On unlexable programming languages

Robert J. Simmons

April 1, 2011

Abstract

One of the features of the Perl programming language is that it is formally unparsable. In this article, we consider the design of a programming language that is similarly unlexable, in that the correct separation of a string of characters into lexical tokens is undecidable in general.

1 Introduction

The inspiration for this paper was a factual claim and an opinion, both of which I attribute with some uncertainly to Larry Wall's October 2010 lecture at Carnegie Mellon University. Both claims, however, are also definitively attributed to Jeffrey Kegler. The factual claim is that it Perl 5 is an unparsable language [Keg08a]. The opinion is that this is a good thing [Keg09]. If a language with undecidable parsing is good, a language with undecidable lexing must be freaking awesome. Hence, this paper.

2 An undecidable family of lexers

The recipe for Perl's unparsability is rather straightforward: Perl is unparsable because ``the only way to parse Perl 5 is to run it or to simulate it using a language of equivalent power'' [Keg08b]. Specifically, the use of `BEGIN` blocks can force code evaluation during the compilation phase and the `eval` function can trigger compilation at runtime [Keg08a].

Kegler, who established Perl's formal unparsability, was unamused that ``Perl-bashers'' picked up his result as a criticism of Perl [Keg09], declaring that Perl's unparsability is a good thing, and that, in fact, ``demanding a parseable language is the sign of weak programmer.'' The underlying truth of this statement is that Perl's unparsability means that Perl can have no meaningful notion of abstract syntax, so that it is impossible to contemplate static analyses, factoring tools, or IDE feedback that works in general on Perl programs, since such tools uniformly work on the level of abstract syntax. Perl can only be interpreted, not compiled or, in a certain sense, reasoned about.

How can we apply similar principles to the design of a programming language with undecidable lexing? In Perl, it is the `BEGIN` blocks which bootstrap parsing into potentially problematic Turing tarpits. We define the *lexing problem* to be the process of taking a stream of *characters* and unambiguously returning either an error or a single lexical token and a sub-stream of the original stream. Lexers

_	0	9	0001	j	111	t	00011111	E	10110	P	01011
0	00	a	1	k	11100	u	00000111	F	10101	Q	01010
1	1111111	b	0110	l	001100	v	11011	G	10100	R	01001
2	01	c	1010	m	110101	w	11010	H	10011	S	01000
3	0111	d	0010	n	101	x	11111	I	10010	T	00111
4	1011	e	1110	o	001	y	11110	J	10001	U	00110
5	0011	f	11	p	100	z	11101	K	10000	V	00101
6	1101	g	010	q	000	A	10	L	01111	W	00100
7	0101	h	110	r	10101010	B	11001	M	01110	X	00011
8	1001	i	011	s	11111000	C	11000	N	01101	Y	00010
						D	10111	O	01100	Z	00001

Figure 1: Standard encoding of letters as bitstrings from the literature [Sim11].

deal with potential ambiguity by always selecting the *longest possible lex*; this is how we ensure that that the C token `elsebob` always parses as a single identifier and not as the reserved word `else` followed by the identifier `bob`.

We specify that the lexer returns a sub-stream to ensure that lexing can only read from the character stream, not add to it.[1] In this paper, we will assume that the character stream available to the lexer is a byte sequence - which we intuitively connect to the standard ASCII encoding of characters.[2]

2.1 The Dec/n languages

The parametrized Dec/n family of languages - where n is a Gödel numbering of the three parameters, as yet unspecified - all share the following properties. Tokens such as `*`, `~`, and `^` are lexed individually, but an identifier is a string of alphanumeric characters (plus underscores) *such that no prefix corresponds to a non-terminating lambda calculus expression*. This, combined with the requirement that lexers return the longest possible token, is the reason why any perfect Dec/n lexer must solve the halting problem. For instance, in the Dec/5 language (defined below), the alphanumeric sequence `a1qa1KLaffq01X1uas0foo` parses as two tokens, `a1qa1KLaffq01X1uas0` and `foo`, since the former corresponds to a non-terminating expression.

A particular member of the Dec/n family is defined by three parameters. The first is a way of interpreting a series of alphanumeric characters as a bitstrings (by giving a bitstring encoding for each alphanumeric character in turn).[3] The second is a way of encoding bitstrings as lambda calculus expressions. The third is an evaluation strategy to use to attempt normalization.

