Taming the Length Field in Binary Data: Calc-Regular Languages

Norina Marie Grosch, Joshua Koenig, Stefan Lucks
Bauhaus-Universität Weimar

***Abstract*—When binary data are sent over a byte stream, the binary format sender and receiver are using a "data serialization language", either explicitly specified, or implied by the implementations. Security is at risk when sender and receiver disagree on details of this language. If, e.g., the receiver fails to reject invalid messages, an adversary may assemble such invalid messages to compromise the receiver's security.**

Many data serialization languages are length-prefix languages. When sending/storing some F of flexible size, F is encoded at the binary level as a pair $(|F|, F)$, with $|F|$ representing the length of F (typically in bytes).

This paper's main contributions and results are as follows. (1) Length-prefix languages are *not context-free*. This might seem to justify the conjecture that parsing those languages is difficult and not efficient. (2) The class of *"calc-regular languages"* is proposed, a minimalistic extension of regular languages with the additional property of handling length-fields. Calc-regular languages can be specified via *"calc-regular expressions"*, a natural extension of regular expressions. (3) Calc-regular languages are *almost as easy to parse as regular languages*, using finite-state machines with additional accumulators. This disproves the conjecture from (1).

Keywords: Security, Language Theory, Data Serialization

I. Introduction

Entities who communicate use a "language" – even when the entities are computers and the communication is binary. When the length $|F|$ of a data field F is unknown to the receiver, the sender must either append an "end-postfix" (an end-of-line sequence, a final quotation mark, ...) to F, or prepend a "length-prefix" (a representation of $|F|$). End-postfix languages match established approaches from Formal Language Theory well. On the other hand, a sound formal analysis of "length-prefix" languages is still missing, and the parsing of such languages is a common source for security vulnerabilities.

A. Heartbleed.

There are plenty of examples of security vulnerabilities caused by improper parsing length-prefix languages. One is the famous Heartbleed bug: to check if a party B is still alive, party A sends a "heartbeat" package to party B, using the ASN.1 binary interface [8]. A well-formed heartbeat package consists of

- a type field (indicating the "heartbeat" type),
- a field for the length ℓ,
- a challenge C of length $|C| = \ell$, and
- an arbitrary number of padding bytes.

B shall respond C to A, without padding bytes. But B must ignore ill-formed packages. The bug was the lack of bounds checking for ℓ. A malicious A could send a short heartbeat package with a large ℓ. Instead of ignoring this ill-formed package, B's ℓ-byte response would then compromise almost ℓ bytes from B's internal memory. Proper bounds checking might avoid such bugs, but, as the Heartbleed bug shows, this should not be left to human programmers and reviewers: the bug had been introduced into the OpenSSL sources by a programmer, survived an independent code-review by one of the core maintainers, and made it into version 1.0.1 of Open SSL 2012. It took two years until this catastrophic bug was eventually discovered, fixed and made public [20].

B. Netstrings and other Data Serialization Languages.

In the world of programming languages, length-prefix notation for strings is an almost forgotten oddity from the early days (see the appendix). But for binary communication, length-prefix notation is common. E.g., netstrings [2] represent a string of length N by the following sequence:

- the number N (in decimal notation),
- the colon as a non-digit separator character ":",
- the N-byte string itself,
- and a final comma (",").

"HELLO WORLD" is encoded as `11:HELLO WORLD,` including the comma after "D". See Section VIII for a specification of netstrings using a newly introduced notation for calc-regular expressions.

Netstrings can be nested ("used recursively" [2]). Lacking type fields, the receiver needs to know the nesting depth in advance. E.g., the netstring encoding of a heartbeat package with challenge C="abc" and padding "YZ" would be `8:3:abc,YZ,` (depth 1 and no type field). Similarly, a Heartbleed attack package could look like `5:9999:,`.

Dan Bernstein, the author of netstrings, motivates his approach by security benefits [2]: the "famous Finger security hole"[1] was based on reading an end-postfix string into a fixed-size buffer without bounds checking. Bernstein claims "it is very easy to handle netstrings without risking buffer overflow". This is true for unnested netstrings. But for nested ones, the need for bounds checking is back.

The number N must be written without leading zeros (except for the empty netstring `0:,`). A unique encoding for every string is security-wise beneficial. But not all netstring implementations enforce this. Even the sample code from [2], which we would consider the reference implementation, fails to reject netstrings with leading zeros.

[1]Bernstein seems to refer to the Morris worm exploits from 1988 [10].

DOI 10.1109/SPW.2017.33

In the appendix, we give a brief overview of common data serialization languages. Some use length-prefix notation for strings, others use end-postfix notation. For the representation of lengths, most languages interpret a stream of bytes as a binary number, whereas a few use a decimal representation, like netstrings. At a first glance, the decimal numbers are surprising, as is netstrings' final comma, which only seems to serve for readability by humans. But firstly, sometimes, readability by humans can still be beneficial in that context, e.g., for debugging purposes. And secondly, the representation of a number as a stream of decimal digits avoids endianness issues and thus eases interoperable implementations.

In addition to strings, or other variably-sized blobs, most data serialization languages support collections (arrays, lists, etc) of objects. Collections may also be represented by length-prefix or end-postfix notation, but one can also prefix the number of objects in the collection, rather than the collection's size. We refer to this as "count prefix".

C. PNG and other Chunk-Based File Formats.

A file format is just another kind of language to encode a message (file content) in binary. Most file formats for video, audio and images are "chunk based", i.e., they internally use length-prefix encoding for their internal data. Improper parsing of such files is a very common source of security issues[2]. See [17, Appendices A and B] for an overview of such file formats. One example is the PNG (Portable Network Graphics) format. PNG chunks are represented as follows:

- a four-byte binary representation of the length N (in big endian byte order),
- a four-byte chunk type,
- a N-byte field for the chunk data,
- and, finally, a four-byte CRC checksum, to discover transmission errors.

Again, we refer to Section VIII for a formal specification of PNG chunks as a calc-regular language.

D. Parsing.

In the context of programming languages, one often separates the process of recognizing the input in a given language into two phases. "Lexing" is turning a stream of characters into a stream of "tokens", according to some regular expressions. "Parsing" is analyzing the stream of tokens according to some grammar. In the current paper, we will just refer to the entire recognizing process as "parsing".

The specification of binary file formats and communication protocols, and their translation into executables, is not fundamentally different from the specification and compilation of programming languages. In the early days of programming languages, building a compiler was a considerable effort. According to Wirth [25], the first compiler for the programming language FORTRAN around 1956 *"was a daring enterprise, whose success was not at all assured. It involved about 18 man years of effort, and therefore figured among the largest programming projects of the time."*

Over years, the art of compiler writing became more feasible and was better understood, in line with the underlying theory: the linguist Noam Chomsky pioneered the idea of formal languages and formal grammars [4]. His main result was the famous Chomsky Hierarchy of different languages with different expressiveness. Though Chomsky was mainly interested in describing natural languages, his approach turned out to be extremely useful for Compiler Construction. Other researchers studied abstract machine models which could parse languages from the Chomsky Hierarchy, such as Finite State Machines for regular languages [12] (the least powerful class of languages in the hierarchy) and Pushdown Automata [13], [16] for context-free languages (the second least powerful class). Already in 1960, a formalism equivalent to Chomsky's context-free languages was first employed for the precise specification of the programming language ALGOL [1]. For the parsing of binary data, and, specifically, for length-prefix languages, a similar evolution should provide:

1) a formalism to precisely specify such languages,
2) a computationally weak abstract machine model to model the parsing process, and
3) an algorithm to turn the specifications into such machines.

E. Related Work.

[17, Section 7] lists tools to translate binary file formats into executables. More recent is the Hammer parsing library for binary languages [5]. This is a pragmatic approach, without trying to enhance the underlying theory.

The poor state of the underlying theory has been observed in 2009 by Kaminski et al [9], who argued that ASN.1 encoding rules [8] require "a context-sensitive parser".[3] Also, the ASN.1 encoding rules are written in English prose, open to misunderstandings and, as [9] put it, not "in a fashion conductive to implementing an ASN.1 parser with a parser generator". In 2011, Sassaman et al [15] considered a language-theoretical view of security issues and proposed to "starve the Turing beast", i.e., to employ parsers of the least computational power required to parse the language at hand, and to design languages (or protocols) parsable with minimal computational power. This also allows to verify "parser computational equivalence" [15], i.e., to formally verify that different parties use exactly the same language. The line of research has been continued in [14], [3], [11].

In 2012, Underwood and Laib [17] studied chunk-based file formats, such as PNG. They proposed attribute grammars for format specification, with attribute rules being C program fragments. This approach benefits from the ability to feed such grammars into existing parser generators. However, the usage of Turing equivalent attribute rules for language specifications seems to be an overkill. It does not match the simple structure most such languages have, and it violates the rules from [15] (which Underwood and Laib did not seem to know about). Furthermore, the approach is bound to a specific programming

[2]We invite the reader to google for "`buffer overflow png`", "`buffer overflow wav`", etc

[3]This is theoretically incorrect, but practically valid, cf. Remark 3(d,e) below.

language, and restricted to generate a recognizer for the language at hand, i.e., the code for the receiver of the message. A declarative approach might also be useful to generate or at least verify the sender's code.

