Lessons from a DSL where all you have is ranges

John Loughran Colvin

Symmetry Investments

In the beginning

In the beginning

There were variables and functions:

In the beginning

There were variables and functions:

```
alias add = (a, b) => a + b;
auto x = 2;
auto y = 2;
auto z = add(x, y);
```

In the beginning

There were variables and functions:

```
alias add = (a, b) => a + b;
auto x = 2;
auto y = 2;
auto z = add(x, y);
```

And it was … ok.

Just after the beginning

Just after the beginning

There were more variables:

Just after the beginning

There were more variables:

```
alias add = (a, b) => a + b;
auto x0 = 2;
auto y0 = 2;
auto z0 = add(x0, y0);
auto x1 = 3;
auto y1 = 3;
auto z1 = add(x1, y1);
```

Just after the beginning

There were more variables:

```
alias add = (a, b) => a + b;
auto x0 = 2;
auto y0 = 2;
auto z0 = add(x0, y0);
auto x1 = 3;
auto y1 = 3;
auto z1 = add(x1, y1);
```

... and it was starting to feel a bit off

And then

And then

There were more functions:

And then

There were more functions:

```
alias add = (a, b) => a + b;
alias timesBy2 = a => a << 1;
auto x0 = 2;
auto y0 = 2;
auto z0 = add(x0, y0);
auto x1 = 3;
auto y1 = 3;
auto z1 = add(x1, y1);
auto ζ = timesBy2(add(z0, z1));
```

What's the problem?

What's the problem?

Iteration

&

Composition

What a Range?

What a Range?

An aggregate that defines `empty`, `front` and `popFront`

What a Range?

An aggregate that defines `empty`, `front` and `popFront`

Do we have anything? What do we have? Go to the next one.

What a Range?

An aggregate that defines `empty`, `front` and `popFront`

Do we have anything? What do we have? Go to the next one.

```
iota(100)
    .map!(i => i * rand())
    .filter!(i => i % 2)
    .writeln;
```

What a Range?

An aggregate that defines `empty`, `front` and `popFront`

Do we have anything? What do we have? Go to the next one.

```
iota(100)
    .map!(i => i * rand())
    .filter!(i => i % 2)
    .writeln;
```

There are other primitives for going backwards, getting an element by offset, saving the current position.

For C++ programmers, it's like a begin/end pair of iterators.

What's our problem?

What's our problem?

Iteration

&

Composition

Goal

- We wanted to allow people who are not currently programmers to do bulk data processing and glue systems together.
- The usual slice-and-dice work that happens in Excel every day, but without the limitations of Excel and the horrors that grow to work around those limitations.
- We needed a language that was easy to use, hard to abuse and expressed the thought at hand clearly.

Why not just use Python?

Or Equivalent

Why not just use Python?

Or Equivalent

- Imperative programming is the hard part, not the easy part.

Why not just use Python?

Or Equivalent

- Imperative programming is the hard part, not the easy part.
- Mutable state opens up the potential for monstrous code and awful bugs

Why not just use Python?

Or Equivalent

- Imperative programming is the hard part, not the easy part.
- Mutable state opens up the potential for monstrous code and awful bugs
- No proper pipeline programming (unless we effectively re-implement what we want as a DSL inside python)

Why not just use Python?

Or Equivalent

- Imperative programming is the hard part, not the easy part.
- Mutable state opens up the potential for monstrous code and awful bugs
- No proper pipeline programming (unless we effectively re-implement what we want as a DSL inside python)
- These languages weren't designed for trivial interoperability with other systems (but that's another talk…)

What did we do?

What did we do?

- Took code.dlang.org/packages/pegged and created a grammar with variable definitions, arithmetic, array literals etc.

What did we do?

- Took code.dlang.org/packages/pegged and created a grammar with variable definitions, arithmetic, array literals etc.

- Created a Variable type (was minimally wrapped code.dlang.org/packages/taggedalgebraic, now totally custom) supporting some basics like string, delegate, int, Variable[string], Variable[] plus an open-ended variant type.

What did we do?

