Novel Features of DM

Thomas Colthurst
thomasc@alum.mit.edu

October 12, 2002

Abstract

DM[1] is a language in design with many new and noteworthy features. They are described here with the hope that they may be stolen by designers of other lightweight languages.

1 Syntax

1.1 Parse Zones

Consider the following DM program:

```
[# Hello, user in DM]
master = ['username];
print ["Hello, [$master].\n];
```

It contains four different types of parse zones, each following the "[c...]" form, where "c" is a character that specifies the parse zone type. The type of a parse zone determines how the characters in it are to be turned into a value. In a comment parse zone, started with `[#`, the characters are all ignored. In a shell parse zone, started with `['`, the characters are executed as if on a command line, and the value of the parse zone is the string generated by the command execution.

In a string parse zone, started with `["`, the value of the parse zone is the string made by its characters. As the example shows, parse zones can be nested; here, the string parse zone contains value a parse zone (started with `[$` and parsed as "normal" DM program text). In all such nestings, the inner zones are evaluated first and the surrounding context gets to decide what to do with their values. The string parse zone converts all the values it encounters in inner parse zones into strings, so in the example, a string similar to "Hello, root.\n" is printed.[2]

DM programs can also define their own parse zone types. To do so, they must supply a single function which takes as input a list which alternates between the (possibly empty) string constants found in the parse zone and the (lazily evaluated) values created by any inner parse zones. The function can return a value of any type, and can even call `parse_value`, a built-in function which parses value zones. As an example, a user defined (not built-in) regular expression parse zone is part of the DM standard library, and no special work is needed to allow values, comments, or even other regular expressions to be embedded inside regular expressions.

Other built-in parse zones include the assembly language zone `[a` (which evaluates to a function), the binary file include zone `[b filename]` (which evaluates to a byte vector), the namespace include zone `[i filename]` (which evaluates to a namespace), and the feedback zone `[f` (instances of which are optionally inserted into the program by the compiler to hold program annotations).

DM uses parse zones for at least three different reasons. First, they put into common form a number of syntactic entities every programming language needs, such as comments and string constants. Sec-

[1] Short for Dead Mathematician.

[2] Here are how some of the other parse zones treat their inner zones: the comment parse zone ignores all of its inner zones, the shell parse zone converts them to strings just like the string parse zone, and the value parse zone treats them like values, except for the empty values returned by the comment parse zones, which it ignores entirely.

ond, the nesting of parse zones provides a principled way for denoting the "interpolate this variable into this string" features popular in Perl and other scripting languages. Finally, allowing DM programmers to define their own parse zones enables them to easily create powerful mini-languages which can both be embedded in DM programs and have DM programs (including other powerful mini-languages) embedded into them.

1.2 Lexically Classed Names

In DM, you can tell whether a name is being used as an infix operator just by looking at it. [3] All prefix operators and nullary names start with alphanumeric characters, and all infix name don't. For example, in the "Hello, username" program above, there are two infix operators, namely the naming operator `=` and the sequencing operator `;`, a single prefix operator `print`, and a single nullary identifier `master`.

Lexically designating infix operators allows programmers to easily parse DM programs, even in the context of new user defined infix operators, which DM allows. The meaning of

```
( foo bar @ bleen ) :  zot
```

may be unknown, but any DM programmer should be able to parse it (especially since they know that in DM, prefix operations always bind tighter than infix operators and that parentheses have their normal meaning). Also, lexically designated fixity mostly agrees with standard mathematical and programming usage: `+`, `*`, `/`, `^`, `==`, and `;` are all infix, while `sin`, `log`, `print`, and `return` are all prefix. Unary minus (as in `-x / y`) is a glaring exception, but many programming languages do give it a different name while reserving `-` for the more common infix subtraction. (In DM, unary minus is `neg`).

Name boundaries are also lexically designated. The character set is divided into five classes:

1. not allowed for names (including all six bracketing characters and all whitespace),
2. may start a prefix name characters (alphanumerics),
3. may start an infix name characters (most of the rest),
4. the three "neutral" characters `?`, `$`, and `!` which may not start but may be used in either type of name, and
5. the underscore character `_`.

The underscore works like a neutral character but also switches a name between infix and prefix when put at the beginning:

```
x = _+ (3,4); y = 42 _mod 7;
```

The underscore must be used when naming infix operators; for example,[4]

```
_| = {(rest in) (first in)};
```

defines a pipe operator for shell-like function composition.

Again, these name boundary rules more or less agree with existing mathematical and programming usage[5] while making programs with user defined operators much more intelligible. As before, the meaning of

```
(foo59**?star!  bar)//zoom
```

may be unclear, but it is guaranteed to be the same as

```
_// ( _**? ( foo59, ( star! bar ) ), zoom )
```

At this point, some might wonder whether user definable infix operators are worth all the trouble. Here are the arguments for them: they let programmers create and use a domain appropriate language, they free the language designer from having to draw an artificial line in the sand denoting a certain fixed list of operators (and no others) as infix, the rules taming their use force a desirable consistency on the built-in operators, and finally, they allow the core of a DM interpreter or compiler to be truly lightweight, as many

[3] This feature is not unique to DM; it is also present in the language Cecil, except that in Cecil it does not apply to the built-in syntax.

