# Research Report: Parsing PEGs with Length Fields in Software and Hardware

Zephyr S. Lucas, Joanna Y. Liu, Prashant Anantharaman, and Sean W. Smith
Dartmouth College
Hanover, NH, USA 03755
{zephyr.s.lucas.gr, joanna.y.liu.22}@dartmouth.edu
{pa, sws}@cs.dartmouth.edu

***Abstract*—Since parsers are the line of defense between binaries and untrusted data, they are some of the most common sources of vulnerabilities in software. Language-Theoretic Security provides an approach to implement hardened parsers. We specify the binary format as a formal grammar and implement a recognizer for this formal grammar. However, most binary formats use constructs such as the length field, repeat field, and an offset instruction. Most grammar formats do not support these features.**

**Building on PEGs and calc-regular languages, we propose *Calc-Parsing Expression Grammars (Calc-PEGs)*, a formalization of parsing expression grammars that supports the length field. We design an algorithm to parse Calc-PEGs in $O(n^2)$ time and a parallel algorithm to parse Calc-PEGs in $O(n)$ time. We also present *Pegmatite*, a tool to generate these parsers in C, with an option to generate VHDL code.**

## I. Introduction

Since parsers are the line of defense between binaries and untrusted data, they are some of the most common sources of vulnerabilities in software. Language-Theoretic Security provides an approach to implement hardened parsers. Many real-world data formats use the length construct, so for a tool to be useful in practice it needs the capability to parse the length construct. In 2017, Marie Grosch, Koenig, and Lucks proposed calc-regular languages as a theoretical foundation for such tools [1].

However, significant problems still remain. First, calc-regular languages do not apply to recursive languages such as ASN.1. We build on Parsing Expression Grammars (PEGs) to define Calc-PEGs. Apart from the operations supported by PEGs, we add an operator that takes two nonterminal arguments. The first argument is the length parameter to support fields of various lengths (1 byte, 2 bytes, etc.). Pegmatite resolves this parameter to extract the value and apply it to the second nonterminal. We discuss Calc-PEGs in detail in Section III-A.

Second, grammar-based packet filtering in the CPU is time-intensive. In prior work implementing Hammer-based parsers for various Internet-of-Things protocols, we saw that the latency added by these parsers were in the order of milliseconds. To support packet filtering on the router, Pegmatite generates VHDL code from parser declarations. We built a toolkit to parse PEG primitives in VHDL. Pegmatite generates invocations to these primitives.

Finally, as of writing this paper, no FPGA implementations are available for PEG parsing algorithms. We present approaches to parsing PEGs and Calc-PEGs using the Scaffold Automata method in FPGAs.

For any parser generator to be useful for data formats, we need to satisfy at least these two requirements:

- **R1**: Input grammar language must be easy to use and debug and must support recursive formats and formats that use the length field.
- **R2**: The parsers we generate must run in $O(n)$ time.

To this end, our contributions are as follows:

- We introduce Calc-PEGs, a language that adds support for the length field in PEGs.
- We present parallel algorithms to parse PEGs and Calc-PEGs in $O(n)$ time.
- We built Pegmatite, a tool to generate VHDL implementations of our PEG and Calc-PEG parsers.

The rest of the paper is organized as follows. Section II presents the necessary background on PEGs, the scaffold automata model, and prior work such as calc-regular languages that explore the length field. Section III presents our core concepts such as Calc-PEGs and Pegmatite. We discuss some limitations and lessons learned in Section IV. Section V presents related work, and Section VI concludes the paper.

## II. Background

### A. Parsing Expression Grammars (PEGs)

PEGs were first introduced by Ford as a foundational deterministic recognition-based formal language syntax [2]. The fundamental difference between PEGs and a more traditional grammar class such as Context Free Grammars (CFGs) is the removal of the non-deterministic choice (|) and addition of a prioritized choice operator (/). The prioritized choice will first try to parse its first option, and only if that rejects will it try the second argument. While this may seem like a fairly minor change, it dramatically affects the expressive power of this class of grammars as well as allows parsing of an arbitrary grammar in linear time. These features—along with elimination of non-determinism—make PEGs more attractive than CFGs.

