Metaprogramming and Data Engineering

A View From The Trenches

Dexter Taylor, Principal, Binary Machines LLC
dtaylor@binarymachines.io

Metaprogramming, also referred to as generative programming, comprises a family of programming methods in which program logic in one language is automatically generated by a specification written in a higher-level language. Looked at one way, the term is almost too broad to be useful, because all high-level languages do exactly this. A C compiler automates the tedium of writing target-specific assembly code; a Java compiler and a Python interpreter both generate bytecode for their respective virtual machines; and so on.

But when we talk about metaprogramming, we usually mean something more specific. The "higher-level language" is usually not a general-purpose programming language, but rather a DSL (domain-specific language) whose idioms relate more closely to the problem at hand. Object-relational mapping frameworks are a very common example of generative programming; a configuration script abstracts away the code required to marshal between objects created in the application code's runtime and the corresponding structures in the database runtime.

Broadly speaking, there are two types of metaprogramming, which we shall refer to as **lexical** and **behavioral**. In lexical metaprogramming, a source code module is explicitly generated (for example, by populating a template or walking a syntax tree) by some program outside of it. Behavioral metaprogramming, by contrast, is intrinsic to the target module; a routine may consult a configuration file in order to determine, say, how many times to execute a for loop or which parameters to pass to a constructor.

Lexical and behavioral programming methods may be used together within the scope of a project; in fact, they can complement each other. One of our mainstay toolsets at Binary is the SNAP framework (Small Network Applications in Python), a generative library for creating HTTP services using Flask. Here is part of a sample initialization file for a SNAP service: it is written in YAML.

```
 globals:
             bind_host:                   127.0.0.1
             port:                        6000
             debug:                       True
             logfile:                     test.log
             project_directory:           $PROJECT_HOME
             transform_function_module:   project_transforms
             service_module:              project_services
             preprocessor_module:         project_decode
…
  transforms:
             test:
                 route:             /ping
                 method:            GET
                 input_shape:       default
                 output_mimetype:   application/json
```

The Snap utility `routegen` uses **lexical** metaprogramming to generate a Flask main application module by processing the config file data through a template. Here is a portion of the generated code:

```
@app.route('/ping', methods=['GET'])
def ping():
    try:
        if app.debug:
            # dump request headers for easier debugging
            log.info('### HTTP request headers:')
            log.info(request.headers)
        input_data = {}
        input_data.update(request.args)
        transform_status = xformer.transform('test',
                                  core.convert_multidict(input_data),
                                  headers=request.headers)
        output_mimetype = xformer.target_mimetype_for_transform('test')
        # additional code omitted
```

Elsewhere in the generated code, we see that in the Flask application kickoff logic, the bind address and port also come from the initialization file:

```
if __name__=='__main__':
        #
        # If we are loading from command line,
        # run the Flask app explicitly
        app.run(host='127.0.0.1', port=6000)
```

So simply by defining a “transform” (the core metaphor in a Snap service) in a higher-level language, we generate the low-level Flask boilerplate for an endpoint which services that transform and sends inbound request data to the appropriate logic module. By specifying the bind address and port in our higher-level language, we generate the application logic necessary to start our service with the correct address and port.

Elsewhere in SNAP, we use **behavioral** metaprogramming to achieve a simple form of dependency injection. In our initfiles we can specify ServiceObjects, which are long-running singletons that initialized on application startup:

```
service_objects:
        postgres:
                class: PostgreSQLServiceObject
                init_params:
                        - name: database
                          value: testdb
                        - name: schema
                          value: public
                        - name: host
                          value: localhost
                        - name: port
                          value: 6432
                        # additional params omitted
```

Then, provided we write canonical ServiceObjects (a ServiceObject constructor takes a single `**kwargs` parameter), we are able to automate our calls to their constructors by calling `initialize_services(…)`, in which every constructor is passed a dictionary of parameters taken from our YAML config file.

```
from snap import snap, common

yaml_config = common.read_config_file(init_filename)

services = common.ServiceObjectRegistry(snap.initialize_services(yaml_config))
```

(At runtime, a ServiceObjectRegistry is passed to each user-defined function that services a SNAP endpoint, so that having registered a ServiceObject named "postgres" in our initfile, we can retrieve the fully initialized object by name and use it in our application logic.)

```
postgres_svc = service_obj_registry.lookup('postgres')
```

Thus far we have demonstrated simple metaprogramming techniques applied to the building of HTTP APIs -- but there are other software engineering disciplines to which metaprogramming is not merely relevant, but essential.

