gcl Documentation

Release 0.6.9

Rico Huijbers

Nov 21, 2017

Contents

Contents:

CHAPTER 1

Introduction

GCL is a declarative modeling language that can be dropped into any Python project. It supports dictionaries with name-value pairs, all the basic types you'd expect, lists, includes, and methods for abstraction.

1.1 Vision

The goal for GCL is to be a modeling language with lots of expressive power, intended to make complex configurations extremely DRY. Behavior is not part of the goal of the language; behavior and semantics are added by scripts that interpret the GCL model.

Scoping rules have been designed to make single files easier to analyze and to predict their behavior (for example, scoping is not dynamic but all external imports have been declared). This analyzability is good for larger projects, but this necesitates some syntactical overhead that may not make the project worthwhile for small configs split over multiple files. Alternatively, don't split up into multiple files :).

1.2 Why not use JSON?

JSON is good for writing complex data structures in a human-readable way, but it breaks down when your config starts to become more complex. In particular, JSON lacks the following:

- No comments, making it hard to describe what's going on.
- No expressions, so there are no ways to have values depend on each other (e.g., `instances_to_start = expected_tps / 1000`.
- No abstraction, which makes it impossible to factor out common pieces of config.
- All the double quotes I have to type make my pinkies sore! :(

1.3 Alternatives

GCL may not be for you. These are some other popular choices that fill the same space:

- JSON: already mentioned above. Not so nice to write, and because of lack of expressive power encourages copy/paste jobs all over the place.
- TOML: simple and obvious. Doesn't seem to allow abstraction and reuse though.
- UCL: looks and feels a lot like GCL, but the difference with GCL is that in typing `section { }`, in UCL the *interpreter* gives meaning to the identifier `section`, while in GCL the model itself gives meaning to `section`. Also, the macro language doesn't look so nice to me.
- Nix language: subconsciously, GCL has been modeled a lot after Nix, with its laziness and syntax. Nix' purpose is similar (declaring a potentially huge model that's lazily evaluated), though its application area is different. Nix uses explicit argument declaration and makes tuples nonrecursive, whereas in GCL everything in scope can be referenced.

CHAPTER 2

Language Basics

2.1 Basic syntax

GCL is built around named tuples, written with curly braces:

```
{
  # This is a comment
  number = 1;
  string =  'value';  # Strings can be doubly-quoted as well
  bool =  true;       # Note: lowercase
  expression = number * 2;
  list = [ 1, 2, 3 ];
}
```

The top-level of a file will be parsed as a tuple automatically, so you don't write the braces there. Semicolons are considered separators. They may be ommitted after the last statement if that aids readability.

The basic types you'd expect are supported: strings, ints, floats, bools and mapped onto their Python equivalents. Lists are supported, but can't really be manipulated in GCL right now.

2.2 Expressions

```
a = 1 + 1;
b = 'foo' + 'bar';
c = 80 * '-';
```

GCL has an expression language, looking much like other languages you're used to. The evaluation model is mostly borrowed from Python, so things you expect from Python (such as being able to use + for both addition and string concatenation).

2.3 Calling functions

```
inc(1)
```

Function application also looks the same as in Python. There's currently no way to define functions in GCL, but you can invoke functions passed in from the external environment.

```
inc 1
```

If a function only has one argument, you can omit the parentheses and simply put a space between the function and the argument.

2.4 Accessing values in tuples

```
tuple = {
  foo = 3;
};

that_foo = tuple.foo;
```

Periods are used to dereference tuples using constant keys. If the key is in a variable, tuples can be treated as maps (functions) to get a single key out:

```
tuple = {
  foo = 3;
}

that_foo1 = tuple('foo');

which_key = 'foo';
that_foo2 = tuple(which_key);
```

2.5 Including other files

```
http = include 'library/http.gcl';
server = http.Server {
    port = 8080;
}
```

External files can be included with the built-in `include()` function. The result of that expression is the result of parsing that file (which will be parsed as a tuple using the default environment). Relative filenames are resolved with respect to the *including* file.

2.6 'if' expressions

Expressions can also include conditions, using the `if` statement:

```
allow_test_commands = if stage == 'alpha' then true else false;
```

```
# Of course, since these are booleans, the above could also be written as:
allow_test_commands = stage == 'alpha';
```

2.7 List comprehensions

Lists can be manipulated using list comprehensions:

```
[ x * 2 for x in [1, 2, 3, 4, 5] if x % 2 == 0 ]
```

2.8 Tuple composition

As a special case, a tuple can be applied to another tuple, yielding a new tuple thats the merge of both (with the right tuple overwriting existing keys in the left one).

This looks especially convenient when A is a reference and B is a tuple literal, and you use the paren-less function invocation:

```
FooApp = {
  program = 'foo';
  cwd = '/tmp';
}

my_foo = FooApp {
  cwd = '/home';
}
```

`my_foo` is now a tuple with 2 fields, `program = 'foo'` (unchanged) and `cwd = '/home'` (overwritten).

This makes it possible to do abstraction: just define tuples with the common components and inherit specializations from them.

