MARK BURGESS

configuration management: models and myths

PART 2: BABEL, BABBLE, TOIL, AND GRAMMAR

Mark Burgess is professor of network and system administration at Oslo University College, Norway. He is the author of cfengine and many books and research papers on system administration.

Mark.Burgess@iu.hio.no

TIME TO PUT THE ADMINISTRATIVE house in order? Then you are going to need a way of describing that house. Configuration management, as discovered in part 1 of this series, is the management of resource patterns. If you can't communicate a configuration pattern, you certainly can't have someone create it, verify it, or maintain it. So, although there are clearly many ways to build your house of cards, you will need to learn the language of patterns if you want to make a bunch of them exactly alike.

Parentheses (and More Parentheses)

Call me not a linguist or a poet by training; my roots were nurtured in that country on Earth with surely the worst reputation for knowing foreign languages (worse even than the United States). Still, I am uncharacteristically both intrigued and betaken by language.

(What? "Betaken"? Not a word, you say? Hmmm . . . stay tuned!)

These days I live in Oslo, in the southern part of Norway, but I started my life in the northwest of England. Ironically, this is the part of England whose culture and language were "impregnated and borrowed" by Vikings from Norway in early A.D. The inheritance of that invasion is still there, to the observant eye and ear.

I lived not far from a charming creek called Beckinsdale (in Modern Norwegian, Bekk i dal, meaning "stream in valley"). People still call their children "bairns" in that part of the world (in Modern Norwegian, "barn" means "child"). There are many examples of such glossal cross-pollination. In fact, the languages of Old English and Old Norse were so alike that the Vikings and their victims probably understood each other quite easily, and today language scholars are often at pains to determine from which of them certain words came.

In dialect, I recall verb endings that sounded perfectly natural to me: "we've meeten, eaten, beaten, moven, proven." Surely, these are older forms of verb endings than in the modern English "we've met, beaten, moved, proved." (The endings sound somehow more Germanic, though I am just guessing; I was reminded of them on playing Deep Purple's song "Space Truckin'," where Ian Gillan sings, "We've meeten all the groovy people . . .")

It is odd that "eaten" alone has survived in the U.K. (as has, occasionally, "proven"; "gotten," however, which has survived in the U.S., is strictly forbidden in the U.K. and yet the derivatives "forgotten" and "begotten" are standard.). Clearly, the rumors of English grammar have been greatly exaggerated.

What of "betaken"? Why is this not a word? It clearly fits the grammatical forms. One even says idiomatically "I am rather taken by that" (preferably with a butler-like inflection) and, of course, there is a similar word "betrothed," which is in the dictionary. In Modern Norwegian it is indeed a word ("betatt") and it means exactly "rather taken by," so I hereby define the word "betaken." And who can stop me?

Indeed, language changes regularly and we are inventing new words all the time, using recognizable patterns. It tends to move from complicated constructions toward simple regular patterns. If you examine which verbs are still irregular (or strong) in language, it is those verbs that are most commonly used (e.g., "to be"). There is a simple reason for this: We only remember the irregularities if we are using them all the time and they are strong enough to resist change. In other cases we forget the "correct" forms and (regularize|regularise) them according to some simple syntactic pattern. Anyone who has seen the British TV show "Ali G" will know from his parodical dialect that there are parts of the U.K. where even the verb "to be" is losing to regularization: "I is, you is, he is . . . , innit." (Prizes will be awarded for guessing the last word's origins.)

In fact we add and change word endings willy-nilly: In the U.S. my least favorite word at the moment is "provisioning" (which I like to call "provisionizationing") although "de-plane" is way up there (and it surely means picking passenger aircraft out of the fur of a cat). These are particularly nasty examples of "verbing" and "nouning," especially American phenomenonizationings. In the U.K., people have an odd habit of saying "orientated" instead of "oriented," fearing possibly that the latter has something to do with a cultural revolution of cheap shoes, or harks of a country they never managed to "civilise." Or, perhaps they are simply so orientitillated that they feel they must.

At any rate, although there are definite patterns to be seen, clearly human language is driven by populism and natural selection, not by total logic or design.

