Simultaneous Productions:
A Fully General Grammar Specification

Danny McClanahan

2021-04-18

1 Motivation for a New Grammar Specification

This paper is the first of several on a parsing method we will refer to as "Simultaneous Productions" (or "S.P." for short). This name was chosen to emphasize two goals of this method:

1. An S.P. *grammar* is composed of a set of *productions*, very similar to most existing concepts of formal grammars [1]. However, unlike many common parsing algorithms [2], an S.P. grammar can represent a recursively enumerable language [3].

2. When parsing a string, these productions can be independently evaluated over separate parts of the input string, and adjacent successful matches can then be merged to form a successful parse. Unlike many common parsing algorithms, this feature avoids any intrinsic serial dependencies that require parsing the beginning of the string first, hence allowing for *parallelism* during the parsing process.

1.1 Goals of This Paper

We have noted that the above two features are not shared by many commonly-used parsing algorithms [4]. In this paper, we will describe the S.P. *grammar-grammar*, i.e. the specification which defines any S.P. grammar. We hope to show that:

- The S.P. grammar-grammar is equivalent to Chomsky's canonical specification of a formal language [5].

[1] cite: chomsky formal grammars

[2] cite: common parsing algorithms – frequency and power? maybe cite history of parsing page?

[3] cite: what is RecEnum?

[4] cite: the history of parsing webpage

[5] cite: chomsky formal grammars again

- An S.P. grammar can *easily and naturally represent* many of the common use cases for parsers. This is a subjective measure of the ease to develop an appropriate grammar for the *grammar-writer*, and can likely be improved upon. The *grammar-writer* is perceived to a be human being attempting to create a grammar to parse some "real-world" input.

- Furthermore, representing a grammar with S.P. **does not introduce any additional complexity** for the grammar-writer over other common grammar specifications, such as those used for regular expressions (DFAs) [6], regex with backrefs (recursively enumerable) [7], as well as EBNF syntax commonly used for CFGs [8].

1.2 Followup Work

Further paper(s) will describe an efficient parsing algorithm to evaluate an S.P. grammar over a specific input string. The intention of this separation is to allow the S.P. grammar-grammar to be reviewed and criticized separately from the evaluation method. This is done because the author believes that the S.P. grammar-grammar has merit in itself, as a "lingua franca" for *executable formal grammars*, that is, grammars which can be efficiently parsed by computer.

1.3 Notation

- Named concepts in this paper will be represented in *italics* when first defined.

- Capital letters generally refer to sets, while lowercase letters generally refer to elements of some set. This may not be true in all cases.

- As an abbreviation, $[n] = [1, n] \, \forall \, n \in \mathbb{N}$, and $\mid$ should be translated as "for some".

2 Definition

We first define the *S.P. grammar-grammar*, as a kind of meta-grammar which specifies all concrete S.P. grammars. We use the term grammar-grammar to emphasize the regular structure of an S.P. grammar. We believe this formulation is relatively simple to analyze, and in subsequent work we will demonstrate that it admits a relatively performant *parsing algorithm*, or *evaluation method*. It is possible this representation can be further improved.

[6] cite: what are DFAs/regex
[7] cite: backrefs are RecEnum
[8] cite: use of EBNF for CFGs

2.1 S.P. Grammar-Grammar

$$SP = (\Sigma, \mathcal{P}). \tag{1}$$

An *S.P. grammar* SP is a 2-tuple with an arbitrary finite set Σ and a finite set of productions $\mathcal{P}$ defined in (equation 2). We refer to Σ as the *alphabet*.

$$p = \mathcal{C}_p \,\forall\, p \in \mathcal{P}. \tag{2}$$

Each *production* p is a finite set of cases $\mathcal{C}_p$, defined in (equation 3).

$$c_p = \{e_j\}_{j=1}^{m} \,\forall\, c_p \in \mathcal{C}_p. \tag{3}$$

Each *case* c_p is a finite sequence of case elements $\{e_j\}_{j=1}^{m}$, where m is the number of elements in the sequence c_p, with e_j defined in (equation 4).

$$e_j = \left\{ \begin{array}{c} t \in \Sigma, \\ p \in \mathcal{P}. \end{array} \right\} \forall\, j \in [m]. \tag{4}$$

Each *case element* e_j is either a *terminal* $t \in \Sigma$ or *nonterminal* $p \in \mathcal{P}$.

3 Parsing

The act of *parsing* given a grammar SP requires introducing a few more concepts. The definition of parsing is completely separated from the definition of a grammar (section 2).

In short, each production $p \in \mathcal{P}$ can be *matched* against some input string I when **any** case $c_p \in \mathcal{C}_p$ matches I. c_p matches I iff **all** terminals and nonterminals are matched against consecutive non-overlapping subsequences of the input string I.

3.1 Input Specification

$$I = \{t_i\}_{i=1}^{n} \,|\, t_i \in \Sigma \,\forall\, i \in [n]. \tag{5}$$

An *input string* I is a finite sequence of *tokens* $\{t_i\}_{i=1}^{n}$ from the alphabet Σ, where $|I| = n$ is the number of elements in the sequence I.

