THE LANGUAGE CLASS CALC-LL(1) AND ITS APPLICATIONS IN PARSING VARIABLY NESTED LENGTH-PREFIX FORMATS

Jannis Leuther
Study course Medieninformatik Bsc.
Winter semester 2018/2019
Bauhaus Universität Weimar
`jannis.leuther@uni-weimar.de`

March 30, 2019

ABSTRACT

In the following report, the achievements of the research project "Language based security - Developing Calc-LL(1) Languages" will be presented. The project carried out at Bauhaus Universität Weimar supervised by Professor Stefan Lucks lasted one semester from October 2018 to March 2019 and was a first step on the road to establishing the Calc-LL(1) language class and its potential applications.

This project was based on previous research about the underlying problem on formal definitions of data serialization formats using length prefix notation as presented in the paper *Taming the Length Field in Binary Data* [1] by Norina Marie Grosch, Joshua König and Stefan Lucks. It is recommended to read through this paper before taking a look at the following report, as the topic of Calc-LL(1) languages relies heavily on the former research done by Grosch et al.

With the creation of Calc-LL(1) languages, an extended definition of Calc-Regular languages[1] shall be established, allowing the usage of length-prefix formats with variable nesting. Following this definition, a model for parsing these languages in a fashion similar to LL(1) parsers for deterministic context-free languages can be constructed employing additional elements specific to Calc-LL(1).

This project, however, limited itself to creating a reference parser implementation for *netstrings* in the scripting language *python*, producing an extended definition of parse tables for the usage in Calc-LL(1) parsing, as well as providing theoretical and practical foundations for further development on Calc-LL(1) parser generation.

1 Introduction & Motivation

Calc-Regular languages [1] are a formal language class introduced to represent the concept of length prefix formats mainly used in data serialization. Calc-Regular languages can represent any language using length prefix notation and with the introduction of Calc-Finite State Machines, they are able to confirm if binary strings employing such a notation are formed correctly. The goal of establishing Calc-Regular languages was to establish a theoretical and formal foundation for data serialization formats using the length-prefix notation as to eliminate potential errors and guarantee secure applications. As of now, there are still bugs, security issues and unwanted behaviour occurring in various usages of formats using length-prefix notation. The most famous issue occurred in 2014 with the 'Heartbleed'-bug, where a security flaw in the OpenSSL library lead to a compromise of *potentially* every bit of data serialized using the OpenSSL library.

The 'Heartbleed' exploit could be described as follows: An attacker would send 'Heartbeat' packages to another entity to check if this entity was still alive. 'Heartbeat' sent a package to the desired entity and received the same package back as an answer. As 'Heartbeat' was using the length-prefix notation, the attacker could issue heartbeat packages with a high length-prefix, e.g. 1000, signaling that the contents of the package consists of 1000 bytes. In reality, though, the contents were empty and the answer from the other entity then compromised 1000 bytes of its internal memory as a message with 1000 bytes was sent back. Therefore it was also not possible to check afterwards **if** or **which** information was compromised. Avoiding extreme security flaws like these but also eliminating other less intense bugs in length-prefix formats is the main motivation of creating Calc-Regular and Calc-LL(1) languages.

Calc-LL(1) languages are the next step in this development, extending Calc-Regular languages so that formats using *variable nesting* are supported as well. As a matter of fact, many current data serialization languages and formats are parsed using basic LL(1) parsing techniques and should therefore be classified as context-free. On a closer look, though, one can see irregularities that point to these languages not actually being context-free as of the Chomsky-class's strict definition. Rather they could be described as a similar language class but with additional restrictions and extensions. This new language class is what we are calling Calc-LL(1). Noam Chomsky's formal language hierarchy's classifications as well as LL(1) languages were mainly established to also ensure that humans can read and grasp these languages somewhat easily. For data serialization languages, human readability is not required and arguably not even wanted as it can complicate languages with unnecessary additions. Calc-LL(1) could therefore be described as a language class including languages that do not have to be readable by humans but only by machines. One could also name those languages 'Calc-context-free' to be more consistent in regard to the Chomsky hierarchy and the former Calc-Regular definition, but as this new language's ultimate purpose is to create parsers that work similar to LL(1) parsers, the name Calc-LL(1) is more suitable in the author's opinion.

2 Defining Calc-LL(1) Languages

An informal definition of Calc-LL(1) languages can be given by taking into account Calc-Regular languages as well as LL(1) languages. While Calc-Regular languages are able to represent languages using a length prefix format without variable nesting, LL(1) languages guarantee an efficient and relatively simple parsing procedure. Calc-LL(1) aims to merge the characteristics of these two classes into one that is then able to parse length prefix notation formats even with variable nesting while employing the efficient, table-based parsing structure that is given in LL(1).