F. Roadmap

This paper is organized as follows.

After the current introduction, we recall some core definitions from Formal Language Theory, related to regular expressions and regular languages in Section II. In Section III, we show that netstrings are not regular, and not even context-free. The proof easily generalizes to other languages with length-prefix and count-prefix notation. Section IV introduces a new class of languages, calc-regular languages, which are similar to regular languages but can cope with length-prefix and count-prefix notation. Similar to regular expressions, we also define calc-regular expressions in Section IV.

Regular languages can be parsed by Finite State Machines (FSMs). Section V introduces calc-FSMs, enhanced FSMs to parse calc-regular languages. We define several different types of calc-FSMs, and the strongest one is actually too powerful for our purposes. In Section VI, the relationship between calc-regular languages and languages from the Chomsky Hierarchy is studied, with a focus on deterministic and nondeterministic context-free languages. Section VII analyses the properties of the class of calc-regular languages. E.g., the concatenation of two calc-regular languages is calc-regular, while there are calc-regular languages who's union is not calc-regular.

After defining calc-regular languages and studying their formal properties, we give some hints how to apply our ideas and results for practical problem solving. Section VIII presents an initial outline for a meta-language,

- to formally *specify* calc-regular languages, without relying on English prose, and
- to use such a specification as the input for a parser generator, such that parsers for calc-regular languages would not need to be written by hand.

We give some examples for calc-regular data serialization languages and file formats specified using the meta-language. Section IX concludes and gives directions for future research.

In the appendix, we elaborate on early versions of the FORTRAN programming language, which actually used length-prefix notation for string constants, and we compare different data serialization languages with respect to their use of length-prefix, count-prefix and end-postfix notation. As it turns out, most such languages are using length-prefix or count-prefix notation and are thus not context-free. This highlights the need for an approach like ours.

II. Regular Expressions, Languages, and Finite State Machines (FSMs)

We assume the reader to be familiar with core notions from Formal Language Theory, see, e.g., [7]. But since our approach is based on extending and enhancing regular languages, we specifically recall the formalism related to regular languages. By Σ, we denote a finite alphabet of input symbols. A language of words over Σ is a subset of Σ^*.

Definition 1. Regular expression *over Σ, and the languages derived from regular expressions are defined as follows:*

- *ε is a regular expression, defining the language $L(\varepsilon) = \{\varepsilon\}$ which consists of the empty word,*
- *0 is a regular expression with $L(0) = \{\}$,*
- *for $a \in \Sigma$, a is a regular expression with $L(a) = \{a\}$,*
- *if r and s are REs, then so is $r|s$ with*
$$L(r|s) = L(r) \cup L(s)$$
(union),
- *if r and s are REs, then so is rs with*
$$L(rs) = L(r)L(s)$$
(concatenation),
- *if r is a RE, then so is $r*$ with*
$$L(r^*) = L(\varepsilon) \cup L(r) \cup L(rr) \cup \dots$$
(Kleene-star), and
- *if r is a RE, then so is (r) with*
$$L((r)) = L(r)$$
(use of brackets).

As shorthand notation, we write r^+ for rr^*, and r^i with $i > 0$ for rr^{i-1}, and r^0 for ε. No other expressions are regular expressions over Σ.

Note that we treat "ε", "0", "|", "(", ")" etc as "meta-symbols", which are distinct from all input symbols (or "terminals") in Σ. The definition of regular languages is straightforward:

Definition 2. *A language $L \subseteq \Sigma^*$ is* regular*, if a regular expression r exists with $L = L(r)$.*

The main formal tool to parse regular languages are finite state machines.

Definition 3. *A* finite state machine *(FSM) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet (as before) δ is the transition relation, i.e., a function*
$$\delta : Q \times \Sigma \to 2^Q,$$
$q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accepting states.

An input word $w = a_1a_2 \dots a_n$ is accepted by a FSM*, if there is a sequence of states $q_i \in \delta(q_{i-1}, a_i)$ with $q_n \in F$. If $q_n \notin F$, then w is rejected. If M is a FSM, then $L(M)$* is the language of words *over Σ which are accepted by M.*

Note that the acceptance definition implies the FSM to detect the end of its input.

Definition 4. *A FSM $M = (Q, \Sigma, \delta, q_0, F)$ is* deterministic*, if for every $(q, a) \in Q \times \Sigma$ there is at most one possible transition, i.e., $|\delta(q, a)| \leq 1$.*

Theorem 1 summarizes well-known results from Formal Language Theory (see any standard textbook on Formal Language Theory, e.g., [7]). Fact (a) implies that FSMs are a good tool to parse regular languages, (b) shows that for FSMs, nondeterminism does not improve computational expressiveness, (d) explains the relationship between regular and deterministic context-free languages, (e) shows that for context-free languages, nondeterminism does improve computational expressiveness. Results (c) and (f) are the famous *pumping lemmas* for regular and context-free languages, respectively.

Theorem 1. *The following are well-known results from Formal Language Theory:*

(a) A language L is regular if and only if a FSM M with $L(M) = L$ exists.
(b) If M is a FSM, then a deterministic FSM M' exists such that $L(M) = L(M')$.
(c) If L is a regular language, then P exists, such that all $w \in L$ with $|w| > P$ can be written as the concatenation $w = xuz$, of substrings $x, u, y \in \Sigma^*$, such that (1) $|u| \geq 1$, (2) $|xu| \leq n$, and (3) for all $i \geq 1$, the string xu^iy is in L.
(d) All regular languages are deterministic context-free. The language a^ib^i is deterministic context-free, but not regular.
(e) The language a^ib^j with $j \in \{i, 2i\}$ is context-free, but not deterministic context-free.
(f) If L is a context-free language, then P exists, such that all $w \in L$ with $|w| > P$ can be written as the concatenation $w = xuyvz$ of substrings $x, u, y, v, z \in \Sigma^*$, such that (1) $|uyz| \leq P$, (2) $|uv| \geq 1$ and (3) for all $i \geq 1$, the string xu^iyv^iz is in L.

III. Length-Prefix Notation is not Context-Free

Below, we prove that netstrings, even without nesting, are not context-free.

As discussed below, this easily generalizes to other forms of length-prefix notation, if the length is unbounded. Recall that netstrings start with a decimal number n, followed by ":", followed by a string of length n, finished by a ",". Without the constraint for the string to be of length n, it would be easy to write a regular expression for netstrings. If the length n would be encoded by unary notation, one could specify netstrings by a context-free language. But netstrings with decimal numbers are not context-free:

Theorem 2. *The language* **N** *of well-formed netstrings is not context-free.*

Proof. Assume the language of well-formed netstrings to be context-free. The characters "9", ":", and "," are specific terminals from the input alphabet Σ, the symbol $\langle$letter$\rangle$ represents an arbitrary terminal.

Let $w =$"9^m:$\langle\text{letter}\rangle^n$," be a sufficiently long well-formed netstring. Using the pumping lemma (i.e., Theorem 1(f)), we can write $w = xuyvz$ as a sequence of substrings $x, u, y, v, z \in \Sigma^*$, such that all $w_i = xu^iyv^iz$ are well-formed netstrings $xu^iyv^iz =$"9^{M_i}:$\langle\text{letter}\rangle^{N_i}$,".

Each w_i defines two numbers: $P_i = 9^{M_i}$ for the number encoded before the first ":", and N_i for the number of letters following the first ":", excluding the final ",". If w_i is well-formed, then $P_i = N_i$.

Neither x nor u can hold the first ":", else P_i would be the same for all i, while the N_i would be different. Thus, x and u hold zero or more "9"-digits, but nothing else. Also, z cannot hold the first ":", else N_i would be the same for all i, while the P_i would be different. The first ":" is either in y or in v. All in all, we can write P_i as $P_i = 10^{\alpha+i|u|} - 1$ for some α and N_i as $N_i = \beta + i|v|$ for some β. From $N_i = \beta + i|v|$, it follows that $N_{i+1} - N_i = |v|$ is constant, regardless of i.

Furthermore, $|u| > 0$; else P_i would, again, be the same for all i.