2.9 Parameterized tuples

Because tuple elements are lazily evaluated (i.e., only when requested), you can also use this for parameterization. Declare keys without giving them a value, to signal that inheriting tuples should fill these values:

```
greet = {
  greeting;
  message = greeting + ' world';
};

hello_world = greet { greeting = 'hello' }
```

If `message` is evaluated, but `greeting` happens to not be filled in, an error will be thrown. To force eager evaluation (to try and catch typos), use `eager()` on a tuple.

2.10 Accessing inherited values

Normally in a tuple composition, variables that you set are completely replaced with the new value you're setting. Sometimes you don't want this; you may want to take an existing object or list and add some values to it. In that case, you can refer to the "original" value (values to the left of the current tuple inside the composition) by referring to a tuple called `base.`. For example:

```
parent = {
    attributes = {
        food = 'fast';
        speed = 'slow';
    }
};
final = parent {
    attributes = base.attributes {
        speed = 'fast';
    }
};
```

2.11 Identifiers

Identifiers should start with a letter, may contain the special characters `-` and `:`, but may not *end* in those characters (and because we're all programmers, _ counts as a letter ;). For arbitrary identifiers, quote them with ‘‘‘‘.

Valid identifiers:

```
hello
hell0
_hello
hello-world
hello:world
he110-1ord      # This is NOT a subtraction
```

The following identifiers need quoting:

```
`hello:`
`world-`
`1-1ello`
```

CHAPTER 3

Scoping rules and their consequences

This section describes the scoping rules that GCL is built on, the design decisions that have been made and the consequences of these decisions. At first glance, GCL might require you to type "too much", or ask you to type things in weird places during tuple composition, but this has been done for good reason: to keep GCL models easy to reason about, even when they grow large.

3.1 Lexical Scope

References in GCL are lexically scoped, where each file starts a new scope and each tuple forms its own subscope.

"Lexically scoped" means that only variables that are declared in the current tuple, or any of its parent tuples, are visible in expressions.

For example:

```
blue = '#0000ff';

knight = {
    armor = 'chain_mail';
};

lancelot = knight {
    favorite_color = blue;  # Visible

    helmet = armor;  # Not visible!
};
```

Note that in the `lancelot` tuple, the assignment of `armor` to `helmet` will fail, because even though a value for `armor` is mixed in from the `knight` tuple, that value is not *lexically* visible!

Behaving like this is the only way to avoid "spooky action at a distance". Let's assume GCL did *not* behave this way, and we would always take the value from the current closest enclosing tuple. Then what would happen if someone else, a colleague, inadvertently added a key named `blue` onto the `knight` tuple?

```
blue = '#0000ff';

knight = {
    armor = 'chain_mail';
    blue = 'smurf';
};

lancelot = knight {
    favorite_color = blue;  # Whoops! I wanted '#0000ff' but I got 'smurf'!
};
```

Especially when the tuples invoved are far apart, such as in different files, these effects become nearly impossible to see and debug. Lexical scoping prevents that.

3.2 Valueless keys are input parameters

To be able to refer to a variable, an expression needs to be able to "see" it. This means that it must be in the same tuple or an enclosing tuple of the one the expression is in.

If you want to pass a variable then from one tuple to another, or one file to another, the receive file must have an "empty declaration" of that variable somewhere. For example:

```
# http.gcl
server = {
    dirname;
    www_root = '/var/www/' + dirname;
};
```

And use it as follows:

```
# main.gcl
http = include 'http.gcl';
pics_server = http.server {
    dirname = 'pics';
};
```

A file behaves like a tuple, so if you need to refer to the same value a lot in a file, you can make it a file-level parameter as well:

```
# http.gcl
port;

server = {
    host = '0.0.0.0';
    bind = host + ':' + port;
};
```

And use it as:

```
# main.gcl
https = include 'http.gcl' { port = 443 };
pics_server = https.server {
    ...
};
```

In this example, we make `https` an instantiation of the `http` library with a variable pre-bound. This is common pattern in large GCL models, and can be thought of as "partial application" of a collection of tuples.

As you can see, the downside of the design decision of lexical scoping is that you need to type more, because you need to declare all the "empty variables" that you're expecting to be using. On the plus side, you know *exactly* what you're referring to, and tuples that are you are going to be mixed into can not affect the binding of your variables in any way.

3.3 The 'inherit' keyword

Let's say you want to copy a variable from an outer tuple onto a particular tuple under the same name. Let's say you want to write something like this:

```
base_speed = 3;
motor = {
    base_speed = base_speed;  # Recursive reference
    speed = base_speed * 2;
};
```

First of all, you may not need to do this! If you just wanted the variable `speed` set to the correct value, there's no need to copy `base_speed` onto the `motor` tuple. You can easily refer to the `base_speed` variable directly.

If you still want to copy the variable, the code as written won't work. `base_speed` on the right refers to `base_speed` on the left, whose value is `base_speed` on the right, which refers to `base_speed` on the right, and so on.

To solve this, you can do one of two things: rename the variable, or use the `inherit` keyword (which copies the variable from the first outer scope that contains it while ignoring the current scope).

