Rosie Pattern Language for faster data mining

IBM Code Tech Talk

Nov 8th 2017

https://developer.ibm.com/code/videos/tech-talk-replay-rosie-pattern-language-faster-data-mining/

>> Go ahead, Jamie.

>> Thanks. Welcome to the second tech talk on the Rosie Pattern Language. The first one was last year, when Rosie was still new, and we introduced a number of ideas that have been further developed in the year since. What I'm going to talk about today is a little bit of a review of some of the concepts I've presented last time, which is about where regular expressions can fall short, in high scale applications like data mining and big data in particular, and what the Rosie Pattern Language is, how it extends the concept of regular expressions. I'm going to conclude with use cases and encourage you to get involved in the Rosie project.

My name is Jamie Jennings, I work in the Cloud division of IBM in a group that builds DevOps tools. We designed Rosie Pattern Language when we were first starting off doing

analytics on data from data that's emitted from DevOps tools, like source code repositories and issue tracking systems and build systems and test systems. There is all kinds of data thrown off by those applications, and we were collecting it and needed to parse things out of it.

What became apparent immediately is that as you get to having a lot of regular expression patterns that match pieces of data, and as you get to having a lot of people, particularly people working in different languages, and as the data size increases, we run into limitations of regular expression.

The technology that my teamworks on is called DevOps insight and it's built on the idea of a open Toolchain where we integrate the tools that you are already using to build your project. You can see at the bottom here, all the, this is a some of the tools we integrate with. There are more.

There are source code repositories as I mentioned, and issue tracking systems and build systems and other kinds of automation. This was the motivation for Rosie was the realization that all of these tools emit data in different formats, and our team had to write regular expressions for pulling out different pieces of data from each of these tools and finding just the pieces of information that we needed in that data.

If we zoom out from the project that I work on to

planet-wide, every day we create over 2.5 exabytes of data, and that number comes from an IBM study which is now a few years old. The estimates are that less than half a percent of data is actually ever analyzed. So what that means is that there is a missed opportunity here, and so we want to know why is there a missed opportunity? Well, because all that data may have useful information in it, but it's a long slog from when you first get the data to when you can analyze it.

A typical flow would be, maybe you are looking at runtime logs and you are trying to do some analysis that relates failures that show up in a log with source code changes that went into the build, that went into the deployment. Well, as soon as I get that log file, I need to write some code or regular expressions and code to parse out the pieces of log that I'm interested in. Then I need to test and correct my parser, I need to add some semantic tags. I have to do a lot of things before I actually get to analyzing the data.

This is behind the classic estimate that 80 percent of the effort in analytics is actually getting the data into a form where you can analyze it. And then if you are lucky, you spend 20 percent of your time actually analyzing.

So the red exclamation point is near the top where there is a lot of frustration and I think a lot of it is due to limitations in regular expression, and that is the spot that

Rosie is trying to address.

We are going to do just a quick review of the, of how regular expressions behave in the wild. So not a review of what they are, but just a review of what it's like to use them. Those of you who have been around a long time will remember that regular expression as a tool for programmers, for developers, for administrators, came about in UNIX. And this time frame was the 1970s. Regular expressions had already been codified as a theory, if you remember automata theory from college, I know some people don't want to remember automata theory from college, but there is a hierarchy of languages, regular languages, context free, context sensitive, etcetera.

The folks that wrote UNIX realized that regular expressions, if you gave them a form that you could easily type, would be good for searching through data. What is amazing is that nearly 50 years later, regular expressions have not only survived, but there are dozens of variants of them, and this screen shot on the right, this little Web clip is the top maybe 25 that are regular expressions flavored.

There is many different kinds and many different tools then for helping you adapt to one kind or another, which is great. This is something we are going to come back to later, but Rosie is actually a super set of regular expressions. It is not something we have time to get into today, but Rosie recognizes

some of the context-free languages, not all of them, and can do everything regular expressions can do as well.

Just a reminder, this is the kind of code that we are dealing with. Someone wants to pull a date out of a file that contains a whole bunch of stuff, and they write an expression like this, in this case it starts with a caret and ends with a dollar sign meaning that the date is on a line all by itself. But we know how to change that if we want to match it in the middle of a line instead.

And this kind of expression looks straightforward when it's short. When the expressions get longer, they get obviously unreadable. This expression here matches an E-mail address. As soon as you get to this level of complexity, regular expressions become really hard to maintain, because somebody who has to fix a bug in this needs to understand it in order to find the bug and repair it.

A preview of where we are going with Rosie is that we are going to give names to patterns and refer to these commonly used patterns with names, and let our users build up libraries of named patterns and combine patterns by name.