The Chomsky Hierarchy

So much for human language. It seems to have very little to do with structure or reliability—qualities we are certainly looking for in system administration. So let's get formal.

In the passages in the previous section, I broke several rules of writing [although *;login:*'s copyeditor may have unwittingly "corrected" some of the more egregious abuses—*copy ed.*] and made you (the reader) work harder than is generally allowed in modern literature. I served a plethora of parenthetical remarks and digressions. I am guessing that you have noticed these (and that you had no trouble in parsing them) but that they were a little annoying, since you had to work slightly harder to understand what I have written. Of course, I was making a point.

The theory of discrete patterns, such as houses of cards or flowerbeds, is the theory of languages, as initiated by researchers including Noam Chomsky in the late 1950s and 1960s. For discrete patterns, with symbolic

content, it makes intuitive sense that discrete words and their patterns might be a good method of description; but when we get to continuous patterns, such as the curving of a landscape, what words describe the exact shapes and infinite variations of form? For that we need a different language: continuous (differential) mathematics, which we shall not have time to mention in this episode.

The theory of formal languages assumes that a discrete pattern is formed from an alphabet of symbols, shapes, colors, etc., much like a pack of cards; patterns are then classified by measuring the complexity of the simplest mechanism or computer program that could generate the pattern. The classes of patterns are called formal grammars. Their classifications and corresponding state-machines are as follows:

- Regular languages (finite automata, or finite state machines)
- Context-free languages (push-down automata)
- Context-sensitive languages (nondeterministic linear bounded automata)
- Recursively enumerable languages (Turing machine)

The syntax of a language is a list of all legal sentences in the language. Lists are not very helpful to us, though: We have trouble remembering things by brute force, so we try to identify the repeated patterns and turn them into rules. These pattern-rule templates are called grammars. The simplest grammars are the regular grammars, and the patterns they represent can be modeled by a simple pattern-matching language: regular expressions.

Regular Expressions

All UNIX users have met (or meeten) regular expressions. They are a well-known and powerful way of matching text strings. The implementations we know are stylized enhancements of the regular expressions of language theory.

A language is said to be regular if it can be constructed from some alphabet of symbols and satisfies a few basic rules. Let us suppose that we have an alphabet, A, which contains a finite number of symbols. Those symbols could be alphabetic, alphanumeric, numeric, glyphs, flowers (as in part 1), or any arbitrary collection of denumerable symbols. The rules are these:

- The empty string and each symbol in the alphabet are regular expressions.
- If E1 and E2 are regular expressions, then so is E1E2, i.e., the concatenation of the two (e.g., expressions “b,” “e,” “be,” “taken,” and “betaken”).
- If E1 and E2 are regular expressions, then so is the union of the two (i.e., we allow alternate expressions to be combined in no particular order). This is written with the vertical bar “|” in most implementations (e.g., we have (met|meeten)).
- If E is a regular expression then so is E* (repeated instances). Hence we have acceptable expressions “provision,” “ization,” and “ing” generating “provisionizationingizationingingingization,” etc. ad lib.
- Nothing else is a regular expression.

The Kleene star (*) is a shorthand for the concatenation zero or more instances of members of a set or expression. This is the parsimonious form of regular expressions. We’ll not delve into implementations for now.

Languages in Configurations

There has been a lot of talk about "configuration languages" as tools for sorting out UNIX systems: cfengine, LCFG, now Puppet, etc. Rumor has it, I wrote one of these myself. But don't let this talk of language trick you back into thinking about these tools. Rather, notice that the very problem of configuration itself involves language—because it is about describable patterns. For example, UNIX file permissions form the simplest kind of regular language. If we take the octal representation, they consist of scalar states of constant length and a fixed alphabet consisting of the following "symbols":

Q = {0,1,2,3,4,5,6,7}

It is easy to represent this as a language. It is simply the union of each of the symbols. That is, if we ignore the foibles of UNIX syntax, then the entire language is simply written

000|001|002|003|004|...|776|777

This is all very well, but so what?