So either do:

```
base_speed = 3;
bspd = base_speed;
motor = {
    base_speed = bspd;
    speed = base_speed * 2;
};
```

Or do:

```
base_speed = 3;
motor = {
    inherit base_speed;
    speed = base_speed * 2;
};
```

CHAPTER 4

Using GCL files from Python

You now know enough GCL to get started. Using the library looks like this:

```
import gcl
from gcl import util

# Load and evaluate the given file
model = gcl.load('myfile.gcl')

# This gives you a dict-like object back, that you can just index
print(model['element'])

# Translate the whole thing to a Python dict (for example to convert to JSON)
dict_model = util.to_python(model)

import json
print(json.dumps(dict_model))
```

`to_python` will respect the visibility of keys as specified in the schema.

4.1 Bindings variables from the script

Add bindings to the top-level scope from the evaluating script by passing a dictionary with bindings to the `load` function:

```
gcl.load('file.gcl', env={'swallow': 'unladen'})
```

This is also how you add custom functions to the model.

4.2 Finding nodes by location: GPath

Sometimes you want to select specific values out of a big model. For that purpose, you can use GPath, which is a hierarchical selector language to select one or more values from a model.

GPath queries look like this:

```
# Select node by name
name.name.name

# Select multiple nodes
name.{name1,name2}.name

# Select all nodes at a given level
name.*.name

# List indices are numbers in the path between square brackets
name.[0].name
```

The command-line tools use GPath for selecting nodes from the model. Using GPath in your own script looks like this:

```
import gcl
from gcl import query

model = gcl.load('python.gcl')

q = query.GPath([
    '*.favorite_color',
    'lancelot',
    ])

results = q.select(model)

# A list of all values found
print(results.values())

# A list of (path, value) tuples of all values found
print(results.paths_values())

# A deep copy of all selected values into a dict
print(results.deep())
```

4.3 Finding nodes by criteria: TupleFinder

For some reason, I keep on using GCL in scripts where I have big collection of "things", and I want to do an action to each particular thing. Since this is a common usage pattern, there are some standard classes to help with that.

You have option to search the entire model for all tuples that have a particular key (`HasKeyCondition`, and another useful pattern is to do an additional level of dispatch on the value of that key), or for all tuples (or elements) that are in lists with particular key names anywhere in the model (`InListCondition`).

If the found tuples contain references to one another, the `TupleFinder` even has the ability to order tuples by dependency, so that tuples that are depended upon come earlier in the list than the tuples that depend on them (unless there is a cyclic reference).

```
import gcl
from gcl import query

obj = gcl.load(model.gcl')

finder = query.TupleFinder(query.HasKeyCondition('type', search_lists=True))
finder.find(obj)

# All matched objects are now in finder.unordered
print(finder.unordered))

# We'll do an ordering based on dependencies
finder.order()

if finder.has_recursive_dependency():
    print('Some nodes have a recursive dependency!')
    print(finder.find_recursive_dependency())
else:
    # Nodes got moved to finder.ordered
    print(finder.ordered)
```

4.4 Thread safety

Currently, GCL evaluation is *not* thread safe. Unfortunately we need to store some global state to track evaluations so we can detect infinite recursion inside the model.

Let me know if this poses a problem for you, we can make it into threadlocals without much issue.

CHAPTER 5

Command Line Tools

5.1 gcl-print

While editing GCL model files, it's helpful to look at the results of evaluating that model. `gcl-print` can evaluate and print a (subset of) a model.

```
$ gcl-print python.gcl

+- blue              => '#0000ff'
+- knight
|  +- armor          => 'chain_mail'
+- lancelot
|  +- armor          => 'chain_mail'
|  +- favorite_color => '#0000ff'
|  +- helmet         => <python.gcl:10: while evaluating 'armor' in '    helmet =␣
↪armor;', Unbound variable: 'armor'>
```

`gcl-print` accepts GPath selectors.

5.2 gcl2json

GCL can be used as a preprocessor for a complicated JSON model, which can then be processed using more standard tools. `gcl2json` loads GCL model and spits it out in JSON format.

```
$ gcl2json python.gcl

{"blue": "#0000ff", "knight": {"armor": "chain_mail"}, "lancelot": {"armor": "chain_
↪mail", "favorite_color": "#0000ff"}}
```

`gcl2json` accepts GPath selectors.

CHAPTER 6

Schemas

When working in a larger project, when multiple people are involved, as many automatic sanity checks should be added to a code base as possible. Static typing, or its cousin in data specifications, schemas, are one of the tools we have to reduce mistakes and frustration.

Especially when the GCL is going to be processed in a dynamically typed language like Python where errors can be delayed for a long time (and writing explicit type checks is cumbersome and often omitted) validating the input data as quickly as possible makes sense.

6.1 Use cases

Typically, we want to guard against the following two scenarios:

- Missing (required) fields; and
- Values of incorrect types provided.

Schemas annotations are also used for automatic extraction of relevant data: a common use of GCL is as a preprocessor for generating JSON. However, typically you'll have additional fields in your GCL tuples that are used for abstraction, which you don't want to have end up in the JSON. We'll use the schema to extract only the relevant fields from your objects.