In fact, if you are playing the home game and thinking ahead to what it would be like to have expressions with names, you would say, well, unless that E-mail is easy to type, but truly that has to be defined using something that looks like the

expression on the left, and that is not really what we are going for here.

What we are going for is E-mail is defined as if you look at the last line in this new box at the bottom of the screen, E-mail is defined as name at host, that is pretties, easy to understand, name is defined as quoted name or unquoted name, etcetera. That is a preview of where we are going.

This concludes reg ex-issue number one, hard to read and hard to maintain. Regular expressions have a cryptic syntax. The fact that it's dense is actually good if you are typing stuff in the command line, you can type very few characters and do a lot. But it is terrible for maintenance. Regular expressions also have flags and modes that change the meaning, like case sensitivity modes and whether or not dot matches new line and things like that. They don't concatenate, they don't compose very well.

You might say, but I don't have to write very many regular expressions because I can just go on-line and get mine from stack overflow or find a gist like this one on the screen -- and yes, it's easy to find them on line but I wonder do we want to, do we want to take something like this that we found on line and cut and paste it into our application? Part of the reason is that again, it is not going to be easy to maintain and once I borrow this, I own it.

I don't want to pick on the person that posted this particular gist, but I looked at this myself and I said, IPv4, IPv6, UR I, those don't actually look too bad. Then I realized that you would have to scroll over to the right. I made a movie of my scrolling to the right. You have to scroll to the right to see the whole expression.

Again, not to pick on the person that created these expressions and posted them, but they are regular expressions, and when they get long, they get really hard to understand. This last one just goes on and on and on.

This is my favorite part actually, if you scroll down, I have found a little bug.

(chuckles).

This kind of thing happens all the time. You can be an expert programmer and the person who posted this probably is, you can be an expert programmer and you can still have a bug, because these things are so hard to develop. Okay.

On to point number 2 about regular expressions. The performance is surprisingly variable. Here is the good, the bad and you know what is coming next, it's the good, the bad and the ugly. Regular expression matching can be done efficiently in linear time. The bad news is almost every implementation uses an exponential time backtracking strategy, which means if you give it poorly formatted input, it will take

a long time to execute. I really recommend the blog post that Cox made about this issue and other issues with regular expression, and he motivates why he wrote RE2 which if you have to use reg ex-I would recommend that as your library.

Here is the ugly part. Two lines of Perl, right, I don't have the actual line that says match this input against this regular expression, but that would be the third line.

Matching a 29-character string can cause Perl, Perl's regular expression processing to do so much backtracking, it takes 36 seconds to do a match. So something that should take microseconds is taking 36 seconds, and the second example, in the second example we have comma separated values. You can see the last four fields look like levels of awards, bronze, bronze, gold, silver, and there is a regular expression there that says, I want to know if the last one is gold. Skip over the first 29 comma separated values and tell me if the last one is gold. Do that match in Perl, 65 seconds. So this is bad.

If you have a big data pipeline, you know you will eventually get badly formatted data, and your streaming processing is going to stall when you get something that is badly formatted. Even if you don't get badly formatted data very often just by accident, someone could maliciously inject badly formatted data in a denial of service attack.

Moving on to one last point about why we wanted to move

beyond regular expressions. I mentioned earlier that having a collection of regular expressions like a library of network expressions and date and time expressions would be a good thing. There are tools other than Rosie that let you collect a bunch of expressions together, and give them all names. And Grok does this. Grok is the part, the plug-in for elastic searches log stash that identifies all of different pieces of your log entries and then tags them so that when you go to elastic search you can search and aggregate by date and IP address and process name and all other kinds of things.

So this little clip here says Grok sits on top of Regex and the Regex library is oniguruma. I'm sure I pronounce it wrong but my apologies to who named it. This type of regular expression is not even on the 25 variants list that I had at the beginning of my slides. Here is a 26th common type of regular expression which means it has slightly different rules than some of the other expressions you might use. Here is another clipping from the log stash website, log stash ships with about 120 patterns. Like Rosie here is a technology that gives you a library and you can add more patterns on your own which you can do with Rosie as well.

Here are the caveats. We use elastic and logstash, these are great tools. But Grok itself as used in logstash has limitations. There are often name collisions, where somebody

defines a pattern called user, and somebody else defines a pattern called user, and they are in different files in the same directory, some versions of logstash will use the first defined version of that name and some will use the second.

There aren't any packages or hierarchy that would distinguish one from the other. There aren't any unit tests to help you understand what these patterns actually match, and of course, still at the bottom, you have regular expressions which are rather unmaintainable and rather unreadable.

