HAL
open science

Domain-Specific Languages in Scientific Computing

Konrad Hinsen

To cite this version:

Konrad Hinsen. Domain-Specific Languages in Scientific Computing. Computing in Science and Engineering, 2018, 20 (1), pp.88-92. 10.1109/MCSE.2018.011111130 . hal-01966145

HAL Id: hal-01966145

https://hal.science/hal-01966145v1

Submitted on 16 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Domain-specific languages in scientific computing

If you have been following developments in software engineering over the recent years, you have probably noticed that the term DSL (domain-specific language) has become a minor buzzword in that field. You may have concluded that this is a hot new idea that is certainly not ready for application in real life. What I want to show in this article is that computational scientists (and others) have been using DSLs for decades and will continue to do so. What is new is not DSLs per se, but the name and the attention given to them.

As the name says, a DSL is first of all a language. In the context of computing, this means a formal language, i.e. a language that can be processed mechanically by algorithms. The definition of a formal language consists of two parts: *syntax* defines which byte sequences are valid elements in the language, and *semantics* assigns a meaning to each valid element. For example, the syntax of HTML (HyperText Markup Language) defines start tags (e.g. `<h1>`) and end tags (e.g. `</h1>`), and prescribes that they must occur as matching pairs, whereas HTML semantics says that the text between `<h1>` and `</h1>` is a top-level heading.

The qualifier "domain-specific" indicates that a language was designed specifically for a restricted application domain, as opposed to general-purpose or broad-spectrum languages that aim at being useful in many application domains. HTML is a DSL, because it was designed to represent Web documents and nothing else. Python is an example of a general-purpose language, because you can encode almost any kind of digital information in Python.

Given that there are general-purpose languages like Python, what's the point of more limited languages like HTML? Why don't we just store Web documents as Python scripts, using the `ElementTree` module for example? There are several good reasons. To start with, the same Web document would be longer and more complex when written in Python than it is in HTML. Web documents would thus be more difficult to read and write for human authors. Moreover, a Web browser would have to know the entire Python language, and come up with some way to detect if the Python script presented to it really is a Web document. It can't just run the script and see what happens, because the script might be malicious and delete all the files on the computer. From a security point of view, a limited language that doesn't allow malicious operations is a clear advantage. And for a language intended to be processed by many application programs, simplicity makes everybody's life easier.

As this example shows, a DSL is (1) more than, (2) less than, and (3) different from a general-purpose language. It is more in providing domain-specific functionality that a general-purpose language would have to acquire via a library. It is less in not providing superfluous functionality that makes the information more difficult to process and at worst could become dangerous. It is different from a general-purpose language in being optimized for a narrow range of information types right from the start, allowing a more concise representation.

To end the theoretical part of my discussion of DSLs, let me add that the distinction between "domain-specific" and "general-purpose" is more one of intended use than of technical characteristics. As an example, consider the TeX language that many scientists use for writing their publications. Its intended purpose is being a domain-specific language for typesetting, but technically it has all the required properties for a general-purpose language, in particular Turing-completeness. TeX can therefore be (ab-)used for other applications, such as the simulating a Mars rover (http://sdh33b.blogspot.fr/2008/07/icfp-contest-2008.html). Exploring the limits of a language has its merits beyond the intellectual challenge of winning a contest, in particular when it comes to detecting security problems. However, I still consider TeX a DSL rather than a general-purpose language, in spite of the possibility of using it for Mars rover simulations.

The two examples I have given until now, HTML and TeX, are both related to electronic documents, and that application domain has many more DSLs to offer. HTML is complemented by other Web-standard DSLs such as MathML for formulas or SVG for vector graphics. More recently, lightweight markup languages such as Markdown or reStructuredText have become very popular. And at the printing-press end of the publishing chain, we find DSLs such as PostScript and PDF.

At this point I can almost hear protesting murmurs. PDF is not a language, it's a file format! But then, isn't HTML a file format as well? Yes, if you prefer. There is in fact a significant overlap between file formats and DSLs. The two terms represent different points of view rather than distinct concepts. The term "file format" emphasizes the technical aspect of data management: how to store data in a file such that a program can read it. The term "language" emphasizes the human-computer interfacing aspect: representing information in such a way that is makes sense to both computers and their users. Purely machine-oriented file formats, and in particular binary formats, would not be considered DSLs, and therefore PDF is indeed a borderline case (it's a container file format inside which page contents are defined in PostScript, which *is* a DSL by any reasonable definition). But every file format meant to be read or written by humans, be they software developers or users, qualifies as a DSL.

