Perl and Undecidability: Perl Undecidable

by Jeffrey Kegler
jeffreykegler@mac.com

This is the last of three articles based on a formal proof of Perl's unparseability that I originally presented on Perlmonks. This winter I retired to the edge of a frozen New England lake to work full-time on a parser generator, one that would generate a parser from any grammar describable in BNF. No such tool is in general use. I hope to create one. An alpha version is on CPAN as `Parse::Marpa`.

Looking for test cases, I considered Perl 5. This led me to Adam Kennedy's `PPI` documentation and his suggestion of how to prove that Perl parsing is not decidable. For some reason, it was not immediately obvious to me that Adam was right. Perhaps my optimism about parsing Perl 5 came from routinely walking on water (frozen lake, remember). In any case, to be convinced, I needed to work the proof out formally for myself.

I posted the result on Perlmonks. Since the Perl unparseability proof is of practical interest rather than theoretical significance, I tried to make my write-up accessible to readers who don't care about math for its own sake, but who do care about things that are of practical use. The Perlmonks posting attracted enough interest for me to be invited to write this series.

The first of these articles proved the Halting Theorem using Perl notation. One purpose was to explain the techniques I would need in the other two articles, but it should not be forgotten that the Halting Theorem itself is an extremely practical and useful result. Imagine for a moment that the existence of unsolvable problems was known only to experts in universities. Imagine in particular that it was not generally known that I can't write a program to find infinite loops in arbitrary code. A lot of time would be wasted.

The second article dealt with Rice's Theorem, a quick and handy rule for spotting undecidable problems. Most programmers know that there are undecidable problems, and that some of them are practical questions. Less well known is just how common undecidability is. Any non-trivial question about what a Perl script does is undecidable, and the same is true of all general-purpose programming languages.

See parts I and II of this series in The Perl Review, Spring 2008 and Summer 2008

■ The proof by way of Rice's Theorem ----------

Perl is unusual among general-purpose languages in that not just Perl's run phase behavior, but also its parsing is, in the general case, undecidable. The second article contained a Perl unparseability proof. The proof used Rice's Theorem, which had several advantages.

1. It was short and quick.

2. It is closest to how the proof would look in a journal if it were a publishable result. (An academic math journal would not print this result because the referees would consider it obvious. They would also reject it because practical programming languages can have limited lifespans, and the journals want results that will be relevant and readable, decades from now.)

3. Using Rice's Theorem, the second article also proved that a wide variety of other questions about Perl were undecidable, showing that the situation with Perl parsing and the Halting Question is far from rare.

The disadvantage of using Rice's Theorem is that it is a bit of a "black box". It might leave me without any feeling for why Perl is not in general parseable.

This article presents two more proofs. Each lifts the cover of the black box. The first is direct, that is, it avoids the traditional approach of reduction to the Halting Theorem. Instead it assumes the existence of a general solution to Perl parsing and uses an example Perl script to show that this assumption simply can't be true. The second proof in this article takes the traditional approach, showing that a general solution to Perl parsing requires a general solution to the Halting Question, which is known (and which was proved in the first article of this series) to be undecidable.

Both the proofs in this article employ a reduction to absurdity —they assume something, and show that the assumption creates a contradiction. This constitutes a proof that the assumption must be false, and that therefore the opposite of the assumption must be true. Ordinary reasoning uses this kind of logic all the time ("If this jerk knows so much about startups, how come he needs us to pick up the tab for lunch?"). But for some reason, when presented in its raw form, reduction to absurdity can seem strange.

■ How not to hit it off with a girl at a party----------

Once upon a time there was a beautiful ballerina. She spotted a mathematician at a party, felt an instant attraction, walked up to him, and said, "It must be wonderful to be able to do mathematics. I could never imagine following things like proofs." The mathematician insisted that, since she was very intelligent, she was certainly capable of not just understanding proofs, but of appreciating the beauty some of them have.

The mathematician had memorized some lines of Edna St. Vincent Millay for just such an occasion:

Euclid alone
has looked on beauty bare. Fortunate they
Who, though once only and then but far away,
Have heard her massive sandal set on stone .

As he quoted these lines, the mathematician felt the blood rush to his face. The ballerina did not notice, or pretended not to. Reassured, the mathematician led the ballerina through Euclid's short, elegant proof that there are an infinite number of primes. (An extremely intelligent and widely read woman, the ballerina already knew that a prime number is a number divisible only by itself and 1.)