Here is another reason why it matters that they are unreadable and unmaintainable. Here is, I've got two Grok trials, original and copy. They have entries in them like quoted string is a name that is defined as this expression on the right. IPv6 is a name that is defined to be that expression on the right.

All of our source code management tools these days and for a long time actually in the past are based on diff. Diff-based tools are line oriented, so if I diff these two files, I find out that quoted string is defined slightly differently in one than in the other. So is IPv6.

Yet, they look almost identical. I can promise you there are changes, there are differences between them because I put them there. But of course if this happened in real life, if it showed you this diff, how would you know what was really

changed? You would have to go character by character, and then you would have to understand if this character changed, what did it actually mean? Does it change what the expression does.

Of course, it's the 17 minute mark of the talk which means we must quote Bob Dylan. There is too much confusion, I can't get no relief and this is why we invented Rosie Pattern Language.

I'm going to pause for a second, to see if there are questions coming in on the chat that I should address right now.

>> Yeah, we are getting questions about if there is any universally accepted expression pattern, is there any standardization of these things? I think you touched a little bit on that. But maybe, what is the industry as a whole trying to do about these problems?

>> JAMIE JENNINGS: That is a great question. To my knowledge, everybody doing big data has their own workaround, their own framework for making sure that their regular expression collections don't get out of control, and that they don't do anything too bad at runtime.

So the Apache framework for natural language processing, whose name escapes me at the moment, has its own, and tools like Grok have their own with limitations. I'm not aware of standardization efforts for these kinds of regular expressions.

An odd benefit perhaps of Rosie being singular is that there is only one RPL set of expressions, there is only one library. But it can be a community library.

So I've got a set of standard expressions that ship with Rosie, and if anyone finds bugs in it, we will fix them. But you can use your own libraries. You can copy ours and make changes, make enhancements, fix bugs on your own. You are not locked into what we do.

>> I think that the second question kind of on the same area, was if people would use predefined modules for the Regex which I think is what you addressed as well.

>> JAMIE JENNINGS: Um-hmm. Okay.

>> Apache is open in LP.

>> JAMIE JENNINGS: Open in LP. It's very popular around here, because that work was done by some of the IBM researchers, and was part of the original Watson that played Jeopardy. I should have known the name.

The solution in my eyes is Rosie Pattern Language which here is the main part on shift from regular expression. It's designed to look like a programming language. It's not a programming language in the sense that it's not complete, there aren't loops and while statements and things like that. You are defining patterns.

But you are defining them using syntax and semantics that

are very familiar to programmers. There are comments, there are modules, there are identifiers, so patterns have names, like in the example on the screen, this parses all of JSON and the definition in this JSON RPL file looks very much like the JSON.org example on the cover page. That is no accident.

There are names like value and number and object in array which are identifiers. You can put any white space in, any place you want. One consequence of that is that literals have to be quoted. If I want to match the word true, I have to put it in quotes. But you know what, we do that in programming all the time anyway. That is actually more conventional than the way regular expressions do it.

Then unit tests, so whenever you are developing these expressions, it's really tempting to go to one of these websites that lets you put in your Regex and put in some data and it will tell you if it matches. Those tools can be useful, and there are hundreds of such websites.

But you have to make sure you configure them correctly. You have to make sure you choose the dialect of regular expression that you are actually using, and that you set switches correctly and all that.

But different approach, which is the one we took in Rosie, is to put unit tests right in with your patterns. Unit tester runs automatically when the RPL code is compiled, and it tells

you whether they pass or not.

We build on top of concepts that are already familiar from regular expressions. Most of the concepts are from regular expressions, are directly translatable into Rosie Patterns. You will see familiar things like pattern star is one or more copies, pattern plus, sorry, pattern star 0 more copies, pattern plus 1 or more, etcetera, I seem to have left out pattern N M for at least N copies but atmost M copies. That should be there as well. Character sets should look familiar, X-Y, everything from X to Y. Near the bottom, look ahead and look behind, those concepts are directly from regular expressions. I changed the syntax to make them more simple, because we are freed from the sort of 1970s legacy of regular expressions, we can have a little bit more sane syntax.

So, greater than is looking at ahead, less than is looking at behind, and P/Q is the Rosie version of P or Q. There are some differences. Rosie assumes that your pattern is anchored at the start as if there were a caret at the beginning of it, although you can certainly search for something anywhere in the input and we will see that later, matches are always possessive and always greedy. Those are choices you have when you use regular expressions as to what is greedy and what is not and what is possessive and what is not. Those choices get people into trouble all the time, because they often, you can make the

wrong choice for your data set and yet it still works on all your sample data. It just doesn't work fully in practice.