- Took code.dlang.org/packages/pegged and created a grammar with variable definitions, arithmetic, array literals etc.

- Created a Variable type (was minimally wrapped code.dlang.org/packages/taggedalgebraic, now totally custom) supporting some basics like string, delegate, int, Variable[string], Variable[] plus an open-ended variant type.

- Created a parse-tree-walking interpreter to recursively build Variables to get the result.

What did we do?

- Took code.dlang.org/packages/pegged and created a grammar with variable definitions, arithmetic, array literals etc.

- Created a Variable type (was minimally wrapped code.dlang.org/packages/taggedalgebraic, now totally custom) supporting some basics like string, delegate, int, Variable[string], Variable[] plus an open-ended variant type.

- Created a parse-tree-walking interpreter to recursively build Variables to get the result.

- The next step was going to be getting array expressions really sorted, e.g. a = b + c where all are arrays, including index matching for indexed data.

And then I went on holiday

And then I went on holiday

And then I got ill

And then I went on holiday

And then I got ill

And then I came back…

And everything was different!

And everything was different!

- Added first-class support for ranges.

And everything was different!

- Added first-class support for ranges.
- Wrapped a large chunk of the D standard library.

And everything was different!

- Added first-class support for ranges.
- Wrapped a large chunk of the D standard library.
- A project was being started to try and use the language in an important piece of day-to-day operations.

And everything was different!

- Added first-class support for ranges.
- Wrapped a large chunk of the D standard library.
- A project was being started to try and use the language in an important piece of day-to-day operations.
- Later on, we decided that maybe modules, if/else, scopes, not overwriting live stack frames and so on were also useful features.

SIL Examples

```
alias add = (a, b) => a + b;
alias timesBy2 = a => a << 1;
auto x0 = 2;
auto y0 = 2;
auto z0 = add(x0, y0);
auto x1 = 3;
auto y1 = 3;
auto z1 = add(x1, y1);
auto ζ = timesBy2(add(z0, z1));
```

SIL Examples

```
add = (a, b) => a + b
timesBy2 = a => a * 2
x0 = 2
y0 = 2
z0 = add(x0, y0)
x1 = 3
y1 = 3
z1 = add(x1, y1)
q = timesBy2(add(z0, z1))
```

SIL Examples

```
add = (a, b) => a + b
timesBy2 = a => a * 2
xs = [2, 3]
ys = [2, 3]
zs = zip([xs, ys])
    |> map(p => add(p[0], p[1]))
q = zs
    |> sum
    |> timesBy2
```

Ranges Save the Day

We didn't have much to work with, but phobos ranges and algorithms are great.

```
weeklyClose = readCsvTable("dailyOHLC.csv")
    |> applyToCol("date", parseDates)
    |> byRow
    |> filter(x => x.date.dayOfWeek == Day.friday)
    |> map(x => [x.date, x.close])
    |> tableFromPairs
    |> writeCsv("weeklyClose.csv")
```

Ranges Save the Day

We didn't have table literals, only `mkTable` that returns an empty table and `addEntry`

Ranges Save the Day

We didn't have table literals, only `mkTable` that returns an empty table and `addEntry`

```
tableFromPairs(a) => a
    |> fold(
        (newT, p) => newT
            |> addEntry(p[0], p[1]),
        mkTable()
    )
```

Ranges Save the Day

We didn't have table literals, only `mkTable` that returns an empty table and `addEntry`

```
tableFromPairs(a) => a
    |> fold(
        (newT, p) => newT
            |> addEntry(p[0], p[1]),
        mkTable()
    )
superSecretHedgeFundTable = [
    [“a”, 1],
    [“b”, 2],
    [“c”, 3]] |> tableFromPairs
```

Ranges Save the Day

We didn't have any builtin functions that operated on the values of a table

Ranges Save the Day

We didn't have any builtin functions that operated on the values of a table

```
apply(tIn, func) => tIn |> keyValPairs
    |> fold(
        (tOut, p) => tOut
            |> replaceEntry(p.key,
                func(p.value)),
        tIn
    )

{“a” : 3, “b” : 4} |> apply(x => x * 2)
// gives {“a” : 6, “b” : 8}
```

Ranges Save the Day

No proper dataframes? No problem, e.g.