Using $P_i = 10^{\alpha+i|u|} - 1$, and setting $A = 10^{|u|}$, we get

$$\begin{aligned} |v| = P_2 - P_1 &= 10^{\alpha+2|u|} - 10^{\alpha+1|u|} \\ &= 10^{\alpha} * A^2 - 10^{\alpha} * A^1 \\ = P_3 - P_2 &= 10^{\alpha+3|u|} - 10^{\alpha+2|u|} \\ &= 10^{\alpha} * A^3 - 10^{\alpha} * A^2 \end{aligned}$$

The equation $P_2 - P_1 = P_3 - P_2$ is equivalent to the cubic equation $10^{\alpha} * (A^3 - 2A^2 + A) = 0$. Since $A = 10^{|u|} \neq 0$ and $10^{\alpha} \neq 0$, we can transform this into the quadratic equation

$$A^2 - 2A + 1 = 0.$$

Its only solution is $10^{|u|} = A = 1$, i.e. $|u| = 0$. This is a contradiction to $|u| \geq 1$ – and thus to our initial assumption of the language of well-formed netstrings being context-free. ☐

Remark 3.

(a) It is straightforward to generalize the proof for any base-B encoding for all $B \geq 2$. For the largest digit $D = B - 1$, replace the prefixes "999...9" by "DDD...D", and replace the powers of 10 by powers of B.
(b) The proof employs numbers "999...9" (or "DDD...D") with the same digit everywhere. Thus, the order of digits in the length representation does not matter.
(c) Similarly, the comma at the end does not matter for context-freeness.
(d) However, the proof is not applicable to languages with an upper bound on N. In fact, length-prefix languages with a maximum length are finite – and thus theoretically *context-free (and may even be regular). E.g., the bound for ASN.1 is $N \leq 2^{8*126} - 1$.*
(e) Nevertheless, we argue that it practically *makes sense to parse these languages as calc-regular languages (see below) or similar. This is comparable to, e.g., arithmetic expressions with at most N pairs of brackets for some large but fixed constant N. Even though the language is regular, the corresponding FSM would be absurdly large, and the tools for general arithmetic expressions would be much more appropriate, in practice.*

IV. Calc-Regular Languages and Expressions

A. Motivating Example

For simplicity, consider relaxed netstrings, where the length field is allowed to have leading zeros. We use ":" and "," as specific nonterminals, "$\langle$digit$\rangle$" is one of the nonterminals in {"0" ..."9"}, and "$\langle$letter$\rangle$" is an arbitrary nonterminal. The attempt to model relaxed netstrings via regular languages leads to a nonempty sequence of digits, followed by a colon ":", followed by an arbitrary number of letters, followed by a comma ",":

$$\langle\text{digit}\rangle+ : \langle\text{letter}\rangle*,$$

This, of course, misses the context-sensitive constraint that the length encoded before the first colon (":")[4] is the number of letters between that first colon and the final comma (","). To formalize such constraints, we must enhance regular expressions:

$$(\langle\text{digit}\rangle+).\text{decimal} : \langle\text{letter}\rangle^{\text{decimal}}.$$

This means that the number N, encoded in decimal by the sequence of digits before the first colon, is the same as the number of characters between the first colon and the last comma.

B. *Prefix-Freeness*

There is one problem, though. A machine parsing the above expression will *first* have to read one character beyond the final digit, *then* detect the colon *and then* conclude that the character before that colon has been the final digit. I.e., it can call the value-oracle only after having read one character beyond the last digit. We prefer to avoid this, and thus restrict ourselves *prefix-free* languages.

Definition 5. *A language L over Σ is* prefix-free*, if, for all $w \in L$, there exists no $(v, x) \in L \times \Sigma$ with $w = vx$.*

A regular expression r is prefix-free*, if the language $L(r)$ is prefix-free.*

In other words, L is prefix-free, if, for all words w in L, there is no prefix v of w in L, except for $v = w$. Specifically, if the empty word ϵ is in L, then $L = \{\epsilon\}$.

We will require the subexpressions of calc-regular languages to be prefix-free. Namely, for a subexpression like $x.f$, as in $(\langle\text{digit}\rangle+)$.decimal with $x =(\langle\text{digit}\rangle+)$ and $f =$ decimal, we will require x to be prefix-free.

Thus, we rewrite the above expression for netstrings as

$$((\langle\text{digit}\rangle+) :).\text{decimal}\ \langle\text{letter}\rangle^{\text{decimal}}.$$

This means, the final colon is, in some sense, part of the decimal number, though the value-oracle "decimal" will just ignore this non-digit.

The restriction to prefix-free languages may seem to constrain our approach, in comparison to the established results from Formal Language Theory. Theoretically, regular languages such as $\langle\text{digit}\rangle+$ and non-regular context-free languages such as $a^ib^ic^j$ are not prefix-free. But actually, the established formalism from Language Theory assumes a parser to know the end of its input. This is usually formalised by extending the alphabet Σ by an additional character $\perp \notin \Sigma$, which is assumed to be the final character of every word. This way, the input *is implicitly prefix-free*: Instead of parsing $\langle\text{digit}\rangle+$ or $a^ib^ic^j$, one then parses $\langle\text{digit}\rangle+\perp$ or $a^ib^ic^j\perp$. Thus, we claim that the restriction on prefix-free languages does not really alienate our approach from established results from Formal Language Theory – we just don't assume a pseudo-terminal like $\perp$ to mark the end of the input.

[4]Note that the subexpression $\langle\text{letter}\rangle*$ may also contain colons.

For this reason, we will claim that calc-regular languages are a superset of regular languages. Whenever a language L' is not prefix free, like $\langle\text{digit}\rangle+$, we use the same sleight of hand as Formal Language Theory does, traditionally, and assume the regular language $L = \{w\perp \mid w \in L'\}$ instead of L' itself. The "trick" is that L' with a method to discover the end of a word is the same language as L without such a method.

The "trick" is not applicable to subexpressions inside a longer expression, because inserting a $\perp$-symbol would really change the language. E.g., consider the concatenation of two numbers, i.e., the language L_2 defined by

$$\langle\text{digit}\rangle+\langle\text{digit}\rangle+$$

Even if the parser knows the end of an input word, it has no way to know when the first $\langle\text{digit}\rangle+$ subexpression ends. Thsi means that, while the language L_2 is well-defined, it is ambiguous. When parsing the word, say, "123" from L_2, there is no way to dedice if the two input numbers are $(12, 3)$ or $(1, 23)$. The restriction to prefix-free subexpressions prevents such ambiguity.

C. *Calc-Regularity*

Below, we formalise a new notation:

- $r.f$ means "read a word x form $L(r)$ and compute a value $f(x) \in \{0, 1, 2, \ldots\}$",
- $t_1\#f$ means "read a word y from $L(t_1)$, with length $|y| = f(x)$, and
- t_2^f means "read the concatenation of $f(x)$ words from $L(t_2)$".

We call the function f a "value-oracle".

Definition 6. Calc-regular *expressions r over Σ and the languages $L(r)$ derived from them are defined as follows.*

- *If r_0 is a prefix-free regular expression, then r_0 is a calc-regular expression, and the calc-regular language derived from r_0 is the regular language $L(r_0)$ (regular base-case),*
- *if r and s are calc-regular expressions, then so is rs with*

  $$L(rs) = L(r)L(s)$$

  (concatenation),
- *if r is a calc-regular-expression, then so is (r) with*

  $$L((r)) = L(r)$$

  (use of brackets).
- *If q, r, s, t_2, and u are calc-regular expressions, t_1 is either a regular or a calc-regular expression, and a function $f : L(r) \to \{0, 1, 2, \ldots\}$ is defined, then both r_1 and r_2 with*

  $$\begin{aligned} r_1 &= q(r.f)s(t_1\#f)u \quad \textit{and} \\ r_2 &= q(r.f)s(t_2^f)u \end{aligned}$$

  are calc-regular expressions.

  Given the calc-regular languages $L(q)$, $L(r)$, $L(s)$, $L(t_2)$, and $L(u)$, the calc-regular or regular language

$L(t_1)$, and a value-oracle f, the calc-regular languages $L(r_1)$ and $L(r_2)$ are defined as follows.

$$\begin{aligned} L(r_1) &= L(q)L(r)L(s)(L(t) \cap \langle\mathit{letter}\rangle^f)L(u), \\ L(r_2) &= L(q)L(r)L(s)L(t^f)L(u). \end{aligned}$$

I.e., a word $w_1 \in L(r_1)$ is the concatenation of

- *a word from $L(q)$,*
- *a word x from $L(r)$,*
- *a word from $L(s)$,*
- *a word from $L(t_1) \cap \langle\mathit{letter}\rangle^{f(x)}$, and finally*
- *a word from $L(u)$.*

Similarly, a word $w_2 \in L(r_2)$ is the concatenation of

- *a word from $L(q)$,*
- *a word x from $L(r)$,*
- *a word from $L(s)$,*
- *$f(x)$ words from $L(t_2)$, and finally*
- *a word from $L(u)$.*

For calc-regular expressions, the same shorthand-notation can be used as for regular expressions. No other expressions are calc-regular expressions over Σ.

Remark 4. *By definition, the language $L(r_0)$ is prefix-free. The language $L(t_1) \cap \langle\mathit{letter}\rangle^{f(x)}$ is prefix free, because all words from that language have the same length. The concatenation $L(a), L(b)$ of prefix-free languages $L(a)$ and $L(b)$ is trivially prefix-free.*

By induction, calc-regular languages are prefix-free.