Sometimes too much choice can be a limitation. Rosie locks down some of these things and forces you to be a little more explicit about what you are trying to do. We will see some examples.

The benefit though, if you write Rosie expressions instead, you get to use the concepts you know from regular expressions, but you get a bunch of benefits in return, so you have to be a little more verbose in what you write, not as compact as Regex, but you get structured output and I'm going to show you that in the form of JSON, whereas regular expressions and tools like Grok give you a flat list of matches. There is a little more sane syntax. You can recognize recursive structures like JSON, and there is a number of other things. We will get to most of these briefly in the talk but not all of them.

The standard library that ships with Rosie that I mentioned earlier contains a number of patterns that are defined as collections. So date.any is the collection of, I don't know, a dozen or a dozen and a half different date formats. If you are not sure which one you need, you can use date.any which is great as a command line.

There is a package of commonly used integer, float, things like that. There is network patterns. There is a lot of time

stamps in different formats, CSV data, JSON data. I grayed out the log files and source code items from this list. They used to be part of the standard library in the pre1.0 days for Rosie. I think what we are going to do is factor them out into a separately loadable sort of library in the Rosie community on GitHub. These I expect to have a lot of changes, a lot of contributions, and so having them outside the Rosie release helps with semantic versioning and things like that.

I want to go through some examples, so that you get an idea of what this looks like at least from the command line. After the examples, I'm going to stop and we will look at questions again.

The next few slides are in lieu of doing a live demo. I really prefer to do a live demo where you can see the stuff happening live in realtime. But then I can't annotate it as easily. Instead, I'm going to reveal a transcript of a shell session bit by bit.

Here is a hand generated file that looks like resolves.con but has bogus data in it that I generated just for testing.

Here is a Rosie command. Rosie has a number of commands, one of the commands is grep and grep essentially does what grep does. Except instead of a regular expression you give it a Rosie expression. In this case, I did grep net.any and net.any you can see on the right in package net is defined as either a

IP address or Mac address, fully qualified domain name, an E-mail address, a URL path and what do I get out? I get out just the lines of that file that has one of those IP addresses or domain names.

Rosie has a number of different ways of generating output and one of them is color which makes it a lot easier to see what is being matched. The color switch says, just show me the matches. So it's different than grep which shows you the full line. But just show me the matches and highlight them according to what actually matched.

This particular set of matches isn't too interesting except that the IPv6 address is underlined so even though all the network addresses are red, the underlined ones is distinguished as being IPv6 as if you couldn't tell anyway.

Here is a preview of something that is I think useful in Rosie 1.0, that we are not going to have a lot of time to talk about (overlapping speakers).

>> Quick question. (overlapping speakers) underline the IPv6 address.

>> JAMIE JENNINGS: I'm going to show that on the next slide.

>> Great, thanks.

>> JAMIE JENNINGS: Your timing is excellent.

>> Thanks.

>> JAMIE JENNINGS: The way the grep command works in Rosie

is that it takes the Rosie match command which is the ordinary like match this pattern against this data, and it prefixes your pattern with findall which is a macrothat says, change this pattern from something that matches at start of the line to something that matches anywhere in the line.

We will come back to this and touch on it briefly, but there is not a lot of time to talk about macros today.

Here is the same command we ran on the last slide, but I ran it through the programs head and tails just to select out the second line which has three domain names on it. Here is what that same command looks like if I change the output format to JSON. Internally, Rosie is producing a JSON structure or a data structure that looks like this JSON structure. So, the output at the very top of the slide was IBM.com and then something else and something else.

If you look at the three highlighted areas, you see IBM.com and then the other thing and then the other thing. What you are getting out actually is a parse tree, and what is significant about that is that it has structure. Each of the net.fully qualified domain name each of those net dot SQVN items is inside something called net.any. Rosie is telling you, yes, I found three instances of net.any, in this case all three of them happen to be net.fully qualified domain name and the value is IBM.com and my local domain and example.

To answer Marc-Arthur's question, the reason that these three things showed up in red at the top of the screen, in the color output, is because there is a piece of configuration for Rosie where you can declare which kinds of things you want to see and which colors.

Net.star is defined as red so that is the default for any network pattern. Net.IPv6 is defined as red underline, and then in the next slide or second one after, you will see things show up, showing up in green, like the path.

>> Also the Mac address, right? (overlapping speakers) underline.

>> JAMIE JENNINGS: Also the Mac address, absolutely. I'm not sure I have an example of that one. But it is there.

>> Cool.