There are many equivalent definitions of a PEG, however all of them are fundamentally based on the recursively defined

*PEG expressions*. A PEG expression is any of the nine possibilities shown in Table I. Note that a PEG expression depends on both an alphabet $\Sigma$, and a set of nonterminals $V$, so we notate the set of PEG expressions as $\mathcal{E}(\Sigma, V)$.

TABLE I
THE OPTIONS FOR A PEG EXPRESSION. WHERE $a$ IS ANY CHARACTER OF THE ALPHABET, $e_1$ AND $e_2$ ARE PEG EXPRESSIONS THEMSELVES, AND $N$ IS ANY NON-TERMINAL IN THE PEG.

| | |
|---|---|
| `Epsilon` | $\epsilon$ |
| `Fail` | $f$ |
| `Character` | $a$ |
| `Any` | . |
| `And` | $\&e_1$ |
| `Not` | $!e_1$ |
| `Prioritized Choice` | $e_1/e_2$ |
| `Concatenation` | $e_1 \circ e_2$ |
| `Nonterminal` | $N$ |

Also note that the sub-expressions given in Table I are not minimal; `Fail`, `Any`, and `And` can all be easily be removed and the expressive power of PEGs will remain unchanged. Similarly, sometimes other operators such as an "optional" or a "greedy Kleene star" are added to the list of sub expressions. [2], [3] While these additional operators can improve usability, they can be constructed from the above constructs and therefore also do not affect the expressive power.

A PEG is a 4-tuple $\langle V, \Sigma, N_0, r\rangle$, where $V$ is a set of nonterminals, $\Sigma$ is the alphabet over which the language is defined, $N_0 \in V$ is the starting non-terminal, and $r : V \to \mathcal{E}(\Sigma, V)$ is a function which maps each non-terminal to a PEG expression. Unlike more classical models of computation which will either recognize or reject a string $s$, a PEG can either *reject* $s$ or *accept* and consume some prefix of $s$. If the consumed prefix of $s$ is the whole string $s$, then it is said the PEG *recognizes* $s$.

To parse an input string $s$, a PEG tries to match the start non-terminal $N_0$ to $s$. When matching any non-terminal $N$ to some suffix $x$ of $s$ (at the start of the operation, while matching $N_0$, $x = s$, but as matching continues, some characters are consumed leaving only a suffix to be matched), $r(N)$ is checked and the following case work is followed:

- $\boxed{r(N) = \epsilon}$ : `Epsilon` will always accept $x$ and consume 0 characters.
- $\boxed{r(N) = f}$ : `Fail` will always reject.
- $\boxed{r(N) = .}$ : If $x = \epsilon$, then `Any` will reject, otherwise `Any` will accept consuming 1 character from $x$.
- $\boxed{r(N) = a}$ : If $x = \epsilon$ or the first character of $x$ is not $a$, then `Character` will reject, otherwise it will accept consuming 1 character.
- $\boxed{r(N) = \&e_1}$ : If $e_1$ accepts $x$, then $N$ will accept $x$ consuming 0 characters, otherwise it will reject.
- $\boxed{r(N) = !e_1}$ : If $e_1$ accepts $x$, then $N$ will reject $x$, otherwise it will accept consuming 0 characters.
- $\boxed{r(N) = e_1/e_2}$ : If $e_1$ accepts $x$, then $N$ will accept consuming the same number of characters as $e_1$, otherwise, $N$'s behavior is the same as $e_2$'s
- $\boxed{r(N) = e_1 \circ e_2}$ : If $e_1$ rejects $x$, then so will $N$. If not, then it will leave some suffix $x'$. Similarly if $e_2$ rejects $x'$, then so will $N$, however if $e_2$ also accepts, then $N$ will accept consuming what $e_1$ and $e_2$ consumed.
- $\boxed{r(N) = N'}$ : $N$'s behavior is the same as $N'$'s behavior.

A parser which recursively follows this definition is called the *naive PEG parser*.