6.2 Specifying a schema

Schemas are specified in GCL itself. Typically, you'd put them on tuples are that are designed to be mixed in with other tuples, to prevent them from being misused. For example, you can specify the types of input keys, or force downstream users to provide particular keys.

6.2.1 Scalars

Schema specification for scalars looks like this:

```
Human = {
    hands : required int;
    fingers = 5 * hands;
};

lancelot = Human {
    hands = 2;
};
```

When mixing the tuple `Human` into another tuple, a value for `hands` must be specified, and it must be an integer. Failing to provide that key will throw an error when the tuple is accessed. The following built-in scalar types are defined:

```
string
int
float
bool
null
```

6.2.2 Lists

Specifying that a key must be a list looks is done by specifying an example type using `[]`, optionally containing a schema for the elements:

```
Human = {
    names : required [string];
    inventory : [];
};

lancelot = Human {
    names = ['Sir', 'Lancelot', 'du', 'Lac'];
    inventory = ['armor', { type = 'cat'; name = 'Peewee' }];
};
```

6.2.3 Tuples

To specify that a key must be a tuple (even a tuple with a specific type), specify an example tuple with the expected schema, OR even just refer to an exising tuple with the intended schema:

```
Human = {
    name : required string;

    father : Human;
    mother : Human;
    children : [Human];
};

lancelot = Human {
    father = { name = 'King Bang' };
    children = [galahad];
};

galahad = Human {
    father = lancelot;
```

```
    mother = { name = 'Elaine' };
};
```

6.3 Controlling key visibility on exports

Sometimes you're building a model that you want to export to another program. A typical example would be to use GCL as a preprocessor to have a deep model with lots of abstraction, "compiling down" to a flat list of easy to process objects stored in a JSON file (for example for a single page web app).

Some keys in tuples will be intended for the final output, and some are just to input keys for the abstraction mechanisms. By marking a key as `private`, it will not be exported when calling `to_python`.

For example, given the model from before:

```
Human = {
    hands : private required int;
    fingers = 5 * hands;
};

lancelot = Human {
    hands = 2;
};
```

`hands` is only an implementation detail. We're actually only interested in the number of fingers that Lancelot has. By running `gcl2json` on this model:

```
$ gcl2json knights.gcl lancelot

{"lancelot": {"fingers": 10}}
```

`hands` is nowhere to be found!

CHAPTER 7

Generating library documentation

If you use GCL to model complex domains, you'll want to add abstractions in the form of new tuples to capture commonalities. You'll probably also want to share these abstractions with your team members. But how do they figure out what kinds of abstractions exist, and how they're supposed to be used?

For that purpose, GCL supports *generated documentation* à la JavaDoc.

7.1 Prerequisites

GCL comes with a tool that can extract documentation to ReSTructured text files (`.rst`), which are intended to be used with Sphinx. You'll need Sphinx on your machine to generate these docs.

7.2 Writing doc comments

Doc comments start with `#.`. The first empty line-separated block will be treated as a title, and blocks indented with 4 spaces will be treated as code samples. Doc comments apply to member declarations in tuples.

For example:

```
#. Greeting generator
#.
#. This is a greeter which greets whatever name you give it.
#.
#. Example:
#.
#.     john_greeter = Greeter {
#.         who = 'john';
#.     }
Greeter = {
    #. The name of the person to greet.
    who;
```

```
    #. The greeting that will be produced.
    greeting = 'Hello ' + who;
}
```

7.3 Special documentation tags

Add tags into the doc comment to influence how the documentation is generated. Tags are prefixed with an `@` sign and must occur by themselves on a line in the comment block, much like JavaDoc. The tags that are currently supported are:

- `@detail`, hide this member from the documentation.
- `@hidevalue`, don't show the default value for this member in the documentation. By default, non-tuple values are shown.
- `@showvalue`, do show the default value for this member in the documentation. By default, tuple values are not shown.

7.4 Generating the documentation

The tool that generates the documentation is `gcl-doc`. Example invocation:

```
gcl-doc -o doc_dir lib/*.gcl
```

This generates an `index.rst` file plus a file per GCL file into the `doc_dir` directory. You should then run sphinx on that directory (either by adding in a `conf.py` file with some config, or by passing all interesting arguments on the command line):

```
sphinx-build -EC -Dmaster_doc=index -Dhtml_theme=classic doc_dir doc_dir/out
```

CHAPTER 8

Functions in GCL

This document contains a list of all the functions that come in the standard distribution of GCL. They're defined in `gcl/functions.py`.

8.1 List of standard functions

8.1.1 `fmt(format_string, [env])`

Does string substitution on a format string, given an environment. `env` should be a mapping-like object such as a dict or a tuple. If `env` is not specified, the current tuple is used as the environment.

Example:

```
host = "x";
port = 1234;
address = fmt '{host}:{port}';
```

8.1.2 `compose_all(list_of_tuples)`

Returns the composition of a variable list of tuples.