>> JAMIE JENNINGS: Here is the last example. I think it's the last example. Here is a pattern that I type on the command line that is more than one item. I'm going to compose a pattern which is word.any followed by net.any. I'm telling Rosie, match all lines in my file resolve.conf, all lines that have the format word.any which is any word upper case, lowercase doesn't matter, followed by a network address, and words print out in yellow. That is just the default configuration.

So I get what you see here. Let's do Rosie grep for

date.any, which is a contrived example because I want to show you that we are doing syntactic matching. We are not looking at any semantics. We are just looking at patterns of characters.

If I grep for date.any I expect every line in the output to contain a date. But those last two lines with nameserver starting at the beginning of them, those don't contain dates. What is happening?

This is the reason that the color output is useful, because by using the color output I can see immediately which part of the line matched and I can tell by the color what did it match, and in this case all dates and times are coming up as blue, but by seeing 2920 I can see that Rosie found something that looks syntactically like a date, 2.9.20, inside the IP address.

This is just a reminder that we are matching based on syntax. If what you wanted was a date as a token all by itself, with a word boundary at the beginning and word boundary at the end, we know we can do that regular expressions with backslash B and you can do it in Rosie too. I tried to avoid identifiers in Rosie that start with a backslash because that was a necessary escape mechanism for Regex that we don't need anymore.

So I used tilde for a word boundary. If you grep for a word boundary and then a date and then a word boundary, you can get

out the only thing in this file that is a date.

It's possible to have Rosie add the word boundary automatically. If I grep for the word update followed by date.any, I don't need to specify that there is allowed to be white space in between. That is another feature that we can't get into for time constraints today.

Finally, earlier in the talk I alluded to a pattern called all.things. And a package at least called all, and in that package is a pattern called all.things which is defined as either a word or a date or a network address or a path or a Mac address, and we will see one more example of that later under use cases.

This is a great way to see, if I have a file full of stuff, and I say Rosie, match all.things, it's a great way to see how much of the file is recognizable already, by patterns that Rosie already has defined.

At this point we are very happy. These penguins would be dancing if I could have found a diff but you get a set of happy penguins sitting there instead. I'm going to pause for a second to take more questions. Then we are going to go into use cases.

>> We got lots of great questions on chat, quite a few. I'm going to start with low-hanging fruit. First, when you use the dash O, you actually show the whole line and then you cover the

area of the line so that they can see it inside of response first.

>> JAMIE JENNINGS: That is a great idea. Dan, file an issue. I want to make that a enhancement.

>> Another one is asking about your performance compared to, he wants to know are you faster or slower, how does that work out?

>> JAMIE JENNINGS: Excellent. I'm going to put up that chart, and then talk to it in a minute after we get more questions.

>> Dan quickly, how does he file an issue against Rosie?

>> JAMIE JENNINGS: Okay, well, there are many ways to find Rosie on GitHub. You can go to GitHub and type Rosie Pattern Language or Rosie Pattern, that should get you there. You can go to trainee.cc/Rosie which will take you to Rosie's blog, and the Rosie blog has a link to Rosie on GitHub.

>> What is RJson PP is that (overlapping speakers).

>> JAMIE JENNINGS: Good question, I keep forgetting to explain that. In order to do pretty printing of JSON, so that it's readable by humans, you need a little utility to do that. Most people use JSON underscore PP, which is a commonly available one. I wrote my own, maybe I'll bundle it with Rosie just for other people's convenience. But yeah, that is one that I did myself.

>> We have a question about interaction and large amounts of text, so I get a large file, can you handle that.

>> JAMIE JENNINGS: Yes, the limit on how much text you can search through in a single go is four gigabytes. That is an addressing limit, so that the address fits into a relatively small topic.

>> We have a lot of questions which I know you cover on slide 22, about what does it work with, what type of languages, etcetera, if they use it with Perl typescripts or JavaScript, Python, etcetera. (overlapping speakers) slide 22 for that.

We have a question about how do you know how many matches you have? Is there a count basically?

>> JAMIE JENNINGS: Interesting question. No, what I usually do is run the output through word count, if you are generating JSON, the JSON will be one JSON object per line. I pipe into word count or if I'm writing in Python, you can count the number of things in the array that you get.

>> We had a question about typing, so they want to know how well does it handle piping in like Linux?

>> JAMIE JENNINGS: Pretty well. I developed on Mac OS these days but I'm a long time Linux user from the first Red Hat release in '94. So I think I'm very happy to support that use case, and will make enhancements to make it better. But it will read from standard in, it will read any number of files.

You can use it with XR, you can type the output. There is output standard out and output to standard error, if you want to enable that the command line switch, so you can get things that don't match, standard error, and things that match go to standard out. You can pipe them differently with a T or redirect them to different paths.