3 Conventional LL(1) parsing

LL(*k*) parsers are top-down parsers for subsets of context-free languages that can be represented using deterministic pushdown automatons. The parser parses from **L**eft to right and performs a **L**eftmost derivation of the input. When using *k* tokens of lookahead while parsing, an LL-parser is called a LL(*k*) parser. Therefore, a LL(**1**) parser uses a lookahead of only one token at each parsing step. All grammars of languages that can be parsed with LL(1) are a subset of deterministic context-free grammars and called LL(1) grammars respectively. For more in-depth knowledge on parsers and their theoretical foundations, we recommend the book "Parsing Techniques" by Grune and Jacobs [2].

Parse tables are an important and necessary component of LL(*k*) parsers as these are so called table-based parsers. A parse table M for a given LL(1) grammar $G = \{N, \Sigma, P, S\}$ is a table with all non-terminal symbols $\gamma \in N$ as columns and terminal symbols $\sigma \in \Sigma$ as rows (or vice versa). Each table entry $m_{\gamma,\sigma} \in M$ may either include nothing or at most one production rule $p \in P$. This table entry content in $m_{\gamma,\sigma}$, if present, tells the LL(1) parser that it has to apply rule p when its current non-terminal being popped from the internal parsing stack is γ and the lookahead token is equal to σ. As a result, there must only be one production rule at maximum in each table cell, for else the parser could not decide deterministically how to continue due to ambiguity. Parse tables for LL(1) grammars are created prior to the parsing procedure using so called *First Sets* and *Follow Sets*. This report will not go into detail on how to create and use these tools as there exists a lot of accessible information about this procedure on the internet or in literature; for example: [2].

4 Length-Prefix Notation Formats

Length-prefix notation depicts a schema that is employed in many data serialization or communication protocols and solves the problem of identifying distinct messages or packages in binary data streams. By declaring the size of the message's actual content at the beginning of each message, the parser can say exactly where this message ends and a new one then might start. The size of the content is represented as the length of the byte string that describes the content. Within data communication protocols, length-prefix notation is used in *Type-Length-Value* formats where a *type*-field is added to convey additional information about the subsequent message. The usage of length-prefixes or the *Type-Length-Value* format is not exclusive to data communication or serialization, as it can be used to encode any binary data efficiently. For example, the *Portable Networks Graphics*-format uses *Type-Length-Value* to encode image data.

Nesting in length-prefix formats describes the ability for single messages to act as a container containing messages from the same format. Calc-Regular languages can only describe these containers with a fixed nesting defined beforehand. As Calc-Finite State Machines [1] can not decide whether a message acts as a container or simply as a single string when looking solely at the length prefix, variable nesting can not be represented. This is where Calc-LL(1) languages come into play, as this language class allows for the description and parsing of formats using *variable nesting.* With variable nesting, containers can not only store simple strings but other containers as well in recursive fashion. As a matter of fact, most data-serialization and communication formats are able to represent variable nesting. Therefore, compared to Calc-Regular languages, Calc-LL(1) languages are closer to the applicable side of the underlying research motivation.

Examples of formats implementing the aforementioned characteristics are *Google Protobuffers (protobuf)* used for serializing structured data or the *Abstract Syntax Notation One (ASN.1)* as an interface description language defining data structures that can be (de-)serialized across multiple platforms. More simple approaches include formats like the *Binary JSON*-representation *(BSON),* which is a binary-encoded serialization of JSON-like documents, or *netstrings* developed by D.J. Bernstein [3] used as a simple way of encoding any string using length-prefix notation. As netstrings are simple and therefore efficient and pleasant to work with, we will be using them as a working example throughout the rest of the report consistent with the previous work in Calc-Regular languages, where netstrings were also examined.

5 Working Example: Netstrings

Netstrings represent an uncomplex encoding of self-delimiting strings using length-prefixes. As an example, we encode the single string "`Hello World`", including the space between the two words, as a simple non-nested netstring:

```
11:Hello World,
```

with "`11`" being the length-prefix describing the amount of bytes followed by the colon, marking the end of the length-prefix; "`Hello World`" being 11 bytes of content (including the space between "`Hello`" and "`World`") and the comma, which marks the end of the string and was only implemented by D.J. Bernstein as a feature to improve human readability. In fact, the comma can not be used for a parser to decide the end of a netstring, as commas themselves are allowed to appear in the message- or content-body and could therefore lead to irregularities.