The mathematician met the ballerina's eyes. They were violet and registered shock. He asked her what part of the proof she didn't understand. She told the mathematician she understood him perfectly, thank you very much.

She pointed out to the mathematician that he had started by saying that since the primes were not infinite, there must be a fixed number of them, and that therefore one of them must be the largest. That was okay, she said.

But then he had acted as if he really believed in the largest prime. He kept pretending until there was a contradiction. That was really what he was after, the contradiction.

Once he had it, he had suddenly changed his tune. He placed the entire blame for the contradiction on the largest prime and discarded it. In fact, he had never really believed in the largest prime, and was just using it the whole time.

She told the mathematician he was a vile little man, and that she would be glad to hear nothing more of him or his proofs. As she walked out of his life forever, the mathematician gazed, mesmerized by the patterns her shoulder muscles made as they rippled across her back.

■ A direct proof ----------

In my proof of Perl unparseability, I will, like the unlucky mathematician, assume that something exists. Then I'll use it to create a Perl script as a counter-example. The counter-example will show that my assumption creates an impossible situation. Since it is impossible for the assumption to be true, the assumption must be false, and that will be my proof.

What I am going to assume exists, is a function named `Acme::Halt::f_nullary()`, which takes two arguments. The first argument is a string containing the name of a file, and the second is a string naming a Perl function. I assume that `Acme::Halt::f_nullary()` returns 1 if, in the file named in the first argument, the function named in the second argument is nullary. I assume that `f_nullary()` returns 0 otherwise.

In the proofs, I treat Perl as if it were equivalent to a Turing machine. Turing machines are the theoretical equivalent of general-purpose computing. Technically, Turing machines don't allow any extensions that produce unpredictable behavior. They could not, for example, simulate the unpredictable aspects of networking. Even the `time` function is, strictly speaking, unpredictable from the point of view of the program, and therefore goes beyond Turing equivalence.

As a practical matter, these undecidability proofs apply just as much to full-featured Perl as they do to its Turing equivalent subset. I'll return to this point.

My counter-example requires `f_nullary` to run in the compile phase. I can be sure it will, since I'm assuming a Turing-equivalent variant of Perl. That is, suppose Perl could determine if a function in a file was nullary, but it required a special Perl built-in available only in the run phase. Even though I can't use that built-in in the compile phase, I know I can simulate the special built-in. I know this because I have Turing-equivalent processing available in the compile phase.

The following counter-example Perl program shows that it is impossible for there to be a `f_nullary` function that behaves as described.

In *direct.pl (Code listing 1, next page)*, I show some ordinary Perl code, and it should run as long as `f_nullary` actually exists. The first observation the Unkind Reader might make is that, if this code proves anything, it's that I don't know how to write a useful Perl program. Let me go back to a point I made in the first article.

The code in these articles, unlike most of the code in *The Perl Review*, is not trying to be useful. This Perl code is intended as the subject of a thought problem, an aid for thinking out an issue. It's like Schödinger's cat, locked into a box with flask of poison gas under a hammer sensitive to quantum mechanical effects. Schödinger's intent was to pose a thought problem, dealing with Heisenberg's uncertainty principle and what it means at the macro level. Schödinger was not trying to deal with the issue of stray cats.

In *direct.pl*, as in the previous articles, I prove unparseability by giving a specific example of an ambiguous Perl parse. The example, which I owe to a Perlmonks post by ikegami:

```
dunno + 4
```

Where `dunno` is a function, this can be parsed in one of two ways. If `dunno` is prototyped as a nullary function (one which takes no arguments), the plus sign is parsed as a binary operator. Otherwise, the plus sign is parsed as a unary operator, and `+4` is treated as an argument to the `dunno` function.

The result of the expression `dunno + 4` can easily be different, depending on the parse. When `dunno` is prototyped nullary, its return value is added to 4. Otherwise, the return value of `dunno + 4` is the return value of `dunno`, and it's up to `dunno` what is done with the `+4`. In my examples, `dunno` ignores its argument.