While the definition of a PEG allows rules to be *any* PEG expression, there is an obvious problem of left recursion. For example, in a context free language, a grammar such as $\boxed{A \leftarrow Aa|\epsilon}$ would be a perfectly valid representation of $a^*$, however, within PEGs, $\boxed{A \leftarrow Aa/\epsilon}$ causes an infinite loop. For any string $s$, the grammar will try to match $A$ by first checking if $Aa$ matches. But this requires checking if $Aaa$ matches, and so on. Because PEGs are deterministic, it will always try matching the $Aa$ subexpression before trying $\epsilon$, and therefore will never move forward. This is known as *left recursion*, and to avoid PEGs which will not terminate, the idea of a *well formed* PEG was introduced [2]. A PEG is well formed if it satisfies a fairly technical structural property that ensures no left recursion will occur (though there are still obscure PEGs that are neither well formed nor left recursive). Conceptually, a PEG is well formed if and only if each nonterminal $A$ will always consume at least one character before it is required that $A$ be matched on the string again. Most useful PEGs are either well formed or can be easily rewritten as a well formed PEG, so often left recursion does not pose a major problem, however, this scaffold parsing technique requires the PEG to be well formed to terminate.

Additionally, while the PEG expressions for a given nonterminal can get fairly large, it must always (by definition) be constructed out of the PEG expressions given in Table I. So while a PEG for the $a^*$ regular expression might be written succinctly as

$$\boxed{A \leftarrow aA/!.}$$

it could also expanded by flattening out the compound expressions resulting in the grammar

$$\boxed{\begin{array}{lll} A & \leftarrow & B/D \\ B & \leftarrow & C \circ A \\ C & \leftarrow & a \\ D & \leftarrow & !E \\ E & \leftarrow & . \end{array}}$$

We call a PEG in this latter case in *PEG normal form*. In general, a grammar is in PEG normal form if each non-terminal maps to a single expression from Table I. Additionally, a PEG in PEG normal form will not have any `Nonterminal` expressions, as those can always be followed until a different PEG expression is reached. For example, a grammar with the rule $\boxed{A \leftarrow B}$ can be removed with any other instances of $A$

within the other expressions replaced with a $B$. While, for usability's sake, it is important to be able express the PEG in compounded form, the Pegmatite parser first compiles that into a normal form before beginning the parsing process.

### B. Scaffold Parsing

The packrat parser, introduced along with PEGs in Ford's original paper, has continued to be used as the standard PEG parser [2]. Packrat is a memoized version of the naive PEG parsing algorithm. While this ensures a linear runtime, it uses high-level infrastructure such as hash tables to compact the memoization as well as a stack to keep track of the current parse tree. As these structures translate poorly to hardware, instead of using the standard packrat parser for PEGs, our code is based on the scaffold automata model introduced by Loff, Moreira, and Reis [4]. Similar to how a PDA is a finite state machine with access to a stack, a scaffold automata is a finite state machine with access to a finite-out-degree directed acyclic graph, called the *scaffold*. For each input character consumed, the state machine transitions to a new state and a new vertex is created and added to the scaffold along with some constant number of edges. In our model, the scaffold is a table with a constant number of rows. Each column of the table represent a vertex in the scaffold, and each entry in the column represents an out-edge by pointing to a 'further down' column in the table. In this way, the finite-out-degree is maintained by the finite number of rows, and the acyclic property is maintained by the ordering of the columns in the table. For each input character consumed, the table is updated by filling in a new column (the new vertex), each cell in the column is filled with a pointer to a previously filled out column (the new edges).

For a grammar $G$ with $k$ non-terminals in PEG normal form parsing an input $s$ with $n$ characters, we initialize a scaffold $A$ of dimensions $k \times (n+1)$. Each entry $A[i, j]$ in the scaffold represents how many characters will nonterminal $i$ consume while recognizing the last $j$ characters of $s$.