In scientific computing, we find two dominant approaches to the human-computer interface, not considering graphical users interfaces because they are used for a very different type of interaction. The traditional one is to write application programs, usually in a compiled language such as Fortran or C++, that have a broad but fixed functionality. Such programs tend to require a large number of parameters and input files, which they read from a file typically referred to as an "input", "control" or "configuration file". Since these files are written by humans, their formats should be treated as DSLs. A more recent alternative approach is to create a collection of interoperating libraries, and let users write scripts that combine items from these libraries for doing a specific computation. The scientific Python ecosystem is probably the best-known

example for this approach. The problem-specific information is contained in a script, i.e. it is expressed in a general-purpose language rather than a DSL.

I will illustrate these two approaches by a toy example: the computation of the electrostatic potential of a set of point charges. A traditional program would require an input file looking like this:

```
0. 0. 0.  1.
0. 1. 0. -1.
1. 0. 0. -1.
1. 1. 0.  1.
```

The program's user manual would explain that each line describes one point charge by four floating-point numbers separated by spaces, the first three numbers being the Cartesian coordinates in nanometers, and the last number being the charge in elementary charge units. Without this explanation, the file would be impossible to interpret for a human reader. Even for a reader who knows that the file describes four point charges, it is not obvious if they are stored by line or by column, nor if the charge comes first or last, nor what the units are.

Next, let's look at a Python script using a hypothetical library called `ElectrostaticPotential`:

```
from ElectrostaticPotential import ChargeSystem
import numpy as np

charges = [(np.array([0., 0., 0.]),  1.),
           (np.array([0., 1., 0.]), -1.),
           (np.array([1., 0., 0.]), -1.),
           (np.array([1., 1., 0.]),  1.)]
energy = ChargeSystem(charges).potential_energy()
print("The energy is", energy)
```

In this case it would be the library's user manual that explains how to store the positions and charges in Python data structures, and in which units the values must be given. For the reader of the file, the Python code provides much useful context. Contrary to the input file containing nothing but 16 numbers, the script explicitly refers to charges and potentials. A reader with a basic knowledge of electrostatics and the Python language could easily identify the position and charge values from this context, but would remain in the dark concerning their units.

The sixteen-number program input file clearly suffers from a lack of explicit context (the type of information, including the units) and from a lack of structure (which values are the charges?) that would permit basic error checking. The Python script adds the structure and some of the context, but still lacks unit information. On the other hand, the Python script also contains too much information for the purpose of describing a point charge system: it refers to a specific library (`ElectrostaticPotential`) and to a specific computation (potential en-

ergy). Moreover, it also contains scientifically irrelevant technical details, such as the use of three different sequence data structures (lists, tuples, arrays). If I want to compute something else, say the total charge of my system, using a different library, I have to write another script from scratch, probably using somewhat different data structures. I end up having two copies of my point charge system definition that I may have to keep in sync for a while as my research project evolves.

So what would a proper DSL for point charge systems look like? A good starting point is to consider how the information would be presented in a scientific article. My choice would be a table:

position [nm]	charge [e]
0. 0. 0.	1.
0. 1. 0.	-1.
1. 0. 0.	-1.
1. 1. 0.	1.

This table contains exactly the information we want to store, neither more nor less. This could be translated straightforwardly into a machine-readable file:

```
| position [nm] | charge [e] |
|---------------|------------|
|   0. 0. 0.    |      1.    |
|   0. 1. 0.    |     -1.    |
|   1. 0. 0.    |     -1.    |
|   1. 1. 0.    |      1.    |
```

To make sure that this table syntax is a valid formal language, I would have to show that it can be parsed unambiguously by a computer program. The proof would consist of actually writing a parser, but I can spare myself that effort because others have already done it: I have borrowed the table syntax from an extension to the lightweight markup language Markdown, which is parsed by the popular file format converter pandoc (http://www.pandoc.org/), among others.

Compared to the original input file, this point-charge-systems DSL has the advantage of being immediately comprehensible to a human reader, because all information is explicit. Note that the presence of a unit indication does not imply that every program reading such a file must be able to handle arbitrary length and charge units. It might well report an error if the units are different from what it expects. In fact, the definition of the DSL (which I have not given) might well prescribe that the input table must have exactly two columns with exactly the headings given in the example, making the first line completely redundant. It's up to the DSL designer to choose the right compromise between generality and simplicity. The main difference between the DSL and file format

point of view is that the former can lead to requiring redundant information for the benefit of human readers, and require software to verify it as part of error checking.