The original definition by D.J. Bernstein did not allow variable nesting in netstrings, as a parser could not decide if a netstring acted as a container or as a simple string. This distinction is necessary, as the parser has to behave differently when parsing a container opposed to parsing a simple string. However, one can effortlessly extend the definition of netstrings to include this distinction in various ways. In the original definition of netstrings, leading zeroes in the decimal representation of the length-prefix were disallowed in any case. We altered this restriction slightly and decided to signal the appearance of a container with **one** leading zero in the corresponding length-prefix. Keep in mind that more than one leading zero is still forbidden and will lead to parsing errors. We encode the words "`Hello`" and "`World`" as separate netstrings packed in another netstring using the leading zero to mark the outer prefix as defining a container:

```
016:5:Hello,5:World,,
```

Keep in mind that the second comma at the end of this netstring is required as this comma belongs to the container with the length prefix "`016`" while the other commas belong to their respective netstrings with length prefixes "`5`".

A formal grammar for netstrings was the next step towards a working parser implementation. Such a grammar is a necessary requirement for LL(1) parsing. We consider a formal grammar $G_n = \{N, \Sigma, P_n, S\}$ for netstrings. With $N = \{S, R, T, D, Z\}$ being the set of non-terminal symbols, $\Sigma = \{$`0`, "`1-9`"[1], `:`, `byte`, `,`$\}$[2] the set of terminal symbols, P_n the set of productions of form $N \to (N \cup \Sigma)^*$ as listed in Table 1 and $S \in N$ the start symbol. Using the productions from Table 1, the creation of every possible netstring structure can be achieved[3]. Rules *a)* and *b)*, each derived from the start symbol S can be employed to create a container (leading zero) or a simple netstring respectively,

[1] The term "`1-9`" represents every digit from 1 to 9 and is shortened here to the term "`1-9`" to increase clarity.

[2] The comma "`,`" is also a terminal symbol.

[3] The correlation between a length prefix and the correct size of the corresponding byte-string can not be validated in a grammar.

Table 1: Productions $\in P_n$

a)	S	$\rightarrow$ 0 D : R ,
b)	S	$\rightarrow$ "1-9" D : T ,
c)	R	$\rightarrow$ S R $\mid \epsilon$
d)	T	$\rightarrow$ byte T $\mid \epsilon$
e)	D	$\rightarrow$ "1-9" Z $\mid \epsilon$
f)	Z	$\rightarrow$ 0 Z $\mid \epsilon$

Table 2: Split parse table of the formal grammar for netstrings

	0	"1-9"	:	byte	,
S	S $\rightarrow$ 0 D : R,	S $\rightarrow$ "1-9" D :T ,			
R	R $\rightarrow$ S R	R$\rightarrow$ S R			R $\rightarrow \epsilon$
D		D $\rightarrow$ "1-9" N	D $\rightarrow \epsilon$		
N	N $\rightarrow$ 0 N		N $\rightarrow \epsilon$		
$\mathrm{T}_{=0}$					T $\rightarrow \epsilon$
$\mathrm{T}_{>0}$				T $\rightarrow$ byte T	

both rules in column *c)* allow the representation of nesting or concatenation. Column *d)* is used to produce byte-strings of arbitrary size and the rules in columns *e)* and *f)* are needed to allow for any decimal representation of the length prefix without employing more than one leading zero.