For a grammar in normal form, each scaffold entry can filled out according to a small set of operations. Each row of the scaffold is associated with a particular nonterminal, and that nonterminal determines how each entry in that row is filled out. Based on the PEG expression for nonterminal $i$ the following logic is applied:

- $\boxed{i \leftarrow \epsilon}$ : Because $\epsilon$ can match any string, this will always accept and consume 0 characters. Therefore $A[i, j] = 0$
- $\boxed{i \leftarrow f}$ : The fail operation will always fail so $A[i, j] = f$. In the Pegmatite implementation, $f$ is the only non-natural number that a scaffold entry can be filled in with and $-1$ is used to represent it.
- $\boxed{i \leftarrow .}$ : Here $j$ is checked to determine if this is off the end of the string or if the parser is at a character in the input string. If $j = 0$, this is matching with the "base case" column of the scaffold and the parse will fail ($A[i, j] = f$). If $j \neq 0$, then this is at a point in the input string, and, because `Any` will accept any character and consume it, $A[i, j] = 1$.
- $\boxed{i \leftarrow a}$ : Just like with the `Any` expression, if $j = 0$ then the character will fail to parse and $A[i, j] = f$. However, if $j \neq 0$, then a further condition must be met, namely that the character $a$ is actually found at the $j$th last position of $s$ (more directly, if $a$ is the first character of the suffix of $s$ containing $j$ characters). So if $s_{n-j} = a$ then $A[i, j] = 1$, otherwise $A[i, j] = f$.
- $\boxed{i \leftarrow \&e_1}$ : Here, $e_1$ is some other non-terminal and has a position in the scaffold, so the parser must check if $e_1$ succeeded or failed at this position in the input string. If $A[e_1, j] = f$, then $e_i$ failed, so $A[i, j] = f$. If $A[e_1, j] \geq 0$, then $e_1$ succeeded, so $\&e_1$ should succeed and consume no input, meaning $A[i, j] = 0$
- $\boxed{i \leftarrow !e_1}$ : The same logic as the `And` case applies. If $e_1$ succeeded ($A[e_1, j] \neq f$, then $!e_1$ should fail and $A[i, j] = f$. If $A[e_1, j] = f$, then $!e_1$ should succeed and consume no input, therefore $A[i, j] = 0$
- $\boxed{i \leftarrow e_1/e_2}$ : Here if $e_1$ succeeded, then $i$ will consume exactly what $e_1$ consumed. Therefore if $A[e_1, j] \neq f$, then $A[i, j] = A[e_1, j]$. If $e_1$ did fail, then the prioritized choice switches over to the second peg expression and tries to parse that. So, if $A[e_1, j] = f$, then $A[i, j] = A[e_2, j]$.
- $\boxed{i \leftarrow e_1 \circ e_2}$ : For the concatenation expression, $e_1$ must first succeed, and then $e_2$ must also succeed on whatever part of the input wasn't consumed by $e_1$. If neither of these fail, then $e_1 \circ e_2$ will succeed and consume the sum of what $e_1$ and $e_2$ consumed. Following this logic, if $A[e_1, j] = f$, then $A[i, j] = f$, otherwise we examine $A[e_2, j - A[e_1, j]]$ (this is $e_2$ at the point in the input after $e_1$ has been consumed)[1]. If $A[e_2, j - A[e_1, j]] = f$, then $A[i, j] = f$, otherwise $A[i, j] = A[e_1, j] + A[e_2, j - A[e_1, j]]$.

Through this algorithm, the scaffold for any well-formed PEG can always be filled out.

### C. Is the length field necessary?

As we noted, many real-world data formats use the length construct, so for a tool to be useful in practice it needs the capability to parse the length construct.

Since proving a particular language or construct is not already in PEG is notoriously difficult, it is not known whether or not PEGs can already express the length field. No known implementations of the length field using the base PEG constructs currently exist. So, if we want parse a grammar which requires use of the length field, we need to add this construct PEGs.