D. Comments

Definition 6 is similar in structure to its counterpart for regular expressions, with two exceptions. We do not define the union and Kleene-star for calc-regular expressions, and we have new notations related to value-oracles: the #-notation in r_1 and the superscript-notation in r_2.

The notations for both r_1 and r_2 occur in pairs:

$$r_1 = \ldots r.f \ldots t\#f \ldots$$

and

$$r_2 = \ldots r.f \ldots t^f \ldots$$

If $x \in L(r)$, then $f(x)$ is a nonnegative integer. Whenever a subexpression $r.f$ occurs exactly one of either $t\#f$ or t^f must follow. And neither $t\#f$ nor t^f are allowed without one previous $r.f$.

The "#" notation in r_1 matches the requirement of some data field being exactly $f(x)$ letters long (in practice, usually $f(x)$ bytes). This is a very natural approach to model length-prefix notation. The superscript notation in r_2 defines a sequence of $f(x)$ words from $L(t)$. We need it to model languages with count-prefix notation.

Note that a calc-regular language $L(r)$ depends on both r and the value-oracles. We assume the value oracles to be implied and fixed. Changing a value oracle would change the language $L(r)$, even without changing the expression r. In typical applications, the value-oracles are indeed well-defined and simple, such as decimal notation, or binary representation either in big- or little-endian order, etc

V. Calc-Regular Finite-State Machines (calc-FSMs)

A. Definitions

Given calc-regular expressions r, and calc-regular languages $L(r)$, we need calc-FSMs to parse calc-regular languages. In fact, we will define different types of calc-FSMs, and we start with the seemingly most powerful Type-1 machines, which are a straightforward extension of nondeterministic FSMs from Formal Language Theory.

As we will show below, Type-1 is more powerful than Type-2. It is trivial to transform a Type-2 machine into Type 1, accepting the same language. Type-3 is equivalent to calc-regular languages. We conjecture that Type-2 is more powerful than Type-3. We show how to transform Type-3 machines into Type-2 ones accepting the same language. See Figure 1 .

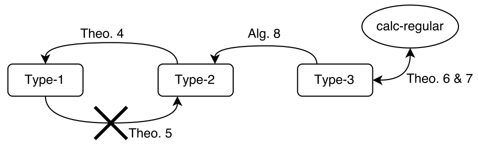


Fig. 1. Overview of calc-FSMs

Definition 7. *A Type-1 calc-FSM is a FSM $M_1 = (Q, \Sigma, \delta, q_0, F, A)$ with $k \geq 0$* accumulators *stored in the set A.*

An accumulator acc is either in string state or in numeric state. Initially, acc is in string state and holds the empty string $\varepsilon \in \Sigma^*$. $\mathcal{S} = \{$ acc.store, acc.dec, acc.count$\}$ are statements, where .dec is used for ˆ and .count for #. Statements and comparisons from $\mathcal{C} = \{ \ acc = 0$ and $acc \neq 0 \ \}$ are performed/evaluated as follows.

In string state,

> *acc.store appends the current input $a \in \Sigma$ to the string $w \in \Sigma^*$ in acc, which then holds $wa \in \Sigma^*$.*
> *If acc is in string state, and the word $w \in \Sigma^*$ is in acc, and then either acc.dec, acc.count or one of the comparisons $acc = 0$ and $acc \neq 0$ is called, then*
>
> 1) *the word w is replaced by the number $num(w) \in \{0, 1, 2, \ldots\}$, where num: $\Sigma^* \rightarrow \{0, 1, 2, \ldots\}$ denotes the corresponding value-oracle,*
> 2) *and then the statement/comparison is performed/evaluated accordingly.*

In numeric state,

> *acc.dec and acc.count decrement the number in acc by one. If acc is already zero, the input is rejected. The comparison $acc = 0$ and $acc \neq 0$ are evaluated the obvious way. If the statement is acc.store, then*
>
> - *for $acc \neq 0$, the input is rejected, and*
> - *for $acc = 0$, the state of acc is changed to string state, with acc holding the empty string, before performing acc.store.*

The transition function is a function $\delta : ((Q \times \Sigma \times 2^{(A \times \mathcal{S})} \vee (Q \times \mathcal{C})) \rightarrow 2^Q$.

Given an input $w = a_1 a_2 ... a_n$ is accepted by a Type-1 calc-FSM, if there is a sequence of states $q_i \in \delta(q_{i-1}, a_i, s_i)$ with $q_n \in F$ and all accumulators $acc \in A : acc = 0$. If $q_n \notin F$ or $acc \neq 0$, then w is rejected. If M_1 is a Type-1 calc-FSM, then $L(M_1)$ is the language of words over Σ which are accepted by M_1.

If the meaning is clear from context and our machine has one single accumulator, we may briefly write "store", "-1", "=0", and $\neq 0$ for the commands and comparisons.

Definition 8. *A Type-2 calc-FSM is a deterministic Type-1 calc-FSM, if for every $(q, r, s) \in Q \times \Sigma \times 2^{(A \times \mathcal{S})}$ and for every $(q, c) \in Q \times \mathcal{C}$ is at most one possible transition, i.e. $|\delta(q, r, s)| \leq 1$, $|\delta(q, c)| \leq 1$.*

The difference between determinism and non-determinism in calc-FSMs is analogous to regular FSMs. In fact, a Type-1 calc-FSM with 0 accumulators is a nondeterministic FSM, and a Type-2 calc-FSM with 0 accumulators is a deterministic FSM.

While Type-1 and -2 calc-FSMs seem to be a natural approach to parse calc-regular languages, they are actually a bit too powerful, which is why constrain Type-1 calc-FSMs in a different way.

Definition 9. *A Type-3 calc-FSM $M_4 = (Q, \Sigma, \delta, q_0, F, A)$ is a FSM with $k \geq 0$ accumulators. The transition function is a function $\delta : Q \times (\Sigma^* \times A) \to Q$. Therefore, a Type-3 calc-FSM can have calc-regular expressions on the edges, but no statements for changing an accumulator. Furthermore, they are restricted to have at most one transition from and to a state:*

$$\forall q_i \in Q : (|\delta(q, w) \to q'| \leq 1 : q = q_i)$$

$$\wedge (|\delta(q, w) \to q'| \leq 1 : q' = q_i)$$

All transitions concatenated to a c-RE define a language, which is the exact language the Type-3 calc-FSM accepts.

In the remainder of the current section, we will study the relationship between our different types of calc-FSMs and their relationship to calc-regular languages.

B. A gap between Type-1 and Type-2

Theorem 5. *For every Type-2 calc-FSM exists a Type-1 calc-FSM, which accepts the same language.*

Proof. All Type-2 calc-FSM can be seen as Type-1 calc-FSM, because determinism is a subset of non-determinism. □

In contrast to regular FSMs, nondeterministic calc-FSMs are more powerful than deterministic ones, which make Type-1 calc-FSMs to a superset of Type-2 calc-FSMs.

Theorem 6. *There are languages a Type-1 calc-FSM accepts, which no Type-2 calc-FSM can accept.*

Proof. In Figure 2 you can see a Type-1 calc-FSM which accepts the language $a^i b^j c^k$ with ($i = j$ or $j = k$). Basically, the Type-1 FSM consists of two different FSMs, one for the language $a^i b^i c^k$ and the other one for the language $a^i b^k c^k$. The first operation of the Type-1 calc-FSM is a nondeterministic choice between the deterministic machines.

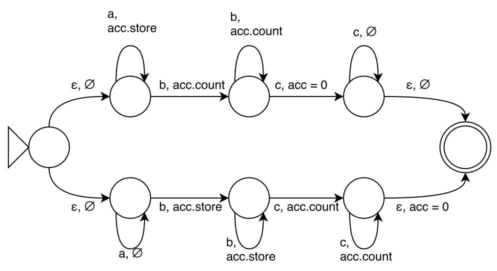


Fig. 2. A Type-1 calc-FSM for the language $\{a^i b^j c^k | (i = j) \vee (j = k)\}$.

No Type-2 calc-FSM can accept this language. We defer the proof to Theorem 12. Thus, $\{a^i b^j c^k | (i = j) \vee (j = k)\}$ is an example for a language which a Type-1 calc-FSM can accept, but no calc-2 FSM. □

Theorem 6 appears to be a heavy drawback for us. We are searching for languages, which can be parsed *efficiently* on *realistic machines*. For ordinary regular languages, nondeterministic FSMs are a good model, because they are not more expressive than deterministic FSMs. For the parsing of calc-regular languages, Type-1 calc-FSMs, the immediate counterpart to nondeterministic FSMs are too strong, and we have to restrict ourselves to Type-2 calc-FSMs, at best.