Code listing 1: The code for direct.pl

```
 1 use 5.010;
 2 use warnings;
 3 use strict;
 4 use Acme::Halt;
 5
 6 sub runtime_nullary {
 7     my $function  = shift;
 8     return 0
 9       if not defined (my $prototype = prototype $function);
10     return $prototype eq q{};
11 }
12
13 BEGIN {
14     *dunno =
15         Acme::Halt::f_nullary( __FILE__, 'dunno' )
16         ? sub   {0}
17         : sub() {0};
18 }
19
20 print 'nullary dunno: ';
21 say runtime_nullary('dunno') ? 'yes' : 'no';
22
23 print 'result is ';
24 say dunno + 4;
```

The file *direct.pl* is my counter-example. If `f_nullary` exists as defined, the existence of *direct.pl* leads to an impossibility. Here's how.

If at line 15 `f_nullary` examines *direct.pl* and returns 0, indicating that *direct.pl* establishes a non-nullary prototype for `dunno`, then in lines 14-17, *direct.pl* will actually set `dunno` up with a nullary prototype. This is a contradiction, so `f_nullary`, as defined, can never return 0.

What if `f_nullary` returns non-zero at line 15? In that case lines 14-17 will set `dunno` up with a nullary prototype. But a non-zero return from `f_nullary` at line 15 by definition means that *direct.pl* does not set up a nullary prototype for `dunno`. This is a contradiction, so `f_nullary`, as defined, can never return a non-zero value.

Since `f_nullary`, as defined, cannot return either zero or a non-zero value, it cannot exist as defined.

The problem with `f_nullary` is not its name. If I rename it `widget` the contradictions remain. Similarly, I can play around with different return values. If I rewrite `f_nullary` to return 42 for a nullary function and 711 for a non-nullary, the test in lines 14-17 might no longer be an elegant ternary. But clearly, if I change minor details of the return values of `f_nullary`, *direct.pl* can also be changed to produce the same contradiction.

The problem with `f_nullary` is its claim to be able to tell whether a file sets a function up with a nullary prototype or not. As long as the definition continues to make that claim, some counter-example very like *direct.pl* I will show that that claim is impossible.

■ Turing equivalence -------

In these proofs I assume that Perl programs are Turing equivalent. In particular I assume that, given a Perl program and its history to any point, what happens after that point will be entirely predictable. This predictability must be not just from some external meta-viewpoint, but from the point of view of the program. Of course, in real life Perl has extensive capabilities which expose it to, or even try to create and exploit, unpredictability.

Perl's `rand` is an example. Technically, it is not random, but pseudo-random. From a point of view that includes the system clock, `rand`'s results are completely predictable. But from the point of view of the Perl script, `rand` is usually unpredictable enough. Similarly, even when network behavior is predictable from a viewpoint that encompasses multiple nodes and their network connections, the network will often be unpredictable from the point of view of the node running the Perl script.

Even though I assume Turing equivalence, these undecidability results apply to Perl as a whole, whether its behavior is Turing equivalent or not. That's because, in practice, all our modern extensions beyond the Turing model create more undecidability. None of them reduce it.

As one example, suppose I allow, as an extension to Turing equivalent Perl, the full use of `rand`. With `rand` available, I can wrapper pieces of the Perl code:

```
if (rand(2) > 1) { ... }
```

Predictable behaviors in the original Perl scripts become unpredictable behaviors in the wrappered versions.

Theoretically, extensions to Turing machines might reduce undecidability. Turing discussed the possibility of "oracles". Turing's oracles could, whether intuitively, by exercising supernatural powers, or by some other means, accurately decide undecidable problems. But Turing doesn't seem to have expected anyone to invent a Turing oracle, and if anyone has, they're keeping it quiet.

The assumption of Turing equivalence is convenient for the proofs. But there's a more important motivation for sticking

Code listing 2: the halts subroutine

```
 1 sub halts {
 2     my $machine                = shift;
 3     my $input                  = shift;
 4     my $code_string_to_analyze = qq{
 5         BEGIN {
 6             run_turing_machine("\Q$machine\E", "\Q$input\E");
 7             sub whatever() {};
 8         }
 9     };
10     s_nullary( $code_string_to_analyze, 'whatever' );
11 }
```

to Turing equivalence. Undecidability, which is caused by functions which go beyond the Turing model, can be avoided by using those non-Turing functions carefully or not at all. The undecidability in the proofs I present in this articles comes from Perl's most basic control constructs. That means the implications of undecidability are harder to avoid. Sticking to Turing equivalence makes undecidability proofs more relevant to real life, not less so.