While we conjecture the length field is not in PEGs, there is some reason to believe that it could be. For some encoding

[1]Note that $A[i, j] \leq j$ so $j - A[e_1, j]$ will never cause an out of bounds error

$$
\begin{array}{lcl}
S_{\text{Start}} & \leftarrow & (\&B_d)B\texttt{1}Z\texttt{!}. \ / \ \texttt{x}Z\texttt{!}. \\
Z & \leftarrow & \texttt{0}Z \ / \ \epsilon \\
B & \leftarrow & \texttt{a}(\&B_d)B\texttt{1}B_n \ / \ \texttt{a}(\texttt{!}B_d)B \ / \ \texttt{x}B_n \\
B_n & \leftarrow & \texttt{0}B_n \ / \ \&\texttt{1} \\
B_d & \leftarrow & (\&D_d)D\texttt{.!}. \\
D & \leftarrow & \texttt{a}(\&B_d)B\texttt{1}B_n \ / \ \texttt{a}(\&D_d)D\texttt{.} \ / \ \texttt{a}(\texttt{!}D_d)D \\
 & & \quad / \ \&(\texttt{x1}) \ / \ \texttt{x}D_i \\
D_i & \leftarrow & \&(\texttt{01}) \ / \ \texttt{0}D_i \\
D_d & \leftarrow & \&(P\texttt{a}P_f \ / \ \texttt{x}) \\
P & \leftarrow & \&(\texttt{a}P_f) \ / \ \texttt{a}\&(P\texttt{a}P_f) \ / \ \texttt{a}P\texttt{a} \\
P_f & \leftarrow & \texttt{x} \ / \ B_d
\end{array}
$$

Fig. 1. A Grammar for the reversed length field. The variable $P$ counts off powers of two since the last time $P_f$ was true. This gets matched with a digit position called $D$, which counts through the string in the length field. $D$ is set to an initial position by $D_i$ and decrements to the next digit position each time $D_d$ is true (which happens each time $P$ shows we've reached another power of two). $B$ keeps track of the current leading one of the binary length field. Each time $D$ reaches the end of the string, $B_d$ becomes true, and we move to the next binary digit $B_n$. This grammar is not at all intuitive, however sheds some small light on how non-trivial behavior is possible in a PEG.

of a number function similar to the reverse of the length field does exist. In particular, we were able to write a PEG for the language:

$$L_{\text{Reverse Len}} = \{a^n x[n]_2\}$$

Where $a$ and $x$ are characters, $n$ is a natural number, and $[n]_2$ is the binary representation of $n$. The PEG for this is extremely convoluted, but can be seen in Figure 1. PEGs are not known to be closed under reverse, so it is very possible that there is no PEG for a generic length field where the length is placed before the data. However, it is still an open problem with no clear answer.

In 2017, Marie Grosch, Koenig, and Lucks, facing similar difficulty parsing practical data formats, extended regular expressions to calc-regular to include it [1]. They showed why the length field was not context free, and extended regular languages to allow for easy parsing of certain file formats. We extend this work further to by adding the length field to PEGs. While regular expressions are well suited for many parsing problems, PEGs strictly more powerful, and if the length field can be added and still maintain efficient parsing, then Calc-PEGs will offer a versatile tool capable of handling many of the real-world data formats and protocols.

## III. Pegmatite: Parsing Real-World Formats in Hardware

### A. Calc-Parsing Expression Grammars (Calc-PEGs)

One of the major contributions of this tool is the addition of the *length field*. This construct augments the PEGs operators by adding the following construct. The production

$$\boxed{A \leftarrow \texttt{LEN}(L, D, f)}$$

is parsed according to the following algorithm:

- Read (and consume) $L$ from the current input. If $L$ is not recognized then LEN construct will fail and not be recognized.
- Decode the expression consumed by $L$ and use $f$ to convert this to a non-negative integer, which we will refer to as $n_L$. If what was consumed by $L$ is not in a form that $f$ accepts, then the LEN construct will fail and not be recognized.
- Read and consume $n_L$ additional characters from the input. If the remaining input does not have $n_L$ characters, we (again) fail and do not recognize the input. We then check to see if the $n_L$ characters that we consumed, will be recognized (as a standalone string) by $D$. If so, then we accept and if not, then we reject.

This means the length field will either reject or it will recognize the input and consume $(|L| + n_L)$ characters.