Once they have cleared their initial design phase, REST API's are relatively stable. By contrast, in the field of data engineering we confront highly dynamic problems: that is, problems whose parameters are subject to frequent change. Analytics requirements are often ill-defined at the outset and may evolve significantly over time. Inputs often deviate from specified data formats. The specified formats themselves may change. We may be asked to read data from multiple sources, and the list of those sources may be updated at very short notice. The level of performance required by a system may increase dramatically, also at very short notice. In short, professional data engineers are asked to design and build software solutions which can evolve very quickly, while remaining operationally correct and keeping costs under control.

These problems are compounded by the fact that unlike product-side engineers, data engineers typically do not ship customer-facing features. The consumers of our work product are almost always internal. This means that stakeholders have an even lower tolerance for updates and features which take a long time to deliver, because those features are used by either (a) automated systems, or (b) people who are not customers, and therefore whose satisfaction is not a common metric of company success.

Under these conditions, metaprogramming is a vital weapon in the data engineer's arsenal. The key advantages of a generative approach are:

- It lets us prototype solutions rapidly
- It allows us to reconfigure existing assets to meet evolving specifications without incurring the risk of introducing defects into core logic
- It allows engineers to use a parent language that more closely models business concerns, enabling engineers and stakeholders to collaborate more closely

Here we should note that generative programming is not cost-free; there are tradeoffs. It is not a silver bullet that will solve all our problems, nor is it a top-down solution that can be adopted after the fact. Direct, hands-on programming of low-level code assets is still a necessity, and will be for the foreseeable future; but for a generative approach to succeed, those assets must be designed from the ground up to allow dynamic reconfiguration and reuse. That design process requires insight, creativity, and the disciplined attention of experienced programmers. Put another way, it costs slightly more upfront than a naïve approach. Nonetheless, we have found that it is nearly always a worthwhile investment.

Let's examine a very simple test case for metaprogramming as applied to a foundational data engineering task: ingestion. We often start with raw data in CSV format:

```
firstname,lastname,email,orderid
John,Smith,jsmith@address,0986831
Jane,Doe,jdoe@address,3726612
Bob,Jones,bjones@address,57621089
```

In the best-case scenario, the ingest target is a table (in-memory or relational) which has the same field names as the source – but we almost never encounter the best case. More often the target combines:

- fields which have the same business meaning as the source, but are named differently
- fields which are not present in the source, but must be looked up or computed

To handle the first case, we can adopt a naïve approach of explicitly mapping source to target fields based on their order in the row. If we look at ingesting this file as a one-off problem, this approach can be tempting, especially if time is short. But if we look at this as a specific example of a general case, we might take the opportunity to solve an entire family of problems. We can write some tooling.

We start by noticing that Python lets us scan any CSV file containing a header and turn each record into a dictionary where the keys are the column names:

```
import csv
raw_records = []
with open(‘file.csv’) as csvfile:
	reader = csv.DictReader(csvfile)
	for record in reader:
		raw_records.append(record)
```

after which we can issue `records[0][‘firstname’]` to get the first name of the first customer in the list.

Once we have dictionary type records created, it should be clear that mapping those records to new ones with the same values but different keys, is a simple operation:

```
field_name_map = {
	‘firstname’: ‘first_name’,
	‘lastname’: ‘last_name’,
	‘email’: ‘email_address’,
	‘orderid’: ‘order_number’
}

new_records = []
for record in  raw_records:
	mapped_record = {}
	for raw_record_key, value in record.items():
		new_key = field_name_map[key]
		mapped_record[new_key] = value
```

after which `new_records` holds the transformed data (with only the field names altered).