6 Split Parse Tables in Calc-LL(1) Languages

Parse tables are indispensable when implementing a working LL(1) parser, as described in 3. Since the Calc-LL(1) parser would fundamentally behave the same way as a regular LL(1) parser but with a few additional procedures, parse tables are also needed for Calc-LL(1) parsing. Due to the nature of Calc-LL(1) languages, however, regular parse tables are not sufficient. This prompted the creation of an extended definition of parse tables, what we call *split parse tables*. We are parsing a Calc-LL(1)-type message, e.g. a netstring, that allows a sequence consisting of arbitrary bytes. At every input token, the parser has to decide if the byte it is currently parsing belongs to the actual content-field of the current message, if it describes a format-byte of the current message (e.g. the end-of-content comma in netstrings), or if it potentially already belongs to a subsequent new message in the information stream. As the parser only relies on the length prefix value to decide when to stop expecting content-bytes as input tokens, a mechanism has to be introduced to signal the parser when to interpret a token as an arbitrary content-byte or as a specific format-byte (e.g. the comma at the end of every netstring is required to form a correct netstring but a comma can also appear as a content-byte). As the parser looks into the parse table to decide which production rule to pursue further, we introduced a split at the corresponding non-terminal column or row in the parse table. One fraction signals the parser to look into this column or row of this non-terminal if the end of the content-field as described by the length prefix is not yet reached, the other fraction will be looked at if the end of the content-field was reached and a format-token or new message may be read. It could also be argued to split the parse table at the terminal symbol row or column. However, we have decided to split at the non-terminal as it only adds the procedure of checking if the content-field length has been worked through or not[4] to the usual routine of regular LL(1) parsers. A representation of the split parse table for our formal grammar for netstrings as described in 5 can be found in Table 2. In the leftmost column you can see all non-terminals $N \in G_n$, the first row includes all terminal symbols $\Sigma \in G_n$. The parse table cell contain either nothing or one of the production rules from P_n. The parse table *split* is realized in the last two columns for the non-terminal symbol *T* . The notation $T_{=0}$ indicates that this column is looked at when the parser has worked through the byte-string and decremented the corresponding length prefix value for each parsed byte-token down to 0. Therefore, for netstrings, a comma is explicitly expected as the next token and not just any byte. Contrary to this is the notation $T_{>0}$ which gets looked at while the length prefix value is still getting decremented and therefore while bytes of the content field are still expected as input tokens. Without this split, a collision in the parse table cell would occur[5] as the terminal symbol *comma* is technically also a byte.

"Byte collisions" is the term that was used during this research for the case where the split parse table is necessary during parsing as described above. With netstrings and formats like *BSON*, this collision occurs at the end-of-content

[4] As indicated by the length prefix value.

[5] which would render the grammar not parsable with the underlying LL(1) technique.

Table 3: Basic structure of a split parse table for Google protobuf

	type field byte	length field byte	content field byte
$=0$	x		
>0			x

symbol. For other formats that do not use an end-of-content symbol, the "byte collision" also appears. As an example, we look at *Google Protobuffers (protobufs)*. To give a crude introduction into the structure of protobufs, we take a closer look at the encoding of the word "`testing`" with characters being encoded using hexadecimal ASCII representation: "`1A 07 74 65 73 74 69 6E 67`". Each encoded message begins with a type field indicating a *field number* and *wire type*. This is realized in the first byte, here marked in the color red. The meaning and specific usage of these properties is currently not important to us. After the first byte follows the length prefix, here colored in blue and representing the value 7 and then the actual content trailing at the end, as many bytes as defined in the length prefix. Now we consider this byte stream of three messages encoded using *Google Protobuffer*:

`1A 07 74 65 73 74 69 6E 67 12 05 48 65 6C 6C 6F 0A 05 57 6F 72 6C 64`

These three messages are a crude example representing very simple protobuf-structures each containing a short string, "`testing`", "`Hello`" and "`World`". As there is no end-of-content symbol with this format, a new message's type field byte follows right after the last content byte of the previous message. As the type field is required and may not be any arbitrary byte representation but a meaningful definition of the *field number* and *wire type*, the parser has to handle these two subsequent bytes differently. This is where the "byte collision" would occur in languages without end-of-content symbols. In this specific byte stream, the collisions would occur as marked below with a ↯-symbol, where one message ends and the next one begins:

`1A 07 74 65 73 74 69 6E 67↯12 05 48 65 6C 6C 6F↯0A 05 57 6F 72 6C 64`

A basic structure for the *split parse table* belonging to Google Protobuffers can be found in Table 3. A *split parse table* generator for formal grammars was developed in *python* during this research and can be found at [4].

7 Parsing Calc-LL(1) Languages

Calc-LL(1) parsing is a slightly extended procedure from regular LL(1) parsing. Besides the already existing conventional LL(1) parsing routine, there are a couple of additional features that have to be implemented for the parser to be capable of parsing Calc-LL(1) languages correctly:

The split parse table as described in 6 is used in the specific case of the so called "byte collisions", as also mentioned in 6. This can be best explained when looking at the formal model of LL(1) parsing. When the parser pops a non-terminal symbol X from its internal stack and X has the two conditional columns or rows with $X_{=0}$ and $X_{>0}$ in the split parse table, the parser evaluates the conditions and proceeds to look at the column or row where the condition is true.

An additional stack has to be implemented as to allow the storage of length prefix values for variable nesting with a depth higher than one. After a container and its contents have been parsed successfully, the length prefix value of its parent container can be popped from this second stack and worked with until this containers contents also have been worked through etc. As this stack is necessary to allow the correct parsing of variable nesting using containers, it was called the *container stack* during research.