The main obstacle to human-friendly DSLs is the effort required to implement them. Computational scientists are not specialists in parser development, so it's not something they are particularly eager to do. There are, however, various libraries and tools, called parser generators, that make the task easier. But even if you prefer to avoid writing parsers altogether (like myself, as I will happily admit), there are useful compromises that are much easier to implement while retaining many of the advantages of human-optimized DSLs. The general idea is to pick a syntax for generic data structures and express your DSL in terms of them. You can then use an off-the-shelf parser for the data structure syntax, and don't have to write your own. As a bonus, you may find that your favorite text editor already supports such data structure syntax.

Let me show you some concrete examples. A popular generic data structure syntax is YAML (http://www.yaml.org/), which calls itself a data serialization language. One way to encode our point charge system in YAML is

```
columns:
  - position: nm
  - charge: e
records:
  - [[0., 0., 0.],   1.]
  - [[0., 1., 0.],  -1.]
  - [[1., 0., 0.],  -1.]
  - [[1., 1., 0.],   1.]
```

Parsing this with the PyYAML library (https://github.com/yaml/pyyaml) returns a simple Python data structure that any Python programmer should know how to process:

```
{'columns': [{'position': 'nm'},
             {'charge': 'e'}],
 'records: [[[0.0, 0.0, 0.0], 1.0],
           [[0.0, 1.0, 0.0], -1.0],
           [[1.0, 0.0, 0.0], -1.0],
           [[1.0, 1.0, 0.0], 1.0]]}
```

Another generic data structure syntax is known as s-expressions. It is probably the oldest one, having been developed originally for the Lisp programming language in the 1950s. A possible encoding of our point charge system in terms of s-expressions looks like this:

```
(point-charges ((position nm) (charge e))
               ((0. 0. 0.)       1.)
               ((0. 1. 0.)      -1.)
               ((1. 0. 0.)      -1.)
```

```
                ((1. 1. 0.)       1.))
```

Again parsers exist for use with many programming languages. Python programmers can turn to the sexpdata library (http://sexpdata.readthedocs.io/) and obtain a Python data structure with next to no effort:

```
[Symbol('point-charges),
 [[Symbol('position'), Symbol('nm')],
  [Symbol('charge'), Symbol('e')]],
 [[0.0, 0.0, 0.0], 1.0],
 [[0.0, 1.0, 0.0], -1.0],
 [[1.0, 0.0, 0.0], -1.0],
 [[1.0, 1.0, 0.0], 1.0]]
```

One of the big ideas of Lisp is to use the flexible data structure notation of s-expressions not only for data, but also for code. This feels weird at first to anyone coming to Lisp from other languages, but to those who persist until they get used to it, the syntactical uniformity starts to show its advantages. One of them is the ease of adding small DSLs to one's programs using Lisp macros. If macros were invented today, they would perhaps be called DSL compilers, because that's exactly what they are. Note however that these are so-called *embedded* DSLs, because they are used inside a program, not in separate files. Python programmers can try this out for themselves using Hy (http://docs.hylang.org/), which is an alternative s-expression syntax for Python. Hy can also be considered a Pythonic dialect of Lisp, because it supports macros. Using Hy macros, you can develop DSLs that are sub- or supersets of Python, or both, with relatively little effort, as long as your DSL syntax fits into the overall s-expression syntax.

In case you end up addicted to DSL development, your ultimate drug is the Racket language (http://www.racket-lang.org/), which claims to be "the world's first ecosystem for developing and deploying new languages." Racket also has a Lisp heritage, and favors s-expressions for everything, but it also lets you plug in your own parser for your own syntax if you prefer. With such a language-development toolkit, designing and implementing DSLs requires no bigger effort than designing and implementing graphical user interfaces (GUIs). Note that I am not saying that either one is trivial, but both are within the reach of motivated software developers. DSLs are a better match than GUIs for many scientific computing tasks, so I hope scientific software developers will explore this option more intensively than they did in the past. Some examples from the Racket ecosystem worth looking at to appreciate the potential of DSLs are Scribble (https://docs.racket-lang.org/scribble/), a DSL for electronic documents, Slideshow (https://docs.racket-lang.org/slideshow/index.html?q=slideshow), a DSL for slide-based presentations, or Video ((http://lang.video/), a DSL for video editing.

The one guiding principle to keep in mind when designing DSLs is to focus on the human user's perspective, starting with your own one. Ask yourself how

you would ideally write down your scientific data or computation for use by a computer. Assume first that there are no software-related constraints, as if you were taking notes for sharing with a colleague. Then try to express the same information using generic data structures, for example in YAML or in s-expressions. Finally, if you believe that moving to a nicer syntax is worth the additional effort, write your own parser.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans and at the Synchrotron SOLEIL in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University. Contact him at konrad.hinsen@cnrs.fr.