### B. Accommodating the length field

In order to accommodate the length field, the Pegmatite parser constructs many scaffolds, one for each character of the input. With respect to complexity, this means it requires $O(n^2)$ space.

The key algorithmic change involves the addition of $O(n)$ scaffolds. Before, a scaffold entry $A[i, j]$ represented the number of characters non-terminal $i$ would consume if given the last $j$ characters of $s$ as input. Now, there are $n+1$ scaffolds each indexed by $l$ with $0 \leq l \leq n$. Each scaffold $A_l$ only has $l + 1$ columns in it, so $j$ is now bounded by $0 \leq j \leq l$ instead of the normal $0 \leq j \leq n$. Entry $A_l[i, j]$ now represents how many characters non terminal $i$ consumed when parsing the first $j$ characters of the last $j + n - l$ characters of $s$ as it's input. In other words, how many characters are consumed when parsing the substring $s[l - j : l]$.

For this new "multiple scaffold" parsing scheme each of the other PEG expressions follow their respective rules for each individual scaffold. The length field adds a new case to the rules and is the only operation which can look into the other scaffolds.

- $\boxed{i \leftarrow \texttt{LEN}(e_1, e_2, f)}$ : This operation is broken down into the three steps associated with the length field's definition: parsing the length parameter, the integer conversion, and parsing the data parameter.
  - First the "length" parameter must be read from the current input string. If $A_l[e_1, j] = f$, then the length expression was not matched by $e_1$ and so $A_l[i, j] = f$. If $A_l[e_1, j] \geq 0$, then the length parameter was matched to some sub-string of $s$, with length $A_l[e_1, j]$. In particular the sub-string indexed by $s[l-j : l-j+A_l[e_1, j]]$ (Including the first index but excluding the last).
  - The function $f$ is then used to convert this into a number, and store this number in $n_L$, specifically we let $n_L = f(s[l - j : l - j + A_l[e_1, j]])$. If $n_L$ is larger than the current rest of the string (after having consumed $e_1$, then this operation should fail. Specifically, if $n_L > j - A_l[e_1, j]$, then $A_l[i, j] = f$.
  - Finally, $e_2$ must accept the substring partitioned out by the length parameter. This means checking if $A_{l-j+A_l[e_1,j]+n_L}[e_2, n_L]$ succeeds or not. If

$A_{l-j+A_l[e_1,j]+n_L}[e_2, n_L] = f$, then should fail as the "data" parameter ($e_2$) was not recognized so $A_l[i,j] = f$. Otherwise, $e_2$ was recognized, so the whole function should accept. The number of characters consumed should be whatever $e_1$ parsed, as well as the whole substring that $e_2$ was parsing (not just what $e_2$ consumed). Therefore $A_l[i,j] = A_l[e_1,j] + n_L$.

With this additional scaffold logic the parser is now able to recognize any well formed Calc-PEG.

### C. Parallel PEGs Parsing

The important extra expressive power gained by the addition of multiple scaffolds, does incur a complexity cost to the space and run time of Pegmatite. For "length field free" grammars, only one scaffold needs be filled in, requiring $O(n)$ time and space. However, with the additional $n$ scaffolds, the Calc-PEG parser now requires $O(n^2)$ time and space. This is where another innovation of Pegmatite, the VHDL code generator, helps. By allowing specifications to be built for hardware, the parsing process can remain efficient.

An important observation of this algorithm is each scaffold entry $A_l[i,j]$ is completely independent of another scaffold entry $A_{l'}[i',j]$ (for $l \neq l'$); meaning that neither entry must be filled in before another. So, these entries can be filled in simultaneously due to the parallel nature of hardware. While the generated VHDL still requires $O(n^2)$ space, we still only require $O(n)$ time to populate all the scaffolds and parse the input.

## IV. Discussion

We are currently in the process of implementing several data formats such as MQTT and various ASN.1-based formats in Pegmatite. We will generate VHDL for these formats and evaluate the generated code on real-world network traffic. Our library is currently 1703 lines of code. To evaluate Pegmatite, we plan to perform the following evaluations.