C. Type-3 calc-FSMs are equivalent to calc-regular languages

Theorem 7. *For every calc-regular expression r exists a Type-3 calc-FSM M with $L(r) = L(M)$.*

Proof. A Type-3 calc-FSM M has only transitions $\delta(q, w) \to q'$ where w is a calc-regular expression.

So we design M with two states q_0 and q_1, q_0 as initial state and q_1 as final state. M has exactly one transition $\delta(q_0, r) \to q_1$ and the alphabet Σ of M is equal to Σ of r.

The accepting language for a Type-3 calc-FSM is defined by the c-RE of the concatenated transitions. M has only one transition with r, so the accepted language $L = L(M) = L(r)$. □

Theorem 8. *For every Type-3 calc-FSM M exists a calc-regular expression r with $L(M) = L(r)$.*

Proof. A Type-3 calc-FSM has only transitions in form of $\forall \delta(q, r) \to q' \in \delta$ where r is a c-RE. We can concatenate all r_i to the calc-regular expression r of the Type-3 calc-FSM with the algorithm of Floyd and Warshall.
Therefore, we enumerate all states of the Type-3 calc-FSM with $1, ..., n$. The algorithm builds a calc-regular expression $r_{i,j}^{(0)}, r_{i,j}^{(1)}, ..., r_{i,j}^{(n)}$ for all $i, j \in \{1, ..., n\}$ step by step.
The algorithm corresponds to the one for transforming a deterministic FSM into a RE [7]. We only have to change the regular expressions to calc-regular expressions.

The result is $r = r_{1,j_1}^{(n)} + ... + r_{1,j_k}^{(n)}$ for q_1 as initial state and $k = 1, ..., n$. Hence, $L(r) = L(M)$, because it is analogical to regular languages.

□

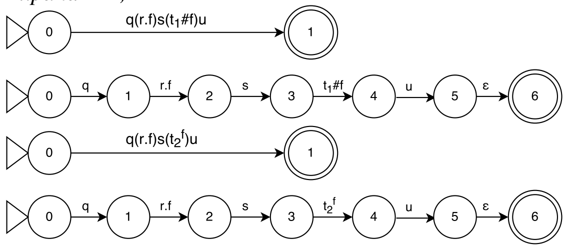
D. Transform Type-3 calc-FSMs to Type-2 calc-FSMs

Algorithm 9. *Given a Type-3 calc-FSM M we can transform it into a Type-2 calc-FSM M'. At first $M' = M$ and is a Type-3 calc-FSM, it will be processed to a Type-2 calc-FSM.*

M' has only transitions $\delta(q, a) \to q'$ where a is a c-RE. Assume every c-RE is in form of $r_1 = q(r.f)s(t_1\#f)u$ or $r_2 = q(r.f)s(t_2^f)u$, like we defined in Definition 6 with $q, s, u \in \{\Sigma^*|\emptyset|r_1|r_2\}$, $r, t \in \{\Sigma^*|r_1|r_2\}$ and $f \in A_M$.

Step 1: Type-3 to Intermediate Machine

- *Expand M',*

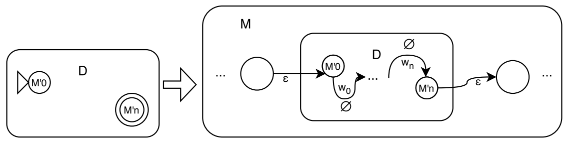


- *repeat until $\forall\delta(q, a, \emptyset) \to q' \in \delta_{M'}$: a is a regular expression or a ends with an accumulator statement.*
- *Since all subexpressions q, r, s, t_2 and u are prefix-free, the behaviour of our Intermediate Machine is well defined (reading subexpressions and moving from node to node).*

Step 2: Intermediate Machine to Type-2+ε

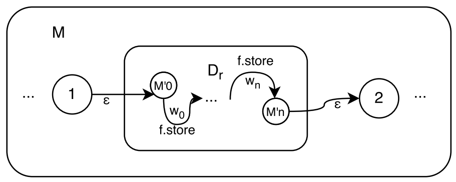
For q, r, s, t_2 and u

- *either the subexpression is prefix-free regular, then we can generate the matching deterministic FSM D , else the subexpression is (only) calc-regular, then we apply the algorithm recursively (see Step 3).*
- *$\forall\delta_D(q, w) \to q' : \delta_{M'}(q, w, \emptyset) \to q'$*



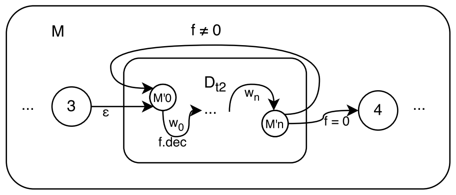
For $r.f$

- *we have to attach $f.store$ to every edge in the deterministic FSM of r*



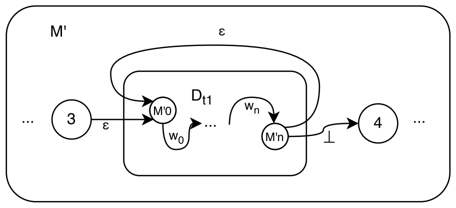
For t_2^f

- *we have to attach $f.dec$ to all transitions with $\delta_{D_{t_2}}(q_{M'0}, w) \to q'$ and $f = 0$ for every outgoing transition and a backwards transition with $f \neq 0$ from all final states of D_{t_2} to the initial state of D_{t_2}*

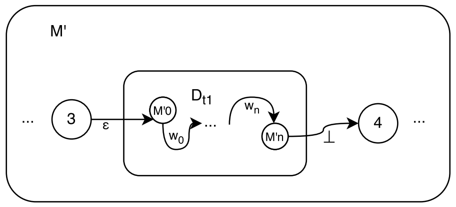


For $t_1\#f$

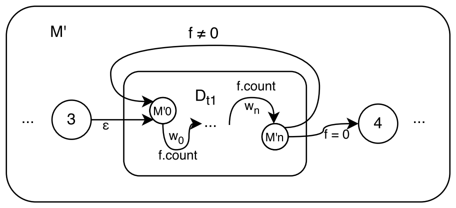
- *If t_1 is prefix-free, then enhance the outgoing edge of D_{t_1} by reading the pseudo-symbol $\perp$ and an ε-transition backwards from the final states to the initial states,*



- *otherwise t_1 is prefix-free but regular, so we get a deterministic FSM for $t_1^*\perp$.*



- *In either case we have a machine accepting $t_1^*\perp$.*
  Now we have to attach $f.count$ to every edge in D_{t_1} and $f = 0$ to all outgoing edges and $f \neq 0$ to all backwards transitions $\delta_{D_{t_2}}(q_{M'n}, w) \to qM'i$, $i \in 0, \ldots, n$



Step 3:

If q, r, s, t_2 or u is calc-regular and not also regular, we have to consider ($b \in \{q, r, s, t_1, t_2, u\}$):

- *$b.f$: for every new transition $\delta(q, c, s) \to q'$: $s = \{s, f.store\}$*
- *b^f : non-hereditary, which means subexpressions of b do not have to consider the statements from "parent"-expressions, and is ignored for expressions in u*
- *$b\#f$: for every new transition $\delta(q, c, s) \to q'$: $s = \{s, f.count\}$*

Step 4: Eliminate ε-transitions

For reasons of clarity we used ε-transitions in Step 2. We can eliminate all ε-transitions, when $!\exists\delta(q, \varepsilon, \emptyset) \to q' \wedge (\nexists\delta(q, c, \emptyset) \to q' : c \neq \varepsilon)$ by merging q and q' to one state. This is the only case in which ε-transitions can appear in this algorithm. By eliminating them the Type-2+ε calc-FSM become a Type-2 calc-FSM

It would be interesting to prove or disprove the following.

Conjecture. *There exists a Type-2 calc-FSM M_2, such that the language $L(M_2)$ accepted by M_2 cannot be accepted by any Type-3 calc-FSM.*

VI. Calc-Regular Languages and The Chomsky Hierarchy

As we elaborated in the introduction, the state-of-the-art in defining and parsing languages is heavily influenced by the famous Chomsky hierarchy. The most general class of languages in the Chomsky hierarchy are the recursively enumerable languages, followed by context-sensitive, context-free and regular languages. The following table lists some of the main results for such languages, which respect to two problems:

1) The word problem: Given a word $w \in \Sigma^*$, is w in the language?
2) The problem of computational equivalence: Given two machines (i.e., parsers) accepting a language from some class of languages. Are both machines accepting the same language?

Following the reasoning from [15], the ability to solve these two problems is important for security applications.

The following table summarizes the main results for the languages from the Chomsky hierarchy, with an additional distinction between general context-free languages, and the subset of deterministic context-fee languages.

	"word problem" is w in L?	"computational equivalence"
rec. enumerable	**undecidable**	**undecidable**
context-sensitive	**exponential**	**undecidable**
context-free	cubic	**undecidable**
det. context-free	linear	decidable
regular	linear $O(1)$ storage	decidable

So how does our new class of languages relate to the classes from the Chomsky hierarchy? By definition, a regular expression is also a calc-regular expression (but not vice versa). [5] Thus, calc-regular languages are a superset of regular languages.