8.1.3 `eager(tuple)`

Turn a lazy GCL tuple into a dict. This eagerly evaluates all keys, and forces the object to be complete. This will also force a schema evaluation of all keys.

8.1.4 `flatten(list_of_lists)`

Flatten a list of lists into a single list.

8.1.5 has(tuple, key)

Return whether a given tuple has a key, and the key has a value.

8.1.6 join(list, [separator])

Combine a list of string using by separator (defaults to a single space if not specified).

8.1.7 path_join(str, str, [str, [...]])

Call Python's `os.path.join` to build a complete path from parts.

8.1.8 split(str, [separator])

Split a string on a separator (defaults to a single space if omitted).

8.1.9 sorted(list)

Return `list` in sorted order.

8.1.10 sum(list)

Sums a list of numbers.

8.2 Custom functions

You can define new functions for use inside GCL (in fact, you can bind arbitrary values to any identifier) by passing in new bindings for the initial environment, using the keyword argument `env` when calling `gcl.read`:

```
import gcl
import string

my_bindings = { 'upper': string.upper }

object = gcl.loads('yell = upper "hello"', env=my_bindings)
print(object['yell'])
```

CHAPTER 9

Syntax Highlighting

- Vim syntax definitions available: https://github.com/rix0rrr/vim-gcl

CHAPTER 10

Why should I care about GCL?

I'd like to take you through the design motivations of GCL as a configuration language. In particular, I'm going to show what the advantages are compared to writing JSON by hand.

10.1 A gentle note of warning

The goal of GCL is to *reduce repetition* in a complex JSON configuration. However, in trying to tame that complexity, we have had to introduce *some* syntactical and cognitive overhead. This means that if your config is tiny, the fancy features of GCL may not be a win in your particular situation. Of course, you can still just use it as a less-noisy, more friendly-to-type JSON variant :).

10.2 Whither JSON?

First off, why are we competing with JSON? JSON is not a *configuration* format. It's a *data serialization* format. It exists so computer programs can exchange structured information in an unambiguous way.

However, it has a couple of things going for it, that make it a popular choice for configuration files:

- JSON has been popularized by the web, so there are libraries to parse and produce it in virtually every language.
- It directly maps to an arbitrarily complex in-memory data structure. Lists, dicts and scalars are enough to make any type of data structure representation you want, in a more compact way than, say, INI files.

The advantages that JSON has are *ubiquity* and *sufficient* expressive power. Not necessarily *great* expressive power, but enough to get by.

In this document, I'll hope to sell you on the fact that GCL is equally good, or better. (Supposing availability in your environment, obviously, which right now means Python and some shell tools)

10.3 Readability/writability

As noted before, JSON is a serialization format. Is has not necessarily been specced to be convenient to write and maintain by people.

10.3.1 Visual noise and pinky stress

First off, the symbols. GCL does away with a bunch of the visual noise and the shifted characters you have to type:

```
{
    "key": "value",
    "other": "value"
}
```

Versus:

```
{
    key = 'value';
    other = 'value';
}
```

Looks very similar, right? The differences are:

- No quotes `"` around the keys. Fewer characters to type.
- `'` and `"` are both allowed to quote strings, but the `'` doesn't require holding the shift key.
- `=` is used to separate value from key instead of `:` (again, no shift key required).
- `;` is used to separate fields instead of a `,`. There's no difference in keystrokes, but note that the `,` is *not allowed* after the last line in JSON, whereas it is allowed (but optional) in GCL. This makes it less likely that you introduce a syntax error while adding or reordering lines.

These tiny differences add up to make GCL more comfortable to write. At least, I've noticed a lot less stress on my pinkies :). And because there's less visual noise on the quoted keys, it's easier to read as well (especially given the syntax highlighting you can find at vim-gcl).

10.3.2 Comments

In a complex configuration, you're more likely that not wanting to add comments telling the people after you why things are the way they are. JSON has not been designed for this, and there is no way to add comments to your JSON configuration file.

There are some hacks, such as adding a `comment` field to your dictionary, which will then be ignored by the processing script:

```
{
    "machines": 500,
    "access_rights": "pedro",
    "comment": "As many machines as we have data shards. Pedro is the guy running the␣
↪analysis."
}
```

In contrast, GCL supports actual comments in the file:

```
{
    machines = 500;           # We have this many shards
    access_rights = 'pedro';  # In charge of the analysis
}
```

10.4 GCL Extensions

At this point, you can already use GCL as a more readable JSON. All the same value types are afforded: dictionaries, lists, strings, numbers and booleans. As a bonus, the files are slightly easier to write.

However, GCL also affords some abstraction capabilities that make it more expressive than just JSON:

10.4.1 Includes

If you have a very large configuration file, you may want to split it up. Includes give you this ability:

```
frontend = include 'frontend.gcl';
```

The included file will always be resolved relative to the file the include is being done from, so you don't have to worry about the script's working directory.