[4] Lispers may read `first`, `rest`, and "`,`" as "car", "cdr", and "cons", respectively.

[5] Unary minus is again an exception: a number of languages would parse `x*-y` as what DM would call `_* ( x , neg y)` rather than as `_*- (x , y )`. Also, mathematical usage is a bit more sensitive to white space than almost any programming language: `sin x+y` would almost always mean `sin(x+y)` to a mathematician, while `sin x + y` is ambiguous, but leans towards `sin(x) + y`.

of the "built-in" DM operators (including all of the arithmetic operators) can be implemented in (albeit sometimes machine specific) standard libraries.

1.3 Syntactic Uniqueness

DM strives to use every syntactic token in exactly one way. Parentheses are used only for parse grouping (and not, say, also for function calls), and nothing else is used for grouping.[6] All naming in DM is done through the `=` operator; there is no special syntax for naming function values.

Even overloaded operators are required to uniquely represent a given role. `+` is always abelian (`x + y == y + x`) and always distributes over multiplication (i.e. `+` is always ring addition). `x ^ 4` always means the same as `x * x * x * x`. Thus, in DM, `*` and not `+` is used for string concatenation. Section 2.3 gives more details about how this restriction is enforced.

Syntactic uniqueness is nothing more than the usual, universally admired principle of orthogonality imported into the syntactic domain. Still, there is one place where DM flagrantly violates it:[7] a DM standard library defines all sequences of three to eighty consecutive dashes to be the same as `/`, except with a very low precedence, in order to allow expressions like

```
root1 = neg b + discrim
        ---------------
            2 * a      ;
```

2 Semantics

DM's semantics is similar to Scheme's: statically scoped names, latent typing (in which types are associated with values and not necessarily with variables), a single function type, immortal values (i.e. the language and not the programmer handles garbage collection), and first class functions that call by value. Unlike Scheme, all values in DM are immutable[8], and there is a rich type universe. (For programs needing mutable values, DM provides boxed values as in ML.) Several of these descriptions require serious qualification, though, as given below.

2.1 Functions in DM

Here is an example DM function:

```
euclid_gcd =
  { (x,y) = in;
    useIdea LinearRing TYPE_OF x;

    case( ( {x == y}, {x} ),
          ( {x < y}, {euclid_gcd(x, y-x)} ),
          ( {x > y}, {euclid_gcd(x-y, y)} )
        )
  };
```

In DM, functions are delimited by curly brackets; besides the named function `euclid_gcd`, there are in fact six smaller unnamed functions in this example. DM functions receive their argument through the distinguished variable `in`. and return the last value in their body (or more precisely, the value of their body). The six unnamed functions all ignore their input, but are still useful in that they allow `case` to only evaluate `x`, `euclid_gcd(x,y-x)`, or `euclid_gcd(x-y,y)` when needed.

As befits a semi-functional language, DM's syntax allows functions to be defined very concisely. Small functions like `{x}` or `dup = { (in, in) }` need not even waste space giving a name to their input.

2.2 Types in DM

Many languages allow the definition of new types; DM is unusual in that it allows the definition of both new type constructors and new primitive (non-constructed) types.

This flexibility is due to DM's definition of a type as simply a distinguished `sizeof` function which

---

[6] Technically, you could use inner value parse zones for grouping and write things like `[$ a + b ] * c` instead of `(a+b)*c` but this would be ugly and is discouraged. On the other hand, inner value parse zones should be used to introduce then throw away temporary names, as in
`x = [$ y = x*x; z = x + sin(x); y*z/(y+z) ];`
which is the DM's equivalent of ML's "in" operator or Scheme's "let".

[7] Admittedly, a hardcore Lisper might argue that anything beyond parentheses violates syntactic uniqueness.

[8] That's why they're called values.

takes as input the starting address of a value (of that type)'s instantation in memory and returns the number of bytes used. Types are distinguished in the same way that two boxes containing the same value are distinguished:

```
IEEE_DOUBLE = MAKE_TYPE! {8};
INT_8 = MAKE_TYPE! {8};
IEEE_DOUBLE == INT_8
```

is false. Thus parametric type constructors, such as `INT n` (the type of an `n` byte signed integer), must memoize their return values.

Values of new types can be created by the `cast` function, which takes as input a type and a list of bytes, checks to see that the list of bytes is of the right size according to the type, and then returns a value `v` such that

```
v == cast( TYPE_OF v,
           read_mem ((TYPE_OF v) address v)
                    address v )
```

where `read_mem n a` returns a list of the `n` bytes appearing in memory starting at address `a`.