Furthermore, from Theorem 2 we know a calc-regular language **N**, the language of well-formed netstrings, which is not regular – and not even context-free. Thus, calc-regular languages are a proper superset of regular languages. Are they also a superset of context-free languages? As it turns out, the answer to this question is negative: There exists a context-free language and even a deterministic context-free language, which is not calc-regular.

The following lemma is some form of a calc-regular counterpart to the famous pumping lemmas for regular and context-free languages (Theorem 1(c,f)). We will use the lemma to prove that certain languages cannot be parsed by a Type-2 calc-FSM and thus are not calc-regular.

[5]Recall the "trick" of assuming a final $\perp$-symbol at the end of every word.

Lemma 10. *Consider a Type-2 calc-FSM M_2 with α accumulators and a set Q of (FSM-) states. Consider inputs v_1w, v_2w, ... Assume that, after reading any of the v_i, all values stored in any of the α accumulators are in the range $\{0, \dots m-1\}$, for some fixed threshold m. I.e., each accumulator can hold any of at most m different values. If there are more than*

$$m^\alpha * |Q|$$

choices for the v_i, then some $v_i \neq v_j$ exist, such that if M_2 accepts v_iw, then M_2 also accepts v_jw.

Proof. In principle, if $\alpha \geq 1$, the state space of M_2 is infinite. But if we restrict the infinite number of choices for each accumulator to m, then there are m^α different α-tuples of values in the entire set of accumulators, and $m^\alpha * |Q|$ different $(\alpha+1)$-tuples of accumulator-values and state. Thus, there are only $m^\alpha * |Q|$ different states the machine can be in, after reading v_i. □

As the first attempt, we will consider the language **L**=a^nb^n, which is a well-known textbook example for a deterministic context-free language, which is not regular. As it turns out, **L** is calc-regular.

Theorem 11. *The language **L**=a^nb^n is a calc-regular language.*

Proof. Let cnt be the (simple) value-oracle which counts the length of its input. For the language $L = a^nb^n$, the c-RE is

$$a.\text{cnt} \quad b^{\text{cnt}},$$

since $\text{cnt}(a^i) = i$ and $b^{\text{cnt}(a^i)} = b^i$. □

Note that the c-RE

$$a.\text{cnt} \quad b\#\text{cnt}$$

also generates the language a^nb^n.

The second attempt for a (nondeterministic) context-free language, which is not regular, is the language $a^ib^jc^k$, with the constraint $(i = j$ or $j = k)$. As it turns out, this language is not calc-regular.

Theorem 12. *The language $a^ib^jc^k$ with $(i = j$ or $j = k)$ can not be parsed by a Type-2 calc-FSM. Thus, it is not a calc-regular language.*

Proof. Consider a calc-FSM with α accumulators and $|Q|$ states. It must accept inputs of the form $a^nb^nc^*$. Lemma 10 implies that there is at least one accumulator which holds a number $m-1$ (or larger) with

$$m^\alpha \geq \frac{n}{|Q|}.$$

But then, this calc-FSM cannot accept a word $a^nb^kc^k$ for small k and large enough n. Namely, when $m-1 > 2k$, that accumulator will still store a nonzero number after reading b^kc^k, as with each input symbol, the accumulator can only be deceased by one.

Thus, after reading $a^nb^kc^k$, there is at least one accumulator with a nonzero storage. By the definition of calc-FSMs, the machine rejects. □

From Theorem 12, we can conclude that the calc-regular languages are not a superset of context-free languages in general. This is good news for us, because otherwise we would have to expect at least cubic run time (in the worst case) for parsing calc-regular languages, and the computational equivalence problem would be undecidable.

As it turns out, calc-regular languages are not even a superset of deterministic context-free languages. We define a language **M**$\subseteq$ { "0", "(", ")", "[", "]" }* of "mixed brackets". The grammar[6] for **M** uses a start symbol S and can be written by three productions $S \to 0$, $S \to (S)$, $S \to [S]$. In short, **M** consists of a (possibly empty) sequence of opening brackets "'(" or '[", followed by "0", followed by a sequence of closing brackets, which must match exactly the corresponding opening brackets. **M** is deterministic context-free. A deterministic pushdown automata will push all the opening brackets onto the stack, then read the "0", and, when reading the closing brackets, check if they match the opening brackets on the stack. Even though this language is quite simple, it is not calc-regular:

Theorem 13. *The language M is not calc-regular.*

Proof. Consider an input w"0"v, with w being a sequence of n opening brackets from { "[" "(" }, v being a sequence of n closing brackets. For any such w, there is exactly one v, such that $w\text{"}0''v \in M$. Thus, a calc-FSM for M must, after reading w, be able to distinguish all the 2^n different choices for v.

For large enough n, there is at least one input v for this machine, such that there is at least one accumulator holding a number greater than n. This can be seen by applying Lemma 10, After reading the n characters from v, the accumulator will still not be zero. By the definition of calc-FSMs, the input must be rejected if there is a nonzero value in an accumulator. Thus, this machine will reject $v\text{"}0''w$, regardless of w holding the closing brackets matching the opening brackets in v or not. □

To summarize the results from the current section: calc-regular languages are orthogonal to context-free languages. There exist languages, such as **N** (well-formed netstrings) from Theorem 2, which are calc-regular and not context-free. There are languages, such as **M** (mixed brackets) from Theorem 13, which are deterministic context-free and not calc-regular. And, there are languages such as **L**=$a^n b^n$ from Theorem 11, which are both deterministic context-free and calc-regular, but not regular. See Figure 3.

VII. Properties of Calc-Regular Languages

How expressive are calc-regular languages? How similar are they to regular languages, regarding closure and decidability properties? In this section, we will scrutinize calc-regular languages and present some first answers.

Theorem 14. *Calc-regular languages are closed under concatenation.*

[6]Once again: We assume the reader to be familiar with core notions from Formal Language Theory, as, e.g., in [7].

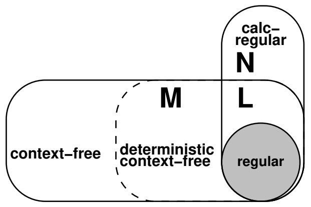


Fig. 3. Orthogonality of calc-regular and context-free languages

Proof. Given the calc-regular languages L_1, L_2 and their calc-regular expressions r_1, r_2 we can construct a Type-3 calc-FSM M, which accepts $L_1 L_2$.

M has three states $Q = \{q_0, q_1, q_2\}$, q_0 is the initial and q_2 the final state. We add the transitions $\delta(q_0, r_1, \emptyset) \to q_1$ and $\delta(q_1, r_2, \emptyset) \to q_2$ to M. Due to the definition of Type-3 calc-FSMs (Definition 9), the accepted language is the language of the concatenated transitions, so $L(M) = L(r_1 r_2)$.

Hence calc-regular languages are closed under concatenation. □

Theorem 15. *Calc-regular languages are not closed under union.*

Proof. Given the languages $L_1 = \{a^i b^j c^* | i = j\}$ and $L_2 = \{a^* b^j c^k | j = k\}$.

Assume we add c^* to the calc-regular language $a^n b^n$ and get the language $a^n b^n c^*$. This language is still calc-regular, because calc-regular languages are closed under concatenation, as shown in Theorem 14 . Thus, L_1 and L_2 are calc-regular.

The union of the two languages would be $L_\cup = \{a^i b^j c^k | i = j \, or \, j = k\}$ and we already showed in Theorem 12 that this language can't be parsed by a Type-2 calc-FSM, therefore it is not a calc-regular language.
Hence, calc-regular languages are not closed under union. □

Since calc-regular languages are not closed under union (Theorem 15), we conjecture the following.

Conjecture. *Calc-regular languages are not closed under Kleene star.*

The definition of the Kleene star L^* operation is based on a massive usage of the union operation:

$$L_0 = \{\varepsilon\}, L_1 = L, L_{i+1} = \{wz | w \in L_i \cap z \in L\} \text{ for i} < 0$$
$$L^* = \bigcup_{i \in N} L_i = \varepsilon \cup L_1 \cup L_2 \cup L_3 \cup ...$$

Thus, the above conjecture would seem obvious. However, we do not have a conclusive proof. Theorem 15 only proves that some calc-regular languages L_1 and L_2 exist, where the union $L_1 \cup L_2$ is not calc-regular. The theorem does not imply anything for the specific L_i from the definition of the Kleene-star operation.

VIII. A Meta-Language and Practical Examples

The current paper's focus is on *understanding* the issues related to specifying length-prefix notation (or count-prefix).

Beyond the theoretical results above, a long-term goal is a non-ambiguous meta-language to specify calc-regular languages and a tool to automatically generate parsers from it. In the current section, we will sketch how the meta-language could look like, and provide some examples for practical languages specified using the meta-language.