2.2 Dec/5

The Dec/5 language is a specific instantiation of the Dec/n family. Characters are encoded as bitstrings using a standard character-by-character encoding from the literature [Sim11], even though this encoding does have the potential disadvantage that many sequences of alphanumeric characters map to the same bitstring. For convenience, we repeat this encoding in Figure 1.

The second piece for instantiating Dec/5 is a way of encoding bitstrings as lambda expressions. We use the encoding from the Jot programming language [Bar]. Jot actually presents itself as a full programming language, not just an encoding of lambda-calculus terms, but its termination behavior is dependent on the evaluation strategy of the host language [Rey72]. This is nevertheless perfect for our purposes: we use Jot merely as one potential encoding of (a subset of) the terms of the untyped lambda-calculus. Dec/5 uses a call-by-value evaluation strategy to attempt to normalize terms to a value.

One useful aspect of the standard encoding of alphanumeric sequences is that any combinator calculus term can be encoded directly as an alphanumeric sequence by writing application (prefix) as ``a'' and the S and K combinators as ``s'' and ``k'' (respectively). For instance, the token `aaksask` can be extended by virtue of the fact that $(KS)(SK)$ is a terminating combinator calculus term [Bar].

3 Implementation

In Figure 2, we can see the signature allowing us to instantiate an (necessarily incomplete) lexer for any Dec/n language. The particular implementation can only prove non-termination by detecting a cycle in evaluation. All divergent terms will fail to be noticed by the implementation and will either cause the implementation to diverge (if `limit` is `NONE`) or else raise an exception.

The implementation is available from `https://bitbucket.org/robsimmons/dec-n`.

4 Conclusion

This paper only scratches the surface of undecidable lexing techniques. Ever since Cohen's seminal work on the area [Coh80], Gödel encodings for programs have received insufficient attention. Our Jot encoding is the most disappointing aspect of the implementation: not only does it fail to capture all lambda calculus terms (merely allowing the simulation of all combinator calculus terms), but the encoding is *boring* from the perspective of non-termination: almost all encodings terminate. As an example, the Dec/n implementation can lex all of Shakespeare's *Othello* (downloaded from Project Gutenberg) under the Dec/5 instantiation without ever finding a non-terminating term.

[1] Investigating ultralexing, where the lexer can modify the character stream, is an exciting direction for future work.

[2] Investigating undecidable UTF encoding techniques is an exciting direction for future work.

[3] Investigating non-compositional encodings of alphanumeric strings as bitstrings is an exciting direction for future work.

```
signature LEX_ARGS = sig

   (* Maximum number steps to look for a cycle or an
    * irreducable expression.
    * NONE - the lexer will not terminate on a divergent sequence
    * SOME n - only for n steps, then raise an exception *)
   val limit: int option

   (* The encoding of a given character as a bitlist.
    * Most-significant-bit first! *)
   val charcode: char -> bool list option

   (* The encoding function that turns bitlists
    * into lambda calculus expressions. *)
   val lambdacode: bool list -> Lambda.exp

   (* A step function for the lambda calculus;
    * NONE implies termination *)
   val step: Lambda.exp -> Lambda.exp option

end
```

Figure 2: The LEX_ARGS signature, parameterizing undecidable lexers.

References

[Bar] Chris Barker. Iota and jot: the simplest languages? http://semarch.linguistics.fas.nyu.edu/barker/Iota/.

[Coh80] Norman H. Cohen. Gödel numbers: a new approach to structured programming. *SIGPLAN Notices*, 15:70-74, 1980.

[Keg08a] Jeffrey Kegler. Perl is undecidable. *The Perl Review*, 5:7-11, 2008.

[Keg08b] Jeffrey Kegler. Rice's theorem. *The Perl Review*, 4:23-29, 2008.

[Keg09] Jeffrey Kegler. Unparseability is a good thing, 2009. http://www.perlmonks.org/?node_id=790624.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717-740. ACM, 1972.

[Sim11] Robert J. Simmons. On unlexable programming languages. In *SIGBOVIK 2011*, pages 79-82. ACH, April 2011.