10.4.2 Expressions

In GCL it's possible to use expressions anywhere a value can appear. So you can do something like:

```
cores_per_machine = 4;  # This is true for all our machines

task = {
    jobs = 100;
    machines = jobs / cores_per_machine;
};
```

As you can see, it's possible to refer to keys OUTSIDE of the current scope as well. You can also use this to build strings:

```
region = 'us-east-1';

photo_bucket = { name = region + '-photos' };
mail_bucket = { name = region + '-mails' };
```

Or, slightly nicer to read, use the `fmt()` function to replace `{placeholders}` inside your strings with the values of keys from the current environment:

```
region = 'us-east-1';

photo_bucket = {
    name = fmt('{region}-photos');
};
mail_bucket = {
    name = fmt '{region}-mails';  # You can leave out the ()
};
```

For functions of 1 argument, you can leave out the parentheses for even less visual noise.

10.4.3 Tuple composition

The big distinguishing feature of GCL, however, is *tuple composition*. If you put 2 (or more) tuples next to each other, they will be merged together to form one tuple. We also call this *mixing in* a tuple. For example:

```
{ a = 1 } { b = 2 }
```

Yields a combined tuple of `{ a = 1; b = 2 }`. Of course, by itself this is not very interesting. The neat thing is that you can override values in a tuple while composing, AND by giving names to tuples, you can make easily-recognizable abstractions. Returning to our earlier example of `task`:

```
cores_per_machine = 4;

# By convention, we use capitalized names for 'reusable' tuples
Task = {
    jobs = 1;        # Default
    machines = jobs / cores_per_machine;
};

small_task = Task {
    jobs = 100;      # Overridden value
};

large_task = Task {
    jobs = 4000;     # Overridden value
};
```

Afterwards, `small_task.machines` is `25`, and `large_task.machines` is `1000`. As you can see, the calculation for `machines` is inherited from `Task` each time, evaluated with their own value for `jobs`.

In the previous example, we had a default value for `Task.jobs`, which could be overridden (or not). It's also possible to *not* give any value for keys we're going to use in an expression, to force the users to fill it in:

```
region;
Bucket = {
    id;
    name = region + '-' + id;
};

# Set 'id', get 'name' for free!
photo_bucket = Bucket { id = 'photos' };
mail_bucket = Bucket { id = 'mails' };

# We can still just set 'name' to bypass the the logic
music_bucket = Bucket { name = 'cool-music' };
```

In this case, we encoded the fact that a bucket name always consists of the region and some other identifier in the `Bucket` tuple. We added another field to fill in the part of the name that changes, and set that when we mix in the tuple.

> Interesting note: actually, the syntax for tuple composition is exactly the same as for function application! You might also put parentheses around the second tuple, as an argument. However, the space-separation looks cleaner, don't you think?
>
> In fact, if you squint really hard at it, you might also consider that a tuple is a function like in normal programming languages. The declared keys without values are its parameters, keys with values are parameters with default arguments, and a composited tuple is the result of evaluating the function against a set of arguments.

10.5 Principle: don't repeat yourself!

The guiding principle behind all of these mechanisms is simple, and it's ancient: Don't Repeat Yourself!

Every piece of policy, every (changeable) decision, should exist in only once place. That has two advantages:

- If the rule ever needs to change, it's easy to do it in one place! If your naming scheme ever changes from `{region}-{id}` to `{component}-{region}-{id}`, you only need to make that change in one place, instead of search-replace through a bunch of files.
- Deviations from the norm stand out more. If every `Bucket` uses the `id = '...'` mechanism to have their `name` derived, *except one because its name needs to follow a different scheme*, you can tell at a glance that a particular bucket is different from the rest and you need to pay more attention. Ideally, that would be preceded by a comment line telling you WHY it's different :).

10.6 Rule: you can always tell where a variable is coming from

I want to highlight this design decision, as it leads to a trade-off that trades some (visible) verbosity for future (not-so-visible) safety. I want to highlight it because, while you're writing some magic incantation, you may be tempted to curse my name for the extra effort you have to put in. Rest assured, there is a reason for this.

The rule is that you always want to be able to decide *what key you are referring to in an expression from looking at a piece of code*. No client using your tuple should be able to change the meaning of any of the keys without you being aware beforehand.

Let me illustrate with a (silly) example involving 2 files:

```
# tasklib.gcl

cpm = 4;  # Cores per machine

Task = {
    jobs = 1;
    machines = jobs / cpm;
};
```

```
# tasks.gcl

lib = include 'tasklib.gcl';

ad_task = lib.Task {
    ads = 100000;
    jobs = ads;

    cpm = 0.01;  # Cost per mille
    total_cost = cpm * (ads / 1000);
};
```

You may see the problem already; in the instantiation of `Task.machines`, what should the expression `cpm` refer to? You would *expect* it to refer to the key `cpm` from its global scope, which has a defined meaning. But all of a sudden, someone else mixed in a key that's *also* called `cpm`! If we were to use the simple rule that we would take the key from the innermost tuple that has it, we would be referring to the 'costs per mille' value instead of the 'cores per machine' value that we intended to.

What's more, we wouldn't be able to tell by looking at any of the pieces of config in isolation that this could happen! And all clients would have to be aware of all variables that exist in the enclosing scopes of a tuple, because it cannot

use the same names for any of its variables! That would lead to some very annoying debugging sessions.