There are plenty of `regex` meta-languages for regular expressions, such as, e.g., the `regex`-support in Perl and the input syntax for the `grep` tool included in most Unix-based systems.[7] [24]. We did consider to pick such a `regex` meta-language, and to add some notation for the subexpressions

$$\begin{aligned} r_1 &= q(r.f)s(t\#f)u \quad \text{and} \\ r_2 &= q(r.f)s(t^f)u \end{aligned}$$

from Definition 6. We decided against this approach, however. Instead, we will base our meta-language on Extended BNF [19]. Apart form what we consider a cleaner syntax, this eases future extensions of our meta-language for some form of "calc-context-free" languages.

A. A Meta-Language for Context-Free Languages.

The core point for a meta-language specification is to distinguish meta-symbols, such as, e.g., brackets, from terminal characters. We use the following convention:

- A terminal can be written by the percent-symbol ("`%`"), followed by the numeric value (in hexadecimal notation).
- If the nonterminal is a printable character, it can also be written in single or double quotation marks. E.g., if we assume ASCII representation of characters, the printable character 'a' can also be written as "a" and as `%61`.
- All other symbols are meta-symbols.

Typically, the input alphabet is $\Sigma = \{$`%0`, …, `%FF`$\}$.

For the concatenation of subexpressions, we write a comma (','). E.g., `%61`, 'b', 'c' represents the string "abc". As a shortcut notation for the concatenation of terminals, we allow multi-character strings such as, e.g., 'abc' for 'a', 'b', 'c'.

For the choice between subexpressions, we use the "|"-character. E.g. 'a' | 'bc' represents either the single character 'a', or the concatenation of two characters 'bc'. Ranges, such as 'a' | 'b' | 'c' | 'd' can be written as 'a' - 'd'.

We use superscript-notation for the concatenation of the same subexpression several times, and use the Kleene-star "*" and the "$^+$"-shorthand as usual.

For our meta-language, we use EBNF productions, as in

```
byte   =   %0-%FF;
```

for arbitrary bytes, and in

```
nonzero-digit   =   "1" - "9";
digit           =   "0" | nonzero-digit;
number          =   "0" | (nonzero-digit, digit*);
pf-number       =   number, ":" ;
```

for decimal number without leading zeroes, and for such numbers with a terminating a colon for prefix-freeness (as in netstrings).

[7]Often with extensions, which allow to also specify some non-regular languages in the meta-language.

We require every nonterminal X to be defined exactly once, i.e., there is exactly one rule $X = Y$ with X being a single nonternimal and the expression Y the *definition* of X. The semicolon (";") terminates a production.

B. From Context-Free to Regular to Calc-Regular.

So far, our meta-language would allow us to specify arbitrary context-free languages – that is, what EBNF has been designed for. To restrict the languages we can specify to regular ones, rather than to general context-free ones, we must restrict the usage of nonterminals right of the "=" symbol:

Constraint 10. *A nonterminal may only be used on the right-hand-side of a production, if it has been defined in a previous production.*

This constraint prevents any recursive usage of nonterminals, either directly, as in "S = '(' S ')';" or indirectly as in "S = '(' T ')'; T = '[' S ']';".

It is straightforward to turn a language specification observing the above constraint into a specification of the same language without nonterminals. Namely, if the nonterminal X occurs in a definition, and the production for X is $X = Y$, i.e., Y is the definition of X, then we can textually replace the occurrence of X in definitions by (Y). After finitely many such replacements, we get a single expression without nonterminals, which then is a regular expression. E.g., if we apply this approach to "pf-number", we get

```
pf-number  =  ("0" | (("1"-"9"), ("0" | ("1"-"9")))), ":";
```

which defines "pf-number" without any nonterminals in the definition.

After first restricting the meta-language, we now extend it. Assuming we know which function is represented by f, we can just write

```
r-one   :=   q, (r.f), s, (t#f), u ;
r-two   :=   q, (r.f), s, (t^f), u;
```

almost as in Definition 6.

Note that Definition 6 does not allow the union operator or the Kleene-star operator for calc-regular expressions – except when the expression is regular. As our meta-language is supposed to model calc-regular expressions, we need to distinguish nonterminals, which are part of a strictly regular subexpression, from nonterminals, which are part of a proper calc-regular subexpression, and we need to constrain the usage of the second kind of nonterminals.

Definition 11. *A nonterminal is restricted, if it is defined by a restricted production. A production is restricted,*

- *if it uses the "(r.f), …, (t#f)"-syntax,*
- *if it uses the "(r.f), …, (t^f)"-syntax,*
- *if there is at least one restricted nonterminal in the definition of the production.*
- *No other productions are restricted.*

To mark a production as restricted, we write ":=", as in the productions for "r-one" and "r-two" above, where we write "=" else, as in the production for "number".

Constraint 12. *A restricted production must not use the union- or Kleene-star operators.*

Note that the notation r^+ is the shorthand for rr^* and thus also represents a Kleene-star operation, which is prohibited for restricted productions.

C. Specifications for Practical Calc-Regular Languages

Though the specification for our meta-language is quite sketchy, it suffices for the examples we give below. We will not repeat the previous definitions for "number" and "byte".

Recall **netstrings** without nesting, as in "5:Hello," and heartbeat-like **nested netstrings**, as in "8:3:abc,XY," with a 3-byte challenge "abc" and two padding bytes "XY". We can specify these calc-regular languages as follows:

```
netstring    :=  pf-number.decimal,
                 byte#decimal, ",";
n-netstring  :=  pf-number.decimal,
                 (netstring, byte*)#decimal, ",";
```

Note that the "(netstring, byte*)#decimal"-part of the specification for nested netstrings formally rules out the acceptance of a netstring whenever the inner netstring exceeds the outer one, as in the attack package "5:9999:;". That does not mean bugs, such as Heartbleed, will magically "go away", when we use our meta-language for specifications. But, if we use an automated tool to turn the specification into a parser, we have to take care *only once*, when implementing the tool, to make sure the tool always puts the proper check at the proper places.

A **PNG** (portable network-graphic) **chunk** [22] consists of a 4-byte field holding the number of bytes in the data, followed by a four-byte the chunk-type, followed by the data, followed by a 4-byte CRC checksum. Critical chunk types are IHDR for the first chunk, PLTE for the chunk storing the list of colors, IDAT for image chunks, and IEND for the last chunk. PNG allows other, "ancillary" chunk types, which we ignore for ease of presentation. Then the calc-regular grammar for PNG chunks would be the following:

```
chnk-type  =   "IHDR" | "PLTE" | "IDAT" | "IEND";
chksum     =   byte^4;
png-chnk   :=  (byte^4).big-endian, chnk-type,
               byte#big-endian, chksum;
```

Note that the CRC-check is not part of our language definition. A PNG-chunk can be rejected due to a CRC-failure, even if it is syntactically correct.

Also note that (byte^4) is of constant length and thus trivially prefix-free.

To pick yet another example, consider the **MessagePack strings and arrays** [6]. Message Pack supports different string types. E.g. strings of length up to $2^{16}-1$ bytes (str16) are represented in MessagePack by a one-byte type-identifier, followed by a two-byte length field, followed by the data of the specified length. Strings of length up to 2^{32} bytes (str32) are similar, except, of course for the type-identifier, and the usage of a four-byte field for the length. MessagePack arrays use count-prefix notation, i.e., store the number of elements, not the number of bytes. Below, we will use our meta-language to specify a string (str16) of less than 2^{16} bytes and an array (arr-of-str16), which can hold up to $2^{16}-1$ such strings.

```
str16         :=  %D9, (byte^2).big-endian,
                  byte#big-endian;
str32         :=  %DB, (byte^4).big-endian,
                  byte#big-endian;
arr-of-str16  :=  %DC, (byte^2).big-endian,
                  str16^big-endian;
```

D. A Possible Extension of the Meta-Language

Alas, we are far from modeling the full MessagePack language. Consider an array, which can hold strings, and each string could be either of type str16, or of type str32:

```
a-str       :=  (str16 | str32);
arr-of-str  :=  %DC, (byte^2).big-endian,
                a-str^big-endian;
```

At a first look, this may appear OK. But actually, this does not match a calc-regular expression, and we cannot write this in our meta-language, because it violates Constraint 12.

So why not remove Constraint 12 from the meta-language? Of course, the set of languages we could specify with the tweaked meta-language would no longer be the calc-regular languages as defined in the current paper. It would be calc-regular languages with additional support for union and Kleene-star operations. Thus, it would include languages, which we do not know how to parse efficiently (cf. Theorem 6 and the discussion in Section V-B). We do not consider this approach advisable.

On the other hand, the MessgePack example shows that different syntactic constructs can be uniquely identified by different type-prefixes (the byte constants `%D9` and `%DB` and `%DC` in the example). The union of such calc-regular languages *should* be easy to parse, even though the union of regular languages in general is, most likely, not easy to parse. Thus, tweaking our definition for calc-regular languages, and then relaxing – but not completely removing – Constraint 12, may be an interesting research direction to explore, in the future.