>> We have a question about how you can change formats of dates, dates vary from country to country, how you can change that.

>> JAMIE JENNINGS: That is a great question. In case you didn't hear, date formats vary from place to place in the world, and how do we deal with that?

Today, the file date.rpl which defines all the dates has common date formats from a lot of places in the world that use the Arabic numerals 0 through 9. And then has a separate set of patterns that have English names for the month and the abbreviations for the month, and the days of the week.

What I would like to have are contributions of similar patterns that use the strings for month names and day names in other languages, and then a really interesting topic of enhancement for Rosie would be to automatically load in the language that is your current locale, or let you select one on the command line. But I've done some thinking about this, but I have not had time to pursue it, and would love to have a

conversation about that.

>> Like I said earlier, lots of questions about what it works with, so when you get to that page, people wanting to know about SQL, wanting to know about all the main large programming languages, and we are getting a lot of questions about, is it agnostic language or frameworks or is it created to match each of those, so how extendible I guess that would be. We have one or two more here that we can get to before you -- most of the questions coming in are about that. Someone wants to know what you wrote Rosie in.

>> JAMIE JENNINGS: It's a mystery (chuckles) I'll tell you about that. There is a small but wonderful language called Lua which was developed at a university in Brazil. It is used widely in the gaming industry, video games and on-line games, because it is easily embeddable and sandboxable. It has great performance.

>> One last one, remaining questions will be answered in the next part, which is they mentioned that you had previously posted about auto magically converted text to RPL and wanted to know [inaudible]

>> JAMIE JENNINGS: If I understand the question about automatically converting text to patterns, it's can you read in text and infer what a pattern would be, create a pattern to recognize that text. If that is the question, then yes, and I

am working with a group of students at NC state university right now doing exactly that.

>> I think you are good to go to the next one. I'll try to see if we address everything after you get through that slide.

>> JAMIE JENNINGS: Awesome. We are going to touch briefly on performance and then platforms and languages. Then I'm going to cover use cases and then we are done.

Performance, Rosie does pretty well, functionally Rosie does more than most regular expression matching engines. Right? You have patterns within patterns, and you can generate nested output. You have a number of different constructs in Rosie, like the one that lets you recognize recursively defined things like JSON which you cannot do with regular expressions.

Rosie being strictly more powerful letting you do more, you would expect that it would not run as fast as most regular expression engines. It's actually pretty competitive. I cherry picked Grok because Grok is the closest thing out there to what I wanted before I wrote Rosie, which is it has a collection of named patterns, and lets you define new patterns using the names of old ones.

That is the most fair comparison I could find for Rosie performance-wise. If you look at this chart, you can see JGrok is Grok running in the JVM and it leverages Java's regular expression matching which is pretty fast. It is still much

slower than Rosie by several factors. But once the JVM starts up what you can see is the first eight or nine seconds in the orange line on the left, once JVM starts up, JGrok is doing pretty well especially compared to plain Grok which is in Ruby.

My test bed for routine tests I use files that contain roughly a couple million entries. This was log entries, so two million log entries, with both Grok and JGrok fell over. They dumped core, they faulted because of the UCF encoding error in the data. This is unfortunate because the data came from a real running system in the IBM Cloud, and so lesson to people who are writing data miners, the data might not be formatted exactly perfectly.

If you are assuming UTF8 and you get something that is not valid UTF8 you better handle it. Rosie does and handles it gracefully. The performance lines for Rosie which are all really good, you can see with JSON output it's twice as fast as it used to be and about, I don't know, five times faster than Grok, six times.

>> We had a question come in, in this area. System T, they want to know how it works compared to system T which is the engine test analytics that is part of IBM's big insights product.

>> JAMIE JENNINGS: That is a good question. I don't know how the relative performance goes. But I would be interested

to find out. I need to come up with somebody who uses that, to help me set up a fair test.

>> The question came from Ronnie Barker.

>> JAMIE JENNINGS: Excellent. I know Ronnie.

>> Maybe you can talk to Ronnie.

>> JAMIE JENNINGS: Absolutely.

Usage and platforms, and then we will cover some use cases. There is a, if you want to use Rosie stand alone which is how I showed the examples so far there is a CLI and there is also a read eval minute loop. The Repel which I don't have in this talk, the REPL is useful because you can type in, you can compose patterns on the fly. You can define new names, you can build patterns out of names you have already defined. You can load in libraries, and you can match patterns against sample data.

That is a really powerful way to develop, because it is iterative. When a match fails, you can, Rosie will print a trace of, it's like a exception trace. It shows all the steps in the matching and where it went wrong. That is really useful.