■ Compiling versus running ----------

As described in *Programming Perl*, Perl runs in two major phases. The first phase is called the compile phase and the second is called the run phase. Most (but not all) of what goes on in the compile phase is compilation. Most (but not all) of what goes on in the run phase is execution.

Code execution in the compile phase happens, for example, in `BEGIN` blocks. `BEGIN` blocks are compiled along with other code during the compile phase, but unlike the other code, `BEGIN` blocks are executed immediately, without waiting for the run phase. Run-phase compilation occurs, as an example, in code supplied to string `eval`'s.

When it's necessary to refer to the kind of processing that is actually begin done, as opposed to the phase, *Programming Perl* speaks of compile time and run time. So assignments inside a `BEGIN` block take place in the *compile phase* but at *run time*. Inlining of constant subroutines defined in a string `eval` takes place in the *run phase*, but at *compile time*.

The distinction between "time" and "phase" is not always made clearly. The `perlmod` man page uses the terms compile-time and run-time in places where *Programming Perl* would insist that the correct terms are "compile phase" and "run phase". As a result, `perlmod`'s explanations of what happens when can be hard to follow.

■ Does `runtime_nullary` show the proofs are wrong?

In *direct.pl*, I included a routine to test the results: `runtime_nullary()`. It returns 1 if its argument string is the name of a nullary function, and 0 otherwise. `runtime_nullary()` really exists. I give the code—it's all basic Perl and Perl built-ins, used as documented. And I've tested it. Doesn't the very real existence of `runtime_nullary()` mean that it is decidable whether a function has a nullary prototype? And doesn't that mean there must be something wrong with each of my three proofs?

Specifically, even if I have a Perl program not parseable in the compile phase, why can't I do the following?

- Add logic to list all functions with nullary prototypes in the Perl program. Put this logic where it will be executed at the end of the run phase. (In so doing, I must not alter the parse of the original code. With caution that should be possible.)

- Next, run the Perl program through the compile phase and the run phase, including through the new logic that produces the nullary prototype listing.

- Take the nullary prototype information produced by this first run. Feed it into a second pass over the Perl script which uses the nullary prototype information to decide the parse.

This two-pass method is kludgy, but it is similar to the way some text processors create indexes and cross-references. At least one text processor which works this way sees widespread, if not complaint-free, use.

First, a weird quibble

The first problem I'll point out with the two-pass method is, I'll admit, a bit of a quibble. Perl allows me to change the prototype as I proceed through the compile phase.

In particular, there can be multiple `BEGIN` blocks, each setting a different prototype for the same function. The setting of prototypes can even be conditional. Perl tells me when I redefine subroutines, and squawks even louder when I change the prototype, but I can turn off both of these warnings.

`runtime_nullary` will only report the most recent prototype. This may not have been the one which was in effect when most of the Perl code was parsed.

At this point, I may say, "That's possible, but it's just such a weird corner case, let's ignore it." It is pretty weird, actually. Fine. I'll ignore it.

Second, run phase prototype changes don't count

There's a more serious obstacle to the two pass method. Function prototypes set up after the compile phase aren't used in parsing. They don't count.

To be precise, they don't count in most cases. In those places where compile-time happens during the run phase, it uses any function prototypes that were set up earlier in the run phase.

But for most run phase processing, the code has already been compiled when the run phase begins. Changes made to function prototypes do not affect the parse, and information about run phase changes to function prototypes is useless.

Third, there's the Halting Problem

The most basic problem with using `runtime_nullary` to assist in parsing, is that there is no way to ensure, in general, that `runtime_nullary` will ever be called. The Perl script might be an infinite loop.

In this direct proof, we didn't reduce Perl parsing to the Halting Question, but the issue of whether or not code is an infinite loop never goes away. And the Halting Question lies behind a question that is commoner in practice: For some length of time N, where N is too long for a direct test to be practical, does some arbitrary code take time N or longer to run? Proving undecidability by reduction to the Halting Question tackles the issue of infinite loops directly, rather than trying to deal with them as a side issue.