E. Deterministic Calc-Context-Free Languages

There is another issue with our meta-language. Most data serialization languages support nesting – and often unlimited nesting of data structures. E.g., the item stored in a MessagePack array could also be another MessagePack array. We cannot model this in our meta-language, due to Constraint 10. Very much like the distinction between regular and context-free languages, calc-regular languages almost always fail when it comes to unlimited nesting, cf. Theorem 13.[8]

Eventually, pushing this research topic forward to define and analyze "calc-context-free languages", and especially "deterministic calc-context-free languages" may be another interesting research direction to explore. In fact, we consider the current paper one step into that direction – which is one reason, why our meta-language has been based on EBNF, rather than on meta-languages for regular languages.

[8]With some very specific exceptions. E.g. set $a=$"(" and $b=$")" and recall the language $\mathbf{L}=a^nb^n$ from Theorem 11.

IX. Conclusion and Future Work

A common design pattern for data serialization languages is the usage of length-prefix notation. I.e., when sending a data blob B of N bytes, the sender will first send the number N and then the data B. Thus, the receiver will know how large B is, before it starts reading B. Even though this design pattern is quite common, the issues of parsing such languages are not well-understood, and implementations are frequently plagued with bugs and vulnerabilities (cf. Heartbleed).

In the current paper, we introduced and studied calc-regular languages, a formal model for languages with length-prefix notation. Calc-regular languages can be defined via calc-regular expressions. As it turns out, calc-regular languages are context-sensitive. Nevertheless – and perhaps a bit surprising – parsing calc-regular languages is almost as easy as parsing ordinary regular languages.

Though we did not write a parser generator for calc-regular languages, we did sketch a meta-language which may serve as the input for such a parser generator. Furthermore, we gave some example specifications for practical languages, such as netstrings and PNG chunks, using the meta-language.

We consider this work a starting point for future research. There are plenty of open problems and challenges, such as

- prove or disprove the conjectures in our paper,
- study more properties of calc-regular languages,
- tweak the definition of calc-regular languages, to support some kind of union operation, as discussed in Subsection VIII-D,
- push the approach to a higher level by introducing unlimited nesting, i.e., by formalize and studying some new class of "deterministic calc-context-free" languages, as discussed in Subsection VIII-E,
- write a parser generator for calc-regular languages, or tweaked calc-regular languages, or deterministic calc-context-free languages,
- ...

We argue that advances in these directions will eventually provide tools to eliminate bugs and avoid security issues, We hope this work will be inspiring for other researchers, to perform similar work or to give us some feedback, regarding the directions for future research.

Acknowledgement

The third author thanks Christian Forler for the initial suggestion to consider this research topic at all. A part of this research has been supported by the research project SIMPLE SCRY (Strong IMPLEmentations for Symmetric CRYptography) from Cisco.

References

[1] J. Backus et al: Report on the Algorithmic Language ALGOL 60. Communications of the ACM 3:5, 229–314 (1960). http://dl.acm.org/citation.cfm?doid=367236.367262.

[2] D. Bernstein (1999): Netstrings, http://cr.yp.to/proto/netstrings.txt.

[3] S. Bratus, M. Patterson, A. Shubina: The Bugs we have to Kill. USENIX;login: vol. 40, no. 4. http://langsec.org/papers/the-bugs-we-have-to-kill.pdf.

[4] N. Chomsky: Three models for the description of language". IRE Transactions on Information Theory (2): 113124. https://chomsky.info/wp-content/uploads/195609-.pdf

[5] Github: Hammer. https://github.com/UpstandingHackers/hammer.

[6] Github: MessagePack https://github.com/msgpack/msgpack/blob/master/spec.md.

[7] J. Hopcroft, R. Motwani, J. Ullman: Introduction to Automata Theory, Languages, and Computation (2nd ed.).

[8] B. Kaliski: A Layman's Guide to a Subset of ASN.1, BER, and DER. An RSA Laboratories Technical Note. Revised November 1, 1993. http://luca.ntop.org/Teaching/Appunti/asn1.html.

[9] D. Kaminski, M. Patterson, L. Sassaman: PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure. Black Hat USA, 2009. http://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[10] B. Kehoe: Zen and the Art of the Internet. A Beginner's Guide to the Internet, First Edition, January 1992. http://www.cs.indiana.edu/docproject/zen/zen-1.0_toc.html.

[11] F. Momot, S. Bratus, S. Hallberg, M. Patterson: The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. IEEE SecDev 2016, Nov. 2016, Boston. http://langsec.org/papers/langsec-cwes-secdev2016.pdf.

[12] M. Rabin, D. Scott: Finite automata and their decision problems. IBM J. Research and Development 3:2, 115–125 (1959).

[13] A. Oettinger: Automatic syntactic analysis and the pushdown store. Information and Control 8:6, 607–639 (1961).

[14] L. Sassaman, M. Patterson, S. Bratus, M. Locasto, A. Shubina: Security Applications of Formal Language Theory. IEEE Systems Journal, Volume 7, Issue 3, Sept. 2013. http://langsec.org/papers/langsec-tr.pdf.

[15] L. Sassaman, M. Patterson, S. Bratus, A. Shubina: The Halting Problems of Network Stack Insecurity. USENIX;login: vol. 36. no. 6, 2011. https://www.usenix.org/publications/login/december-2011-volume-36-number-6.

[16] M. Schutzenberger: On context-free languages and pushdown automata. Proc. Symposia on Applied Math. 12, American Mathematical Society (1963).

[17] W. Underwood, S. Laib: Attribute Grammars and Parsers for Chunk-Based Binary File Formats. Georgia Tech Research Institute ICL/ITDSD Technical Report 12-01, 2012.

[18] Wikipedia: Comparison of Data Serilization Formats (Jan. 2017) https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats.

[19] Wikipedia: Extended Backus-Naur form (Jan. 2017) https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form.

[20] Wikipedia: Heartbleed (Jan. 2017) https://en.wikipedia.org/wiki/Heartbleed.

[21] Wikipedia: Hollerith Constant (Jan. 2017) https://en.wikipedia.org/wiki/Hollerith_constant.

[22] Wikipedia: Portable Network Graphics (Jan. 2017) https://en.wikipedia.org/wiki/Portable_Network_Graphics.

[23] Wikipedia: Protocol Buffers (Jan. 2017) https://en.wikipedia.org/wiki/Protocol_Buffers.

[24] Wikipedia: Regular expression (Jan. 2017) https://en.wikipedia.org/wiki/Regular_expression.

[25] N. Wirth: Compiler Construction. Zürich, Feb. 2014. (Slightly revised version of the book published by Adison-Wesley in 1996.) http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf.

Appendix

Early versions of FORTRAN did support "Hollerith constants" for strings [21]. A Hollerith constant is a netstring using "H" as the non-digit separator symbol (in honor of Herman Hollerith), instead of the colon, and without a comma at the end. Examples: `11HHELLO WORLD` and `14HHELLO "WORLD".` Note that the dot at the end of the second example is still part of the string.

In most programming languages today, including modern FORTAN, string constants are enclosed in single or double quotes. This seems to match human perception much better. The second quotation mark is the "end"-postfix for the string, as in `"HELLO WORLD"`. A quotation mark within the string constant needs "escaping". This is often done by doubling the (single- or double-) quotes, as in Pascal: `"HELLO`

`""WORLD"".`" An alternative is to prepend a backslash, as in Python: `"HELLO \"WORLD\"."`

While length-prefix notation is essentially gone for good from programming languages, it seems quite typical for data serialization languages, even though a few data serialization languages also support end-postfix notation. Note that we consider a representation, which encloses data in brackets or quotes, a form of end-postfix notation.

For containers (arrays, lists, ...), some data serialization languages also use count-prefix notation, i.e., storing the number of elements rather than the number of bytes. If the sizes of the elements are not known in advance, the number of the elements does not allow to compute the number of bytes to store the container.

Following the links from [18], we compiled a list of data serialization languages and which of these use either length- or count-prefix or end-postfix notation. We write LPre for length-prefix, CPre for Count-prefix and EPost for end-postfix notation.

Language	Strings	Containers
Apache Avro	LPre	CPre
ASN.1	LPre	LPre
Bencode	LPre	EPost
Binn	LPre	CPre
BSON	LPre	-
Cap'n Proto	LPre	LPre
CBOR	EPost	EPost
Colfer	LPre	EPost
D-Bus	LPre +EPost	LPre
Fast Infoset	LPre	LPre
google protobuf	LPre	LPre
GVariant	EPost	-
JSON	EPost	EPost
KMIP	LPre	-
Message Pack	LPre	CPre
Netstring	LPre	-
OGDL	end-postfix	-
OpenDDL	EPost	-
smile	LPre	EPost
s-expression	EPost	EPost
thrift	LPre	-
XDR	LPre	CPre