1) What is the programmer effort required to implement various real-world formats in Pegmatite?
2) How do our Pegmatite-generated VHDL parsers compare with parsers generated from Hammer-based C parsers using HLS tools?

Apart from answering these questions to evaluate Pegmatite, we also believe we need to address some limitations in our current system. Currently, our parser starts with the last byte received and moves from right to left. The first byte of the packet is processed last. This feature of the scaffold-automata model of parsing creates an optimization problem. We do not start parsing a packet until we receive the entire packet.

*a) Reversing the grammar:* Hence, one optimization considered, however, has yet to be incorporated into Pegmatite, is to reverse the grammar. This approach's upside is that the input string can be parsed from front to back, allowing processing to begin before the full input string has been received. This *could* allow the parsing to occur at line speed.[2]

*b) Pipelining:* Another optimization we are working on is to parse multiple packets at the same time. We are using a pipelining approach to fill the scaffold step by step. When a particular packet moves to the writing state (when we transmit the packet again after ascertaining that it is well-formed), we begin parsing another packet.

*c) TCP Reassembly:* Our VHDL implementation is a standalone parser that only parses the application layer packets. Packets often use IP fragmentation or TCP disassembly to split a packet across multiple packets. Since our parser needs the entire packet to parse and make sure it is well-formed, TCP reassembly presents a challenge. Several approaches have been employed to tackle TCP reassembly in network intrusion detection. We plan to use such an approach to ensure that our parsers receive entire packets [5] [6].

*d) Seek and Repeat:* On a more theoretical note, other constructs such as a "seek" field (reading an address, and checking that the bytes that far into the file / stream match some nonterminal) or a "repeat" field (reading in a number and checking that there are that many of some object afterward) are also useful tools to handle real-world data formats. These constructs are not currently present in many of the theoretical parsing models, like PEGs or CFGs, so it would be useful to extend PEGs even further to handle these. While the "seek" field can be added to the scaffold model under a certain (practical) set of assumptions, the repeat field has proven more difficult. We would like to implement both in the future.

## V. Related Work

### A. Parsing in FPGAs

Researchers have explored parsing context-free grammars (CFGs) in FPGAs. Ciresson et al. [7] presented an algorithm to parse unrestricted CFGs in FPGAs and demonstrated a 240x speedup to use it in natural language processing applications. Bordim et al. [8] implemented the Cocke-Kasami-Younger algorithm to parse CFGs in an FPGA and showed a 750x speedup over a software implementation. Taylor proposed generating FPGA code from C using High-Level Synthesis (HLS) tools [9]. Instead of providing our C code to an HLS engine, we generate the corresponding VHDL from our C code.

In this paper, we build on prior work implementing dynamic programming-based parsing algorithms in VHDL. We presented algorithms to parse PEGs and Calc-PEGs on FPGAs.

### B. Parsing Approaches

In this paper, we build on several prior works on parsing. Grosch et al. [1] in their paper showed that the length field is not context-free. They then proposed "calc-regular languages": where they extended regular languages to support length fields.

[2]In particular, if we are allowed some constant (based on the grammar (less than the number of non-terminals)) number clock cycles between receiving each byte of input, we can process the string as it arrives.

We build on this prior work by adding support for the length field to PEGs.

In 2020, Blaudeau et al. [10] built a verified PEG parser in PVS. They defined what a well-formed PEG is, and proved that their implementation terminates for all well-formed PEGs. TRX also formalized the notion of termination for PEGs [11]. Van Geest et al. [12] described binary languages using the data types in a general purpose dependently typed language. They demonstrated their parser and pretty printer on IPv4 protocol.

Ramanandro et al. [13] also built a parser generator for tag-length-value languages. They generated parsers using a simple message format description and verified a parser-combinator toolkit using F* and Low*. Although they proved that their C implementations did not have any memory corruption bugs, they did not focus on any specific formal class of languages. Instead, we focus on a new class of languages, Calc-PEGs, and show a parsing algorithm for this class of languages.