If you want to use Rosie as a library from Python or C or go or Ruby or whatever, Rosie is a shared object, it's a library. You can dynamically load this in a Mac OS or on Linux. You can statistically link if you want to. It's only about 500K, for

all the IBMers listening, that's K, not megabytes. Kilobytes. We are used to thinking in terms of how many megabytes does that take or how many tens of megabytes. No, it's half a megabyte. I grayed out Ruby, Node.js and Go because for Rosie version .99 I wrote those interfaces and need to update them for 1.0. They will be out soon.

Platforms, I test them about six different Linuxes, and OSS, I test in Docker, and it sort of works on Windows. It will work but not really on Windows, it works on the Bluetooth subsystem on Windows.

>> One language [inaudible] everyone wants to know if it works with what they are doing exactly, .net or C sharp?

>> JAMIE JENNINGS: I don't have a .net or C sharp client. But it should be possible.

>> SQL [inaudible]

>> JAMIE JENNINGS: That is great. That is another enhancement area that I want to explore with people that use those. The idea of using Rosie as the matching engine to speed up or to make more reliable some of these other query languages I think is a very compelling idea.

It's a wonderful idea. I would very much like to continue that conversation.

>> JavaScript, Python, PHP, those core --

>> JAMIE JENNINGS: I keep forgetting how popular PHP is and

I will add it to my list.

>> Somebody is saying they are having difficulties with your git page link program. Maybe that is something (overlapping speakers).

>> JAMIE JENNINGS: I will look into that.

>> Questions about languages [inaudible] COBOL, USF, Java, things like, people like what you are doing and want to know if it works with (overlapping speakers).

>> JAMIE JENNINGS: Awesome. I have not tried it in the Sigmund environment on Windows. It may compile there. It may not. I would certainly be willing to help someone who needs to work in that environment or wants to work in that environment.

PL1, I don't use, but I would be happy to help somebody write a client. Lib ffi is the interface I've been using to write the clients in all of these languages. If there is a lib ffi for your language then this is prettiesy -- pretty, easy, straightforward.

>> Is there a way if you have a language that you want support for, that they could contact you and maybe afterwards get some tips on how to either use what you are doing in the language or what can be done to help them?

>> JAMIE JENNINGS: Absolutely. The best way to bring a request to our attention is to go to the GitHub page for Rosie, and create an issue. You can label it as an enhancement

request, if you want. You don't have to add any labels. I can add those later. But type in what your suggestion is, what language you want supported, and I will be in touch.

>> Yeah, I think that would probably, you will have 20 or so different people asking questions around that. That will be very helpful.

>> JAMIE JENNINGS: Okay. I'm going to go through a couple use cases and skip over a couple use cases, for time so that I can show you the roadmap of things that we are already working on, in addition to some of the ideas that have come up during today's talk.

Here is a use case that we are familiar with in my team. We do a lot of mining of DevOps data, which is data about source code. It's about contributions and changes of source code. It's about unit tests and builds and deployment of what was once source code. It's about repository actions like merging and branching, and pull requests. But if you want to actually mine the source code, if you want to examine source code itself, it takes a little bit of work, and Rosie is making that easier.

There are many possible reasons why you might want to mine source code. There is a graduate student doing cool work at NC State and one of the things he is interested in is finding anti patterns in code.

If the best practice happens to say don't do X, and we can pattern matcher on a whole bunch of source codes to look for instances of X, we can bring that to people's attention and say this is against best practice so don't do this. It's also useful for detecting security vulnerabilities, and our X force team does this kinds of thing all the time.

The obstacle to doing anything with source code is that there is so many different languages that you might want to parse and for most of those, there is no parser available. For some, there are, like you can get a Python parser, you can get a Go parser. Go is probably the exception because they made the Go parser easy to use. But for a lot of languages, if a parser is available it's hard to integrate into your work flow.

In other words, if I want to analyze a repository and the repository has Python, JavaScript and Go in it, then there might be parsers available for those but three different implementations that I have to support and run. They are going to generate three different kinds of output that I have to deal with.

Our solution is write Rosie Pattern language to extract out just the parts of the source code that you want. You don't have to parse the entire language. I back this claim up by citing a recent paper from Stanford from last year, which does a really cool thing. They are doing static analysis of code.

They said, look, we are just going to write some Haskell code to parse out fragments of programs. They call them microgrammers. Fragments of programs like the if/then clause that you see in the little array.

Without having to parse an entire C language file, which is difficult, you can still do something useful by extracting out the pieces that you are interested in, and then working on those.