The current position of the active input token being parsed has to be saved and updated every time the parser advances one input token. This is a very simple but indispensable task, as this allows the parser to determine when the message has to end according to its length prefix. This *current-position-value* is also used to determine if a nested message exceeds its upper containers size as described in the length prefix values. This is done by adding the length prefix value of the current message/container to the *current-position-value* and comparing it to the value on top of the *container stack*. If this sum is **not** smaller than the value on top of the stack, the nested message's or container's length prefix exceeds the boundaries of its upper container and is therefore formed incorrectly and should not be parsed.

A reference parser implementation for the working example format in *netstrings* had been created during the research period of this project with its code being located at [4]. Table 4 presents an overview of error messages that the

Table 4: Possible error messages returned by the netstring parser

Error message	Situation	Short form
Container size must be numerical.	Non-digit characters appear after a leading zero where digits of the length field are expected.	Container size not numerical!
Netstring of size `$string_size$` exceeds upper container boundaries!	A simple nested netstring's length field indicated that the length of the netstring's contents would exceed its upper container's size	String exceeds upper limit!
Container of size `$container_size$` exceeds upper container boundaries!	A container-netstring's length field indicated that the length of the container's contents would exceed its upper container's size.	Container exceeds upper limit!
Expected another netstring to begin due to leading zero in the upper container.	After reading a container size (with leading zero) and a colon (`":"`), the next byte is not a digit and therefore invalid as a new netstring has to start after a container declaration using leading zero.	Netstring expected!
Expected a length-prefix definition at string-position `$current_position$`.	A netstring did not start with digits and therefore not with a length prefix and is therefore invalid.	Length-prefix expected!
Unmatched ",". Netstring contents possibly longer than length field indicated.	A comma was expected after a netstring's content-bytes reached the size defined by the length-prefix but another byte was read instead.	Contents exceed length prefix value!

parser is able to return when spotting ill-formed netstring structures. Finally, Table 5 is an overview of several netstring examples that are both either well formed or ill-formed. The left arrow symbol in the column "Parser output" indicates that the output corresponds to the input as defined in the first column, therefore it also indicates a correctly formed netstring.

8 Outlook

The research presented in this report established some foundations on why and how to approach Calc-LL(1) language parsing. As we focused on creating implementations and validating our conjectures and ideas using practical approaches, a clean formal and theoretical representation of Calc-LL(1) parsing machines in the form of an extended deterministic pushdown automaton is yet to be established. Of course, also a more precise definition on the structure of Calc-LL(1) languages and grammars goes hand in hand with such an automaton definition. With these foundations eventually established, creating parser generators could be the next logical step in the research process. As this topic is highly practical, implementing more potential reference implementations should always be considered.

References

[1] Grosch, Koenig, Lucks: Taming the Length Field in Binary Data: Calc-Regular Languages
in *2017 IEEE Security and Privacy Workshops (SPW)*, IEEE, 2017.

[2] Grune, Jacobs: Parsing Techniques
Springer, 2006

[3] D.J. Bernstein: Netstrings, 1997
`https://cr.yp.to/proto/netstrings.txt`

[4] Netstring parser and split parse table generator programming code
`https://github.com/YxJannis/Calc-LL1-project-2019`

Table 5: Examples for correct parsing behaviour of the netstring-parser

String to be parsed	**Parser output**	**Error thrown**
0:,	←	-
0:a,	ERROR!	Contents exceed length prefix value!
03:0:,,	←	-
04:0:,,,	ERROR!	Length-prefix expected!
06:0:,0:,,	←	-
05:0:,0:,,	ERROR!	Container exceeds upper limit!
07:0:,1:d,	←	-
028:03:0:,,06:0:,0:,,07:0:,1:d,,,	←	-
3:abc,	←	-
3:abcd,	ERROR!	Contents exceed length prefix value!
3:abc,tgfr	3:abc,	-
03:abc,	ERROR!	Netstring expected!
04:03:abc,,	ERROR!	Container exceeds upper limit!
05:4:0000,,	ERROR!	String exceeds upper limit!
0ab:3:abc,,	ERROR!	Container size not numerical!
ab:cde,	ERROR!	Length-prefix expected!
7:03:abc,,	←	-
09:2:ab,1:c,,	←	-
024:011:2:abc,2:de,,5:abcde,,	←	-
010:3:abc,1:d,,010:3:abc,1:d,,	010:3:abc,1:d,,	-
016:5:h3ll0,5:w0r1d,,hjk	016:5:h3ll0,5:w0r1d,,	-
016:5:12,::,5:::,:,,,::,	016:5:12,::,5:::,:,,,	-