x..20 \
           for z in y+1..20 if pyth_quad(w,x,y,z)}

# Triple cubes

fun triple_cubes(x, y, z) = x*x*x + y*y*y = z*z*z

exists (x in 1..20) exists (y in x..20) exists (z in y+1..20)
       triple_cubes(x,y,z)

# Quadruple cube

fun quad_cubes(w, x, y, z) = (w*w*w + x*x*x + y*y*y = z*z*z)

exists (w in 1..20) exists (x in w..20) exists (y in x..20) \
       exists (z in y..20) quad_cubes(w,x,y,z)

pqc = {(w, x, y, z) for w in 1..20 for x in w..20 for y in x..20 \
           for z in y+1..20 if quad_cubes(w,x,y,z)}
```

8.4 Prime numbers

```
# Simple definition of primality using forall. Note that the percent symbol
# denotes the modulo operation: x % y is the remainder when x is divided by y.

fun primal(x) = forall (y in 2..x-1) x % y != 0

# Here is an alternate version that says exactly the same thing but draws
# attention to the fact a prime is divisible by itself and the number one.
# In addition, it is not executable because y ranges over an infinite domain.
#   fun primal(x) = forall (y in 2..x-1) x%y != 0}

# We can use the above definition of primality within the set generation
# notation (typically called set comprehension in the context of programming
# languages) to generate the set of all prime numbers less than a given N.

fun primes(N) = {x for x in 2..N if primal(x)}

# Use the above definition to print all primes less than 100.

primes(100)

# Let us build a more efficient version of prime number generation using an
# algorithm called Eratosthenes Sieve. It starts with a list of all the numbers.
# It will then successively eliminate multiples of known primes from this list.

fun rm_multiples(S, n) = [s for s in S if s % n != 0 ]

# An interesting property of the method is that at any time, the first number
# remaining in the list will be a prime. So, the algorithm operates by
# outputting the first number in the list, and then proceeding to eliminate
# the multiples of this number from the list. At the end of this step, we are
# again left with a first number in the list that will be a prime.
#
# In essence, each pass over the list produces one prime number.
#
# We implement repeated passes using recursion: the first pass is effected by
# the function rm_multiples given above. Following this step, we apply the
# exact same procedure on what is left in the list.
```

```
fun sieve(S) = if (len(S) == 0)
                  then []
               else sieve(rm_multiples(S, S[0])) ++ [S[0]]

# At the top level, we start things off by generating the initial list and
# then calling sieve to print all the prime numbers in it.

fun sieve_primes(N) = sieve(range(2,N))

# By deault, DML prints the values of any expression or variable binding that
# appears at the top level. However, the size of the output is elided after a
# certain size in order that things don't overwhelm the user and/or scroll off
# the screen. This limit can be changed using the following command.

set_printlen 100000
sieved_primes_under_1000 = sieve_primes(1000)

# Goldbach's conjecture: every even number greater than two can be expressed
# as a sum of two primes. We don't know if it is true for all numbers, but
# we sure can try to verify it for a finite set that isn't too large.

fun evens(N) = range(4, N, 2);
fun goldbach_cond(primes) =
   forall (x in evens(max(primes))) exists (y in primes) \
          x > y && x - y in primes
fun goldbach(N) = goldbach_cond(toset(sieve_primes(N)));

goldbach(100);
goldbach(1000);
goldbach(3000);
```

8.5 N-Queens problem

```
fun abs(x) = if (x < 0) then -x else x

# For an NxN chess board, a board position is a set (or list) of N queen
# positions, each of the form row=n1, col=n2. The following function checks
# to ensure that no two queens in the board position are attacking each other.

fun okpos(pos) = forall (x in pos) forall (y in pos)
    x != y -> x.col != y.col && abs(x.row - y.row) != abs(x.col - y.col)

# Generate all possible positions for a 4x4 chess board and output the subset
# that are OK.

fourq = [[{row=0,col=a}, {row=1,col=b}, {row=2, col=c}, {row=3, col=d}] \
           for a in 0..3 for b in 0..3 for c in 0..3 for d in 0..3
             if okpos([{row=0,col=a}, {row=1,col=b},
                                {row=2, col=c}, {row=3, col=d}])]

# Now, let us try to formulate Nqueens.

fun nqposgen(l) = [{row = i, col = l[i]} for i in range(0,len(l))]

okpos(nqposgen([0, 5, 4, 1, 3, 2]))
okpos(nqposgen([1, 3, 5, 0, 2, 4]))

fun perm(S) = if (S = {})
                 then [[]]
              else [T++[x] for x in S for T in perm(S-{x})]

fun Nq(n) = S = 0..(n-1)
```

```
        Pos = perm(S)
        [ pos for pos in Pos if okpos(nqposgen(pos)) ]


let q3=Nq(3); in print("3 queens has", len(q3), "solutions: ", q3)
let q4=Nq(4); in print("4 queens has", len(q4), "solutions: ", q4)
let q5=Nq(5); in print("5 queens has", len(q5), "solutions: ", q5)
let q6=Nq(6); in print("6 queens has", len(q6), "solutions: ", q6)
#let q7=Nq(7); in print("7 queens has", len(q7), "solutions: ", q7)
#let q8=Nq(8); in print("8 queens has", len(q8), "solutions: ", q8)
```

9 Credits

The DML system was developed by the course instructor with contributions from Samuel Buena, Uma Ganapathy and Dileep Pemmani, with additional help from Hanke Kimm and Jong Kwon Park.