Ranges Save the Day

No proper dataframes? No problem, e.g.

```
getRow(t, i) => {
    ks = keys(t)
    vs = values(t)
    in zip([ks, vs |> map(v => v[i])])
        |> tableFromPairs
}

{“a” : [3, 4], “b” : [7, 8} |>getRow(1)
// gives {“a” : 4, “b” : 8}
```

Don't be clever

Locals and scopes are quite nice

```
split(hay, needle) => (
        i => [hay[0 : i], hay[i : $]]
    )(hay |> indexOf(needle) |> value)
```

Don't be clever

Locals and scopes are quite nice

```
split(hay, needle) => (
        i => [hay[0 : i], hay[i : $]]
    )(hay |> indexOf(needle) |> value)
```

v.s.

```
split(hay, needle) => {
    i = hay |> indexOf(needle) |> value
    in [hay[0 : i], hay[i : $]]
}
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    return r.map!(x => x * v);
}
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    return r.map!(x => x * v);
}
```

We can easily create lambdas that capture context, (just a `struct` with an `opCall`).

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    return r.map!(x => x * v);
}
```

We can easily create lambdas that capture context, (just a `struct` with an `opCall`).

This is not a problem with capturing by value in lambdas, it's a problem with `map`

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    static struct Callable {
        T v;
        auto opCall(ElementType!R x) {
            return x * v;
        }
    }
    auto c = Callable(x);
    return r.map!c;
}
```

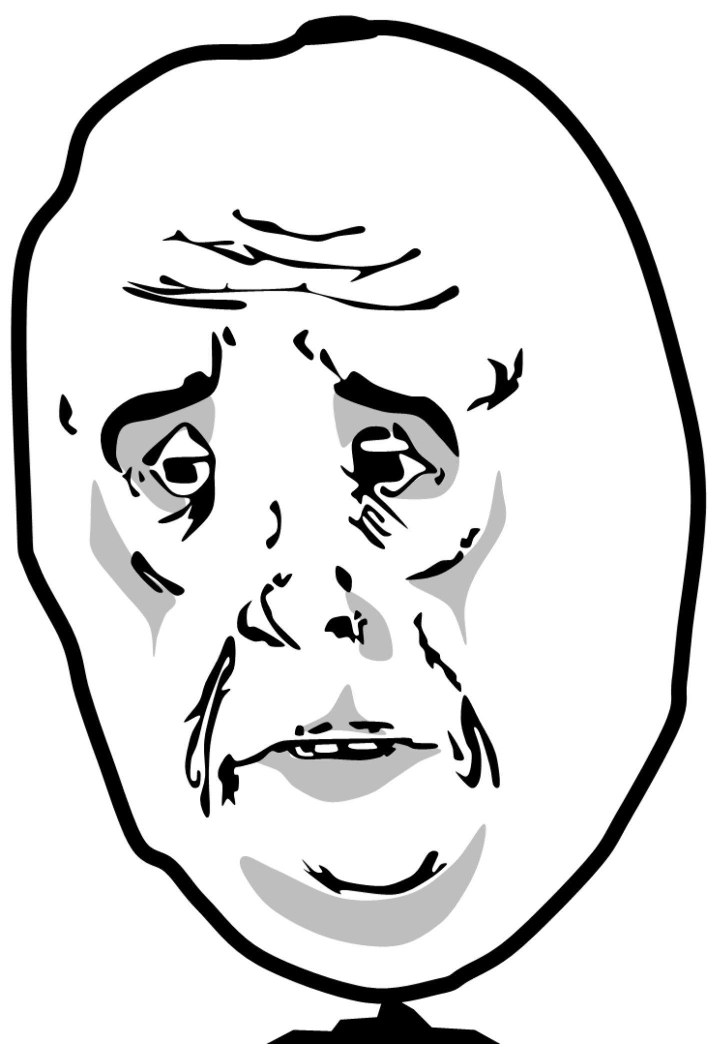
```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    static struct Callable {
        T v;
        auto opCall(ElementType!R x) {
            return x * v;
        }
    }
    return r.map!(Callable(v));
}
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    static struct Callable {
        T v;
        auto opCall(ElementType!R x) {
            return x * v;
        }
    }
    static Callable c;
    c = Callable(v);
    return r.map!c;
}
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
    static struct Callable {
        T v;
        auto opCall(ElementType!R x) {
            return x * v;
        }
    }
    static Callable c;
    c = Callable(v);
    return r.map!c;
}
auto a = r.save.scale(3);
auto b = r.save.scale(4);
assert(a == r.scale(3)); //nope…
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
   zip(r, repeat(v)).map!(p => p[0] * p[1]);
}
```

```
auto scale(R, T)(R r, T v)
if (isInputRange!R
 && is(typeof(r.front * v))) {
   zip(r, repeat(v)).map!(p => p[0] * p[1]);
}
```