This low level memory manipulation may seem inappropriate for a high level language. Indeed, they will often not be necessary, as most values and types are easily created from standard library constructors that provide for vectors, dictionaries, records, and sets. Still, the fine-grained flexibility has a number of advantages besides the stated power to create new types and type constructors. First, it once again allows features (in this case, types and type constructors) to be moved out of the compiler or interpreter and into a portable and more easily maintained library. Second, it allows DM to be used for systems programming tasks. Finally, it eases the construction of foreign language interfaces: a DM program can fairly easily arrange its data into a form that is acceptable to a C function for input, and then easily convert the function's output into values of standard DM types.

While the type universe of DM is rich, the type hierarchy itself is rather boring: `TYPE_OF TYPE_OF x == TYPE` for all x, and the type of every function is `FUNC`. DM does provide a way to organize similar types, but this is done non-hierarchically, as described in the next section.

2.3 Idea-Oriented Programming

Consider the following code fragment:

```
Semigroup = NewIdea! [l T, _*, semigroup_1 ];

IEEE_DOUBLE_addition_as_Semigroup
  = [n T = IEEE_DOUBLE,
       _* = IEEE_DOUBLE_plus;
       semigroup_1 = IEEE_DOUBLE_zero; ];

IEEE_DOUBLE_addition_as_Semigroup
  _implements Semigroup;
```

It contains two new, but easy to explain, parse zones. The first, `[l`, is the identifier list parse zone; in this example, it would be the same as writing `( ["T], ["_*], ["semigroup_1] )`. The second, `[n`, is the namespace parse zone, which parses and executes its contents just like the value parse zone `[$` does. Rather than returning the value created by the execution, though, it returns the final namespace, which is a map between any names defined and their defined values.

The first line of the example introduces a new "idea," `Semigroup`. `Semigroups` are required to implement the three names, and the other statements in the example show how an idea implementation is specified.

On one level, then, ideas in DM are just collections of names that implementations are required to implement, while implementations are just types, functions, and identifiers bundled into a namespace. It is hopefully clear that neither ideas nor implemenations are the same as types; for one thing, a given type may implement the same idea in multiple ways. (Our example could have been extended to also define a `IEEE_DOUBLE_multiplication_as_Semigroup`).

On another level, though, ideas are placeholders for semantic critera. For example, all `Semigroup` implementations are expected to satisfy certain relations (`*` should be associative, `semigroup_1` should be the identity) for all `x` in `T`). DM never actually checks that these criteria are satisfied, but it does allow you to define separate Semigroup and AbelianSemigroup ideas, and to say that some implementations implement one and not the other.

New ideas can be created from old ones:

```
Ring = Semigroup
       * NewIdea! [1 _+, ring_0, neg];
IOable = Printable * Scanable;
IOableRing = IO * Ring;
```

and DM is smart enough to know that every implementation of an IOableRing is also an implementation of a Ring, an IO, a Printable, and a Scanable.

Actually using ideas and implementations requires a two step dance:

```
register IEEE_DOUBLE_addition_as_Semigroup;

useIdea Semigroup IEEE_DOUBLE;

assert( semigroup_1 * semigroup_1
        == semigroup_1 );
```

When an implementation is registered, the next time useIdea is called with an idea the implementation implements and the type specified by the implementation's required T name, the operators named in the implementation are modified to additionally dispatch on the type T to the implementation's definitions.

This setup has the following properties:

1. All functions in an idea are type polymorphic. This allows * to be used both on IEEE_DOUBLE and on M_STRING (machine string, the type of all values created by string parse zones) in the same block of code, for example. (Functions in DM aren't inherently type polymorphic; this behaviour is created behind the scenes by NewIdea! and useIdea.)

2. On the other hand, ideas often have associated functions which are generic rather than type polymorphic. For example,

```
_- = { useIdea Ring TYPE_OF (first in);
       (first in) + neg (rest in) };
```

3. The nullary names in an idea aren't type polymorphic, because they have no arguments on which to be polymorphic. Thus, the identity of semigroup_1 depends entirely on which implementation was last loaded through a useIdea Semigroup statement.

4. The effects of registers and useIdeas only last until the end of the block in which they occur; this is useful in temporarily overriding registrations:

```
[# Print only x in scientific notation]
[$ register
     IEEE_DOUBLE_scientific_as_Printable;
   print x; ]
```

where it is assumed that print, like euclid_gcd, contains a
useIdea Printable TYPE_OF in;
statement.

5. The separation of implementation registration and loading, which at first might seem artificial, is in fact necessary for functions to be able to express their semantic requirements. If euclid_gcd merely used whatever < and - operators were currently defined, there would be no easy way for it to be sure that they came from the same implementation or that they together satisfied the axioms of a linear ring (such as 1 > 0).

DM's ideas are obviously similar in concept to say Java's interfaces, except that they are inherently type generic and not at all object oriented. In use, the idea oriented programming encouraged by DM is most closely related to the generic programming used for example in C++'s standard template library.

3 Conclusion

DM is a language in design, so there is no working implementation nor even a formal demonstration that the presented features are consistent. Still, it is hoped that the ideas sketched here stimulate thought and interest among other language designers.