And working with another group of students at NC State, NC State is doing a great job, I say, lots of collaboration going on with IBM. But anyway working with a group of students there, they developed patterns in about ten different languages to pull out comments and dependencies and different parts of code and we are filling in the rest of that matrix now.

This is the only use case I'm going to talk about, but this slide here demonstrates if you are trying to find things like a line comment, you want to find a line comment that is not part of, for instance, a string, like in a print up statement and see there might be a double slash. That double slash does not start a comment. In Python if you are looking for a hash comment on that is embedded in a triple coded string, that triple coded string might be a string in your program, so that line is not a comment.

When you do this kind of source code parsing you want to do

it correctly. It turns out the RPL and there is a fragment up above that is a illustration, these Rosie Pattern Language for this stuff is not hard. It is not hard to do it correctly.

If you forgive me, I'm going to skip ahead, just for lack of time. I'm so glad there was so much discussion. This slide by the way has the green stuff I was talking about before where path names come up in green.

There is a secure engineering use case which is awesome, the secure engineering use case is sanitize your input, and parse everything that you get before you use it. If you want a way to parse everything efficiently and reliably, I think you should use Rosie.

If I can go back a slide, there we go, and so that is the end of the use cases.

I want to tell you about the roadmap, and then take any last questions. This proverb has unknown provenance. If anybody can claim definitively where it comes from, I will buy you a beer. Don't say Africa. That is too general.

Here is the roadmap. We are implementing right now unicode predicates so that you can find a character class as being any predicate on unicode characteristics. The color assignments that I talked about earlier, customizations are in the works to let you define your own colors, and you can define colors for your own patterns of course, not just for the standard ones.

We are working on the microgrammer work for pulling features out of source code, and we are going to publish the RPL for that on GitHub.

There is a number of ideas in the works for making Rosie better as a command line tool, to make it more like grep. For some people that is a important use case.

Extensibility, we didn't talk about macros but they are cool. They let you do, macros do in Rosie what they do in other languages. You find something you are doing repeatedly, you find yourself repeating the same things when you type an expression, then you should be able to write a macro, so that you can type something once and have it expand into the bigger thing that you needed.

We are working on automatic pattern generation, again that is something we are doing in collaboration with NC State but also in other areas, and I would welcome collaborators on that. Finally, compiler optimization, Rosie's performance I have to say is really good, I'm biased of course but the numbers are looking good. Yet, there are still opportunities for compiler optimization.

Because Rosie was written with the structure of a traditional compiler, we have an opportunity to use compiler optimizations like common special elimination to simplify patterns so that internally when Rosie compiles what you wrote,

you don't have to change anything that you write, but Rosie can compile it better so that it will execute even faster.

That's the set of items that are already sort of being designed or being worked on or being thought about, and I would welcome ideas and contributions for any of those things.

So I will leave you with this. Please join the Rosie community. Clone the repository and type make and you can start doing some of the things that I showed in this talk.

I'm looking for people who want to contribute patterns to our open source, repo, who want to write tools, we can use a linter that would be really nice, people who want new features, port languages. I look forward to hearing from you on GitHub through the issues or on Twitter at Jamie the riveter which is also on the title slide.

Last set of questions.

>> This one may be a great question you can do through your GitHub, but is [inaudible] source doing the same thing or what is the difference? You talked a little about that, differences or -- (overlapping speakers) specific that, you can do that later, maybe.

>> JAMIE JENNINGS: Yeah, that would be interesting to talk about off line. App scan does some of the same things, not using Rosie at the moment. But I sense an opportunity.

>> Could you use Rosie not only to parse the code but also

ensure a specific piece of code, for instance a security code is embedded in all [inaudible] for instance, if you have parsed a file and you have updates to all those legacy code, could you somehow do that.

>> JAMIE JENNINGS: Yeah, absolutely. It sounds like it would be a straightforward small Python program, or a not too hard shell script actually. I think that is a really good use case.

>> (overlapping speakers).

>> I hate to interrupt. But we are at the top of the hour. So I think we need to wrap it up. Thank you very much, Jamie, for a great presentation, and Joe, for answering those questions. Next week we have two tech talks scheduled, one for November 13, it's going to be about tin pac patterns and the one for November 15 on gaming. Please do come back and join us for those next week. Until then, thank you for your time, until then be well. Good-bye.

>> JAMIE JENNINGS: Thanks, everybody. Bye-bye.

(end of call at 11 :01:00 a.m. CST)

Services Provided By:
Caption First, Inc.
P.O. Box 3066
Monument, CO 80132
800-825-5234
www.captionfirst.com

***

a totally verbatim record of the proceedings.

***