The significance of regular expressions for configuration policy is that there is a provable equivalence between regular languages and finite state machines, i.e., the simplest kind of algorithms, using a fixed amount of memory. This means that regular strings are relatively easy to parse, identify, and understand. This, at least partly, accounts for their ubiquity in computer software where pattern matching is required.

Regular expressions occur in editors, where searching and replacing is required, in intrusion-detection and spam-detection software, in all kinds of policy languages, and on the UNIX command shell (as "globbing"). They are a central part of Perl, a language designed for pattern extraction (though Perl is not a regular language). Today, no computer toolbox is complete without a regular expression library.

Bring on the Toil (Parentheses Again)

In spite of the multifarious uses for regular expressions, they are only the lowest level of sophistication in the Chomsky hierarchy. The computer languages we are most familiar with for programming or markup are almost all context-free languages. Such languages can only be approximated with finite memory. They contain nested parenthetic structures that require an extensible stack to process. Here, for instance, are some examples of languages that use parentheses to identify information by type:

1. <account>
   <uname>User1</uname>
   <passwd>x7hsk.djt</passwd>
   <uid> 100 </uid> ... </account>
2. (account (uname User1) (passwd x7hsk.djt) ...)

If the level of parenthetic nesting in a grammar is not large, we can simulate common cases of context-free languages by treating fragments as regular expressions with balanced pairs of symbols (as anyone who has written a simple parser will know). This is useful because it means that a simple finite state machine can make a good attempt at interpreting the string and this is cheap.

However, to ensure full generality one must go beyond regular language tools and enter the realm of stack-based tools such as Yacc and Bison for

context-free grammars. Each level of the Chomsky hierarchy grows in its computational complexity (costing us more to parse parenthetic remarks (as you (no doubt) experienced in my introduction)). The most general patterns require a full Turing machine (a computer with infinite memory) to solve.

The trouble with this next level of computation is that it is a drastic step. It requires a whole new level of sophistication and toil in modeling, describing, and understanding to master. We want to use higher-grammatical patterns to design, classify, and maintain structures that are context free. Worse yet, the structures might be inside files, in packet streams, distributed around a network, or inside a database. The difficulty of going beyond finite state automata partly explains why pattern-recognition systems (such as network intrusion detection systems), which obviously need to deal with parentheses (e.g., TCP-SYN, TCP_FIN), generally do not record such state, but rather rely on regular expression rules applied as fragments. This is "doable," if not optimal.

Data Types and Bases

In configuration management we meet information in a variety of forms. Lists of values are common. Line-based configuration files are ubiquitous in UNIX. Windows has a simple key database in its registry. What kinds of languages do these data form?

- Scalar permissions are regular languages.
- Lists of regular objects are also regular.
- A line-based text file is a list and hence is regular.
- Text files containing higher grammars such as XML are context free.

Relational databases have been used to store data almost since computing began. They add a new twist to the idea of languages, namely that the words one forms from the basic alphabet of a language (and sometimes even the symbols of the alphabet) can be classified into types. Consider Figure 1.

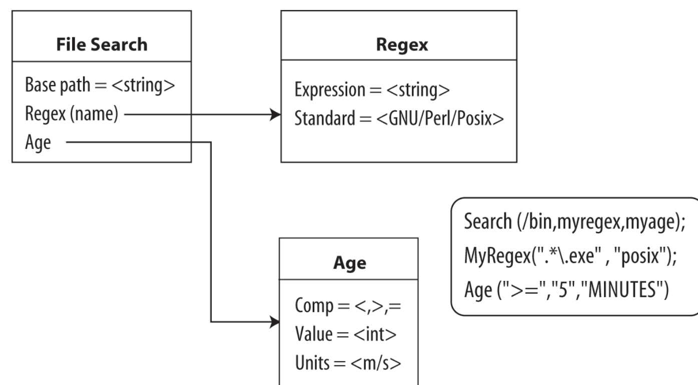

FIGURE 1: SOME TABLES IN A RELATIONAL DATABASE

The figure shows the basic idea of a relational database. Certain types of data are grouped together in tables or records. Such data structures have eventually ended up in programming languages too, in the form of records, structs, and now even object-oriented "classes." The main point of putting information into a predictable structure is that one imposes a linguistic discipline on the data. The tables are simple parentheses around a number of regular language items that are given names. In the first table we have a string (which is a regular object) with a name "base path," a "regex,"

which is a new kind of table or parenthetic grouping, and an age, which is yet another parenthetic grouping. The “regex” has two regular members: a regular expression (which is a string and is hence also a regular object) and a label (string or number), which is regular. Similarly, “Age” consists of a list of three regular objects.