Bangert et al. [14] implemented a parser generator, Nail. Pegmatite differs from Nail in that it generates recognizers for Calc-PEGs, whereas Nail generates invocations to a parser-combinator with additional predicate functions. Kaitai [15], Parsley [16], and DFDL [17] are other parser generator toolkits that support wide ranges of protocols. Pegmatite, however, supports languages that use the tag-length-value construct, as well as recursive languages.

## VI. Conclusions

In this paper, we designed and implemented Pegmatite, a tool to generate parsers from a Calc-PEG grammar description (R1). Pegmatite-generated parsers run in linear time (R2).

Much of the future work to support a broader set of data formats remain. Our efforts are focused on three areas:

*a) Parsing in FPGAs:* We are currently implementing a parser for Calc-PEGs in VHDL. We implemented a set of PEG primitives in VHDL. Pegmatite will generate invocations to these VHDL primitives.

*b) The seek and repeat field:* Formats such as DNS include a *repeat* field, where we parse an integer field *n* and ensure a term repeats *n* times.

*c) Supporting Data Description Languages in Pegmatite:* Currently, Pegmatite uses a Backus Normal Form (BNF) notation as grammar input. In the future, we would like to support several Data Description Languages such as Parsley [16], Kaitai [15], and DFDL [17] that support parser generation from grammar input.

## Acknowledgments

We would like to thank Garret Andriene and Sameed Ali for their inputs during the early stages of this project.

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075 and Contract No. HR001119C0121. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

## References

[1] S. Lucks, N. M. Grosch, and J. König, "Taming the Length Field in Binary Data: Calc-Regular Languages," in *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2017, pp. 66–79.

[2] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 2004, pp. 111–122. [Online]. Available: https://doi.org/10.1145/964001.964011

[3] C. Blaudeau and N. Shankar, "A verified packrat parser interpreter for parsing expression grammars," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 3–17. [Online]. Available: https://doi.org/10.1145/3372885.3373836

[4] B. Loff, N. Moreira, and R. Reis, "The computational power of parsing expression grammars," *Journal of Computer and System Sciences*, 2020.

[5] R. Yuan, Y. Weibing, C. Mingyu, Z. Xiaofang, and F. Jianping, "Robust tcp reassembly with a hardware-based solution for backbone traffic," in *Fifth International Conference on Networking, Architecture, and Storage*, 2010, pp. 439–447.

[6] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 36–43.

[7] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An fpga-based syntactic parser for real-life almost unrestricted context-free grammars," in *Field-Programmable Logic and Applications*, G. Brebner and R. Woods, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 590–594.

[8] J. L. Bordim, Y. Ito, and K. Nakano, "Accelerating the cky parsing using fpgas," *IEICE Transactions on Information and Systems*, vol. 86, no. 5, pp. 803–810, 2003.

[9] S. Taylor, "Protecting embedded systems from zero-day attacks," in *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, 2018, pp. 165–168.

[10] C. Blaudeau and N. Shankar, "A verified packrat parser interpreter for parsing expression grammars," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020, pp. 3–17.

[11] A. Koprowski and H. Binsztok, "Trx: A formally verified parser interpreter," in *European Symposium on Programming*. Springer, 2010, pp. 345–365.

[12] M. van Geest and W. Swierstra, "Generic packet descriptions: Verified parsing and pretty printing of low-level data," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, ser. TyDe 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 30–40. [Online]. Available: https://doi.org/10.1145/3122975.3122979

[13] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1465–1482.

[14] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 615–628. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[15] A. Herrera, A. Bulski, C. Leimbrock, D. Reba, P. Pučil, M. Yakshin, T. Koczka, and S. Mandalas, "Kaitai Struct," http://kaitai.io.

[16] P. Mundkur, L. Briesemeister, N. Shankar, P. Anantharaman, S. Ali, Z. Lucas, and S. Smith, "The Parsley Data Format Definition Language," in *6th Language-Theoretic Security Workshop at IEEE Security and Privacy Symposium*. IEEE, 2020.

[17] R. E. McGrath, "Data Format Description Language: Lessons Learned, Concepts and Experience," University of Illinois, Tech. Rep., 2011.