What are they bad for?

What are they bad for?

- Good at walking, not good at wandering

What are they bad for?

- Good at walking, not good at wandering
- Good performance is reliable when the code is trivial. Theoretical savings, practical catastrophes

What are they bad for?

- Good at walking, not good at wandering
- Good performance is reliable when the code is trivial. Theoretical savings, practical catastrophes
- Writing your own ranges is really, really interesting.

`std.range.generate`

?

`std.range.generate`?

- I didn't know about it until today.

`std.range.generate`?

- I didn't know about it until today.
- Can't skip, can't stop.

betterGen

```
auto map(alias foo, R)(R r)
{
    return r.betterGen!(R, typeof(foo(ElementType!R.init)),
    (s) { with (s)
    {
        if (input.empty)
            return stop;
        return val(foo(input.front))
            .popInput;
    }});
}
```

betterGen

```
auto filter(alias foo, R)(R r)
{
    return r.betterGen!(R, ElementType!R,
    (s) { with (s)
    {
        if (input.empty)
            return stop;
        auto inFront = input.front;
        if (foo(inFront))
            return val(inFront)
                .popInput();
        return nothing
            .popInput;
    }});
}
```

betterGen

```
// YES THIS IS NONSENSE, I KNOW
auto chunkBy(alias foo = (a, b) => a == b, R)(R r)
{
    return IterState!(R, /*something*/,
    (s)
    {
        if (s.input.empty)
            return s.stop;
        auto inFront = s.input.front;
        return s.val(
            s.input
                .until!(x => !foo(inFront, x)));
    }
}
```

Implicit Conversions

```
auto blah()
{
    if (rand() % 2)
        return null;

    if (auto a = rand() % 2)
        return nullable(iota(3).map!(x => x + a));

}
```

Types of iteration

A commonly described split:

- Internal
- External

Internal Iteration

Internal Iteration

The iteration happens *inside* the code of `forEach` in JavaScript:

```
[1, 2, 3].forEach(x => console.log(x))
```

Internal Iteration

The iteration happens *inside* the code of `forEach` in JavaScript:

```
[1, 2, 3].forEach(x => console.log(x))
```

`sum` in numpy (Python):

```
np.array([1, 2, 3]).sum()
```

Internal Iteration

The iteration happens *inside* the code of `forEach` in JavaScript:

```
[1, 2, 3].forEach(x => console.log(x))
```

`sum` in numpy (Python):

```
np.array([1, 2, 3]).sum()
```

`opApply` in D:

```
struct S {
    int opApply(int delegate(ref int a) dg) {
        foreach (i; 0 .. 5) dg(i);
        return 0;
    } }
foreach (i; S()) writeln(i);
```

External Iteration

External Iteration

The iteration happens *outside* the code of a pair of iterators in C++:

```
std::vector<int>::iterator begin, end;
```

External Iteration

The iteration happens *outside* the code of a pair of iterators in C++:

```
std::vector<int>::iterator begin, end;
```

A generator in python:

```
[x * 5 for x in range(30)]
```

External Iteration

The iteration happens *outside* the code of a pair of iterators in C++:

```
std::vector<int>::iterator begin, end;
```