■ A traditional proof----------

Here is the Perl unparseability proof in the traditional form of a reduction to the Halting Question. First, I will prove Kennedy's Lemma:

If you can parse Perl, you can answer the Halting Question

I call this Kennedy's Lemma, because I first saw it stated in Adam Kennedy's `PPI` documentation.

The proof of Kennedy's Lemma is another reduction to absurdity. Once again I assume I have a routine that returns 1 if a subroutine has a nullary prototype, and 0 otherwise. This time I assume it is named `s_nullary`, that it analyzes a string of Perl code that is its first argument, and that the name of the subroutine is its second argument.

Perl is Turing-complete. I will also assume that some helpful and theoretically-oriented chap has written a subroutine named `run_turing_machine` which takes a Turing machine representation as its first argument, and input for that Turing machine as its second argument.

On these two assumptions, I can write a Perl subroutine I name `halts` (*Code listing 2*, previous page), which solves the Halting Question for an arbitrary Turing machine with arbitrary input.

Specifically, `s_nullary` in line 10, in order to figure out whether `whatever` is given a nullary prototype, has to figure out whether line 7 is ever executed. To do this `s_nullary` must somehow figure out whether line 6 will ever finish. Line 6 runs a arbitrary Turing Machine with arbitrary input, and so in order to know the prototype of `whatever`, `run_turing_machine` must be able to solve the the Halting Question. This proves Kennedy's Lemma.

With Kennedy's Lemma proved, a simple reduction to absurdity proves that Perl is unparseable. By Kennedy's Lemma, if I can parse Perl, I can solve the Halting Question. But I can't solve the Halting Question. Therefore I can't parse Perl.

■ Conclusion ----------

These articles have shown that, in general, Perl parses are not decidable during Perl's compile phase, and that carrying the problem over into the run phase will not help. Perl cannot always parse Perl. And, however I define static and dynamic, Perl is not, in general, either statically or dynamically parseable.

Perl's unparseability comes from one of its basic, deepest properties—it gives Turing-complete power to the programmer at compile time. With Turing-completeness comes undecidability.

Perl unparseability is not a bug or a misfeature. It's an inseparable aspect of a feature—Perl's full power is available when setting up its own compilation environment. This is a valuable feature, purchased mainly with the depreciated currency of theoretical purity.

■ References----------

Programming Perl, 3rd Edition by Larry Wall, Jon Orwant, and Tom Christiansen discusses Perl compilation in Chapter 18. It sets out a distinction between compile/run time and compile/run phase. The *perlmod* man page attempts to cover the same ground.

Randal Schwartz's Perlmonks node "On Parsing Perl" was a important development in this discussion: *http://www.Perlmonks.org/?node_id=44722*

ikegami's elegant example of parsing ambiguity in Perl from Perlmonks: *http://Perlmonks.org/?node_id=688260*

"Perl Cannot Be Parsed: A Formal Proof", my original Perlmonks post on Perl's unparseability: *http://www.perlmonks.org/?node_id=663393*

"Perl and Undecidability", The Perl Review 4.2 (Spring 2008), by Jeffrey Kegler, is part 1 of 3 in this series.

"Rice's Theorem", The Perl Review 4.3 (Summer 2008), by Jeffrey Kegler, is part 2 of 3 in this series.

My general BNF parser, currently in alpha: *http://search.cpan.org/dist/Parse-Marpa*

Adam Kennedy's PPI module: *http://search.cpan.org/dist/PPI*

■ About the author ----------

Jeffrey Kegler has been using Perl since 1987. At the time, he'd bid a fixed-price gig and took a chance that the newly-released Perl 1 would be better than shell scripting. Betting on Larry Wall proved to be a good move. Jeffrey is the author of the `Test::Weaken` and `Parse::Marpa` CPAN modules.

Jeffrey is a published mathematician, has a BS and an MSCS from Yale, and was a Lecturer in the Yale Medical School. In 2007 he published his first novel, *The God Proof*. It centers on a little known part of Kurt Gödel's work—his effort to prove God's existence. Gödel worked out his proof in two notebooks: notebooks which were missing from his effects when they were cataloged after this death. *The God Proof* begins with Gödel's lost notebooks reappearing in a coastal town in modern California. It's available as a free download: *http://www.lulu.com/content/933192*. You can purchase print copies at Amazon: *http://www.amazon.com/God-Proof-Jeffrey-Kegler/dp/1434807355*.