So instead, we have the rule that names in expressions will only bind to keys that are actually *visible in the code* from the point of view of the expression (either in the tuple itself or one of the enclosing tuples). *New keys* that are mixed in later on will be ignored.

This means, slightly annoyingly but very safe, that you have to declare all keys you're going to be expecting or using from your 'sibling' tuples.

So you've already seen this pattern, a reusable tuple expecting a parameter:

```
Lyric = {
    object;
    phrase = fmt "There's a {object} in my bucket";
};

first = Lyric { object = 'hole' };
```

But mixing goes both ways, if a reusable tuple defines some subtuple that a specialization might want to use:

```
Lyric = {
    object;
    phrases = {
        complaint = fmt "There's a {object} in my bucket";
        salutation = 'Dear Liza';
    };
};

first = Lyric {
    object = 'hole';

    phrases;
    song = fmt '{phrases.complaint}, {phrases.salutation}, {phrases.salutation},
↪{phrases.salutation}';
    formal_letter = fmt '{phrases.salutation}, {phrases.complaint}. Regards, Henry.';
};
```

In this case, the right-hand tuple of `first` had to declare `phrases` as a value-less key to bring it *into scope*, so that it can refer to it in expressions. Without this declaration, it wouldn't know to get the value of `phrases` from a tuple that's mixed-in on the side, but would have to go and look for it in an enclosing scope (where no such key exists).

10.7 Influences on script interpreting GCL models

GCL affords a lot of expressivity, allowing to you keep config files textually simple, while (in essence) "generating" big flat objects when all of the abstractions are evaluated away.

This has an influence on the scripts that you write to parse your config. Since you can a do bunch of stuff in GCL, the script doesn't have to implement mechanisms to keep configs maintainble by humans.

- For example, to have both generic configs with "template holes" and then a separate set of values that fill the holes with string substitution.
- Also, the script writer doesn't need to think of what would be useful abstractions (as the config writer can notice them while writing the config, and abstract them out immediately);
- And the script writer doesn't need to invent additional mechanisms to override the default abstractions when users inevitably run into a situation that doesn't fit the mold.

Instead, the script will typically operate on arrays of big fat, flat objects that have every value needed directly available in the dictionary to do whatever it is the script needs doing. This makes writing the script at lot easier.

As an example of something else the script writer doesn't have to deal with: GCL contain a tool called `gcl-print` to print an exploded config file. Helpful to config writers.

The flip side of the possible complexity is that GCL configs are actually code, and should be treated as such [^1]. That means that naming and documenting all the convenient "abstractions" are just as important in the config as they would be in regular code.

A schema language, to assist with this, will be forthcoming soon. Good ideas are very welcome!

[^1]: But then again, I'm of the opinion that ALL config should be treated as part of code. JSON files and INI files have just as much of an implicit contract with their interpreting script as GCL, which should be documented just as well.

CHAPTER 11

Expressive Power

It's the eternal problem of a DSL intended for a limited purpose: such a language then gets more and more features, to gain more and more expressive power, until finally the language is fully generic and any computable function can be expressed in it.

> "In a heart beat, you're Turing complete!" – Felienne Hermans

Not by design but by accident, GCL is actually one of those Turing complete languages. It wasn't the intention, but because of the abstractive power of tuples, lazy evaluation and recursion, GCL actually maps pretty closely onto the Lambda Calculus, and is therefore also Turing complete.

Having said that, you should definitely not feel encouraged to (ab)use the Turing completeness to do calculations inside your model. That is emphatically *not* what GCL was intended for. This section is more of an intellectual curiosity, and should be treated as such.

11.1 Tuples are functions

Tuples map very nicely onto functions; they can have any number of input and output parameters. Of course, all of this is convention. But you can see how this would work as I define the mother of all recursive functions, the Fibonacci function:

```
fib = {
    n;
    n1 = n - 1;
    n2 = n - 2;
    value = if n == 0 then 0
       else if n == 1 then 1
       else (fib { n = n1 }).value + (fib { n = n2 }).value;
};

fib8 = (fib { n = 8 }).value;
```

And then:

```
$ gcl-print fib.gcl fib8

fib8
21
```

Hooray! Arbitrary computation through recursion!

11.2 A more elaborate example

Any time you need a particular function, you can inject it from Python, *or* you could just write it directly in GCL. Need `string.join`? Got you covered:

```
string_join = {
    list;
    i = 0;              # Hey, they're default arguments!
    sep = ' ';

    next_i = i + 1;
    suffix = (string_join { inherit list sep; i = next_i }).value;
    my_sep = if i > 0 then sep else '';

    value = if has(list, i) then my_sep + list(i) + suffix else '';
};

praise = (string_join { list = ['Alonzo','would','be','proud']; }).value;
```

We make use of the lazy evaluation property here to achieve readability by giving names to subparts of the computation: the key `suffix` actually only makes sense if we're not at the end of the list yet, but we can give that calculation a name anyway. The expression will only be evaluated when we pass the `has(list, i)` test.