A generator in python:

```
[x * 5 for x in range(30)]
```

a range of directory entries in D

```
dirEntries("/usr/lib/", "libphobos*.so.*");
```

External Iteration

The iteration happens *outside* the code of a pair of iterators in C++:

```
std::vector<int>::iterator begin, end;
```

A generator in python:

```
[x * 5 for x in range(30)]
```

a range of directory entries in SIL

```
dirEntries(“/usr/lib/”, “libphobos*.so.*”)
```

Which is this?

```
foreach (x; iota(100))
    writeln(x);
```

Or this?

```
auto a = [1, 2, 3];
for (int i = 0; i < N, ++i)
    printf("%i\n", a[i]);
```

Which is this?

```
foreach (x; iota(100))
    writeln(x);
```

```
auto a = [1, 2, 3];
for (int i = 0; i < N, ++i)
    printf(“%i\n”, a[i]);
```

Which is this?

```
foreach (x; iota(100))
    writeln(x);
```

```
auto a = [1, 2, 3];
for (int i = 0; i < N, ++i)
    printf("%i\n", a[i]);
```

They are clearly both internal and external

- The loop is iterating
- The iterable is being iterated

Internal

“Sure, I know how to iterate over my stuff, I even know some different ways, just tell me what you want done and I’ll make it happen”

Great when you know everything you want to do per-element up-front.

Internal

“Sure, I know how to iterate over my stuff, I even know some different ways, just tell me what you want done and I’ll make it happen”

Great when you know everything you want to do per-element up-front.

External

“I have no idea what you want, don’t even try and explain it me. Just tell me when you want me to spit out the next item”

Composable, you can build up the work in pieces

Internal

External

"I have no idea what you want, don't even try and explain it me. Just tell me when you want me to spit out the next item"

Composable, you can build up the work in pieces

Internal

“I iterate things”

External

“I have no idea what you want, don’t even try and explain it me. Just tell me when you want me to spit out the next item”

Composable, you can build up the work in pieces

Internal

"I iterate things"

External

Internal

"I iterate things"

External

"I can be iterated"

Most Ranges are Both

- They iterate a source range (internal)
- They are iterable (external)
- Internal aspect is trivial for `map`, not trivial for e.g. `filter` or `cache`

OMG! Who cares?

OMG! Who cares?

- Internal iteration is a closed model

OMG! Who cares?

- Internal iteration is a closed model
- External iteration is composable

OMG! Who cares?

- Internal iteration is a closed model
- External iteration is composable
- This is the same pattern as many things in D: allowing choices to be pushed further and further up the call stack.

OMG! Who cares?

- Internal iteration is a closed model
- External iteration is composable
- This is the same pattern as many things in D: allowing choices to be pushed further and further up the call stack.
- This is also the unix philosophy. Do one thing and do it well.

OMG! Who cares?

- Internal iteration is a closed model
- External iteration is composable
- This is the same pattern as many things in D: allowing choices to be pushed further and further up the call stack.
- This is also the unix philosophy. Do one thing and do it well.
- Everything else is someone else's problem.

How many things does this function do?

```
double[] vecMul(double[] a, double[] b)
in (a.length == b.length)
{
    auto r = new double[](a.length);
    r[] = a[] * b[];
    return r;
}
```

How many things does this function do?

```
void vecMul(double[] a, double[] b, double[] r)
in (a.length == b.length)
in (r.length == b.length)
{
    r[] = a[] * b[];
    return r;
}
```

How many things does this function do?

```
auto vecMul(double[] a, double[] b)
in (a.length == b.length)
{
    return zip(a, b)
        .map!(t => t.rename!("elA", "elB"))
        .map!(p => elA * elB);
}
```

This effect is fractal

What if `with` was an expression?

```
iota(1000)
    .enumerate
    .map!(expand!((index, value) => index + value))
```

```
iota(1000)
    .enumerate
    .map!(p => with(p) index + value)
```

Symmetry

INVESTMENTS

Come work at Symmetry Please.

Symmetry
INVESTMENTS

Come work at Symmetry Please.

Now.

Come work at Symmetry Please.

Now.

Please.

Symmetry
INVESTMENTS