A relational database is therefore a context-free language. SQL is a query language that uses regular expressions embedded in a table model to locate data in the database (which has its own context-free language pattern). We cannot escape from languages or these basic pattern ideas in configuration management. They recur at all levels.

Data types are a powerful idea. They allow us to distinguish among seemingly equivalent patterns of data and therefore open up a range of flavors or colors to the flowers in our garden. This is the purpose of having tables in relational databases: We can group together objects into comparable clusters. Syntactically, all objects of the same type have the same basic structure and are therefore comparable, i.e., they form the same subpattern.

Markup

The trouble with databases is that they are not very transparent—they can only be read with special tools, so it is hard to see the structures in data in an intuitive way. This is less of a problem in computer programming languages where class hierarchies are written down in ASCII form. For many, the answer to this problem has been to adopt XML, a generic markup representation for a context-free data structure, which adopts the best of both worlds. Not only does XML offer a standardized encoding of a context-free structure, it claims to make it parsable by humans as well as machines. (Let us say that the rumors of its human-readability have been greatly exaggerated.)

Every pair of tags in a markup language such as HTML or XML makes a data type out of the parenthesized region. For example:

```
The <adj>quick</adj> brown <noun>fox</noun> <verb>jumps</verb>
over the lazy dog.
```

The current adoration of XML has no real significance as far as problem-solving goes, but it is interesting that the trend in system design is to move away from regular line-based data, as is traditional in UNIX and DOS, toward context-free data. This opens the door to much greater complexity, with attendant consequences that we shall consider as the series progresses.

Revolution or Regex?

Toil, work, and difficulty relate to grammars or patterns rather than to symbols. Noah Webster, as a slap in the face to the British, rewrote the spelling of the American English as a political act after the revolution. (No doubt my own spellings “colour,” “flavour,” etc., have been magically transformed into American “color” and “flavor” by the copy editor. [Indeed—*copy ed.*]) The adaptation has almost no consequence (except to annoy self-righteous Brits immensely); many readers hardly even notice this change. Had Webster altered the grammar of the language, there would have been serious trouble. But the fact is that, although he obscured some of its etymology, the basic patterns of the language did not change, and therefore even the most obtuse of colonialists can still read American (although Canadians seem totally confused about how they are supposed to spell).

The patterns that we are able to discuss and represent are key to mastering the problem of configuration management. Many system administration and management tools try to force users into doing either what the tools can do or what is considered manageable. By asking users to limit the complexity of their configurations they plump for a reasonable strategy that strives for predictability. This might be all right in practice, for the time being, but if we are going to fully understand the problem, we must go beyond quick fixes. The challenge for any theory of configuration lies in describing what people really do, not in trying to force people to do something that is easy to understand.

In the next part of this series, I would like to run through some of the data models that have been applied to the problem of system management. We shall ask the following: How can we measure their complexity, and why are none of them ever really used?

Save the Date!

www.usenix.org/fast07

5th USENIX Conference on File and Storage Technologies

February 13–16, 2007 San Jose, CA

Join us in San Jose, CA, February 13–16, 2007, for the latest in file and storage technologies. The 5th USENIX Conference on File and Storage Technologies (FAST '07) brings together storage system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. Meet with premier storage system researchers and practitioners for 2.5 days of ground-breaking file and storage information!

Sponsored by USENIX in cooperation with ACM SIGOPS,
IEEE Mass Storage Systems Technical Committee (MSSTC), and IEEE TCOS

USENIX