11.3 Multi-way relations

Because all keys are lazily evaluated and can be overridden, we can also encode relationships between input and output parameters in both directions. The *caller* of our relation tuple can then determine which value they need. For example:

```
Pythagoras = {
    a = sqrt(c * c - b * b);
    b = sqrt(c * c - a * a);
    c = sqrt(a * a + b * b);
}
```

Right now we have a complete relationship between all values. Obviously, we can't evaluate any field because that will yield an infinite recursion. But we *can* supply any two values to calculate the remaining one:

```
(Pythagoras { a = 3; b = 4}).c     # 5

(Pythagoras { a = 5; c = 13}).b    # 12
```

11.4 Inner tuples are closures

Just as tuples correspond to functions, nested tuples correspond to closures, as they have a reference to the parent tuple at the moment it was evaluated.

For example, we can make a partially-applied tuple represents the capability of returning elements from a matrix:

```
Matrix = {
    matrix;

    getter = {
        x; y;
        value = matrix y x;
    };
};

PrintSquare = {
    getter;
    range = [0, 1, 2];

    value = [[ (getter { inherit x y }).value for x in range] for y in range];
};

my_matrix = Matrix {
    matrix = [
        [8, 6, 12, 11, -3],
        [20, 6, 8, 7, 7],
        [9, 83, 8, 8, 30],
        [3, 1, 20, -1, 21]
    ];
};

top_left = (PrintSquare { getter = my_matrix.getter }).value;
```

11.5 Let's do something silly

Let's do something very useless: let's implement the Game of Life in GCL using the techniques we've seen so far!

Our GCL file is going to load the current state of a board from a file and compute the next state of the board–after applying all the GoL rules–into some output variable. If we then use a simple bash script to pipe that output back into the input file, we can repeatedly invoke GCL to get some animation going!

We'll make use of the fact that we can `include` JSON files directly, and that we can use `gcl2json` to write some key back to JSON again.

Let's represent the board as an array of strings. That'll print nicely, which is convenient because we don't have to invest a lot of effort into rendering. For example:

```
[
  "............x....",
  "...x.............",
  "....x.......xxx..",
  "..xxx.......x....",
  "............x...",
  "................",
  "................",
```

```
  "...x.x..........."
]
```

First we'll make a function to make ranges to iterate over.

```
# (range { n = 5 }).value == [0, 1, 2, 3, 4]
range = {
    n; i = 0;

    next_i = i + 1;
    value = if i < n then [i] + (range { i = next_i; inherit n }).value else [];
};
```

Then we need a function to determine liveness. We'll expect a list of chars, either 'x' or '.', and output another char.

```
# (liveness { me = 'x'; neighbours = ['x', 'x', 'x', '.', '.', '.'] }).next == 'x'
liveness = {
    me; neighbours;

    alive_neighbours = sum([1 for n in neighbours if n == 'x']);
    alive = (me == 'x' and 2 <= alive_neighbours and alive_neighbours <= 3)
        or (me == '.' and alive_neighbours == 3);
    next = if alive then 'x' else '.';
};
```

On to the real meat! Let's find the neighbours of a cell given some coordinates:

```
find_neighbours = {
    board; i; j;

    cells = [
        cell { x = i - 1; y = j - 1 },
        cell { x = i;     y = j - 1 },
        cell { x = i + 1; y = j - 1 },
        cell { x = i - 1; y = j     },
        cell { x = i + 1; y = j     },
        cell { x = i - 1; y = j + 1 },
        cell { x = i;     y = j + 1 },
        cell { x = i + 1; y = j + 1 }
    ];

    chars = [c.char for c in cells];

    # Helper function for accessing cells
    cell = {
        x; y;

        H = len board;
        my_y = ((H + y) % H);
        W = len (board my_y);
        char = board (my_y) ((W + x) % W);
    }
};
```

Now we can simply calculate the next state of the board given an input board:

```
next_board = {
    board;
```

```
    rows = (range { n = len board }).value;
    value = [(row { inherit j }).value for j in rows];

    row = {
        j;

        cols = (range { n = len board(j) }).value;
        chars = [(cell { inherit i }).value for i in cols];
        value = join(chars, '');

        cell = {
            i;
            neighbours = (find_neighbours { inherit board i j }).chars;
            me = board j i;
            value = (liveness { inherit me neighbours }).next;
        };
    };
};
```

We've got everything! Now it's just a matter of tying the input and output together:

```
input = {
    board = include 'board.json';
};

output = {
    board = (next_board { board = input.board }).value;
};
```

That's it! We've got everything we need! Test whether everything is working by running:

```
$ gcl2json -r output.board game_of_life.gcl output.board
```

That should show the following:

```
[
  "....x...........",
  "............x....",
  "..x.x.......xx...",
  "...xx.......x.x..",
  "...x.............",
  "................",
  "................",
  "................"
]
```

Hooray, it works!

For kicks and giggles, we can turn this into an animation by using `watch`, which will run the same command over and over again and show its output:

```
$ watch -n 0 'gcl2json -r output.board game_of_life.gcl output.board | tee board2.
↪json; mv board2.json board.json'
```

Fun, eh? :)