If we look at the lines-of-sight for this code, we can see that initializing the `key_map` dictionary is a relatively uniform stretch where all we are doing is loading pairs of strings into memory. Repetitive actions along a line of sight in source code are often a good indication that we can abstract something out of the code and into a config file. If we were to write a text file containing

```
Firstname: first_name
Lastname: last_name
```

And so on, then we could start to automate the mapping process by:

- loading the init file
- for each line in the file, creating an entry in the field_name_map dictionary where <key> is the token before the colon and <value> is the token after it

as follows:

```
with open(<initfile>) as f:
        for line in f:
                tokens = line.split(':')
                key = tokens[0]
                value = tokens[1]
                field_name_map[key] = value
```

If we add the ability to pass the initfile name to our program as a parameter, we then have a script which is capable of handling most of the source-to-target field name mappings we are likely to encounter – and we can change its behavior without modifying code.

This takes care of the first case: ingesting to a record where the same fields are present in both the source and the target, but the field names differ. The second case – accounting for computed or looked-up values -- requires a bit more design. To handle this case, we first observe that our fundamental task – our core verb, if you will -- is mapping. We then postulate that *any core verb which is sufficiently complex becomes a noun*. That is, it is no longer sufficient to say that **we are mapping** from source to target; instead, **we need a structured data type called a mapping** to describe what is happening. Then we design our structured data to suit the particular case.

So: what is a mapping, for our purposes? It is an object which contains:

- a source type (as opposed to a source field). That is: when looking at the data source, we no longer assume that the data we need is in the source. We acquire data differently, depending on the source type.

- An optional source field name, allowing us to short-circuit lookups when the field is present in the source record
- a target field name
- a reference to some number of functions capable of generating computed fields; that is, fields not present in the source record. Because we need to call different functions depending on the field we are transforming, this will look like a Python dictionary where the keys are field names and the values are functions.

Here is a nested dictionary representing a mapping from a source field called `FIRSTNAME` to a target field called `first_name`:

```
lookup_mappings = {
        'first_name': {
                'source_type': 'record',
                'source_key': 'FIRSTNAME'
        }
…
}
```

Given the above dictionary, if we are trying to generate the `first_name` field in our target record, we simply look up the mapping (the key itself is the target field name), which specifies that the value is present in the source record under the key 'FIRSTNAME'.

On the other hand, if we need to look up some data not present in the source, we can specify the following:

```
lookup_mappings = {
        …
        'customer_status': {
                'source_type': 'lookup',
                'source_key': 'lookup_customer_status'
        }
        …
}
```

Here, the mapping object stored under the `customer_status` key tells us that the data for that field must come *not* from a source field, but from an action we will call a lookup. We can stipulate in our design that value of the `source_key` field will be the name of a (user-defined) lookup function.

As it happens, the Binarymachines codebase features a variation on just this pattern. The datamap.py module (in the Mercury repo) uses a YAML file to drive the loading and dynamic transformation of CSV files. An operator can – without touching source code – transform a given record from a file into an in-memory record with some or all of its fields renamed; and that same operator, at the cost of a single user-defined function per field, can add any number of calculated fields to the transformed record.

We have shown two different generative designs applied to tasks which appear (and perhaps are) relatively trivial. Nonetheless, these tasks are consequential in the aggregate, because we are called upon to perform them many times within the scope of a project, and each time the requirements are subtly different. In our own practice, we have found that generative tools and methods allow us to respond to dynamic and even chaotic circumstances with greater speed and agility than would otherwise be possible.

Whether or not the investment of time and effort needed for a generative solution is worth it in a particular case, is an open question; and there may be no single correct answer. What we are advocating is a way of looking at software projects which takes into account not only the "big wins" – i.e., long-term project goals -- but also the small wins that can come when talented developers apply metaprogramming to the everyday, seemingly trivial tasks that, taken over time, can sap project momentum and developer bandwidth.

Dexter Taylor

Principal, Binary Machines