3.2 Partitioning the Input

We would like to be able to reason independently about how different subsequences $\bar{I}$ of the input I may match a certain production $p \in \mathcal{P}$. We eventually make use of this reasoning in (section 3.3), where we describe how to check whether an arbitrary production p matches an arbitrary $\bar{I}$. In later work we hope to show that this enables highly scalable performance gains through caching and parallelism techniques.

3.2.1 Substrings, Bookmarks, and Subsequences

We define two methods to represent a *subsequence* $\bar{I}$ of the input string I: *substrings* $\bar{I}_{(l_1,l_2)}$ and *bookmarks* $\widehat{I}_{(l^+)}$.

$$\bar{I}_{l_1,l_2} = \{t_i\}_{i=l_1}^{l_2} \quad = \text{substring}(I, l_1, l_2) \quad | \quad l_1 \leq l_2 \leq n \in \mathbb{N}. \tag{6}$$

$$\widehat{I}_{(l^+)} = \{\} \quad = \text{bookmark}(I, l^+) \quad | \quad l^+ \in [n+1]. \tag{7}$$

$$\{\bar{I}\} = \{\bar{I}_{l_1,l_2}\} \amalg \{\widehat{I}_{(l^+)}\} \quad = \text{subsequences}(I). \tag{8}$$

The *substring* $\bar{I}_{l_1,l_2}$ is the subsequence of I from indices l_1 to l_2, inclusive. The *bookmark* $\widehat{I}_{(l^+)}$ is an empty sequence (technically an empty subsequence of I) which is inserted **before** the index l^+. In the case that $l^+ = n+1$, the bookmark $\widehat{I}_{(l^+)}$ is considered to be at the **end** of the input string I.

We use the notation $\{\bar{I}\}$ to denote the disjoint union of these two types of *subsequences* of I. Bookmarks are essentially only needed to represent productions which match the empty string (which are perfectly legal): see the matching process in (section 3.3).

3.2.2 Sorting Subsequences

We would like to be able to compare subsequences from separate, possibly-overlapping parts of the string, in order to produce a data structure that looks like a "parse tree", but which can also represent non-local dependencies, such as those found in context-sensitive and recursively enumerable languages [9].

We first establish the "leftmost" and "rightmost" functions, and introduce the concept of "adjacency" for subsequences. We then produce an "adjacency mapping" construct which splits up the input I (section **??**).

$$\begin{array}{rcl} \left\{ \begin{array}{lcllc} \text{leftmost}(\widehat{I}_{(l^+)}) & = & l^+ & \in & [n+1], \\ \text{leftmost}(\bar{I}_{(l_1,l_2)}) & = & l_1 & \in & [n]. \end{array} \right\} & = & \text{leftmost}(\bar{I})\, \forall\, \bar{I} \in \{\bar{I}\}. \\ \text{leftmost}(\bar{I}) & : & \{\bar{I}\} \to [n+1]. \end{array} \tag{9}$$

$$\begin{array}{rcl} \left\{ \begin{array}{lcllc} \text{rightmost}(\widehat{I}_{(l^+)}) & = & l^+ & \in & [n+1], \\ \text{rightmost}(\bar{I}_{(l_1,l_2)}) & = & l_2 & \in & [n]. \end{array} \right\} & = & \text{rightmost}(\bar{I})\, \forall\, \bar{I} \in \{\bar{I}\}. \\ \text{rightmost}(\bar{I}) & : & \{\bar{I}\} \to [n+1]. \end{array} \tag{10}$$

TODO: ???

[9] cite: cite or prove for context-sensitivity requiring non-local deps!

$$\{\text{LookDirection}\} = \{\text{Left}, \text{Right}\}. \tag{11}$$

$$\{\text{Result}\} = \{\text{Success}, \text{Failure}, \text{IDK}\}. \tag{12}$$

$$F_{-}^{+} : (\{\bar{I}\} \times \{\bar{I}\} \times \{\text{Left}, \text{Right}\}) \rightarrow \{\text{Success}, \text{Failure}, \text{IDK}\}. \tag{13}$$

$$F_{-}(\bar{I}, \bar{I}') = F_{-}^{+}(\bar{I}, \bar{I}', \text{Left}). \tag{14}$$

$$F^{+}(\bar{I}, \bar{I}') = F_{-}^{+}(\bar{I}, \bar{I}', \text{Right}). \tag{15}$$

$$F_{-}(\bar{I}, \bar{I}') = \left\{ \begin{array}{lcr} \text{leftmost}(\bar{I}) = \text{leftmost}(\bar{I}') & \Rightarrow & \text{Failure}, \\ \text{leftmost}(\bar{I}) < \text{leftmost}(\bar{I}') & \Rightarrow & \text{Success}, \\ F_{-}(\bar{I}', \bar{I}) = \text{Success} & \Rightarrow & \text{Failure}, \\ \text{otherwise} & \Rightarrow & \text{IDK}. \end{array} \right\} \tag{16}$$

$$F^{+}(\bar{I}, \bar{I}') = \left\{ \begin{array}{lcr} \text{rightmost}(\bar{I}) = \text{rightmost}(\bar{I}') & \Rightarrow & \text{Failure}, \\ \text{rightmost}(\bar{I}) < \text{rightmost}(\bar{I}') & \Rightarrow & \text{Success}, \\ F^{+}(\bar{I}', \bar{I}) = \text{Success} & \Rightarrow & \text{Failure}, \\ \text{otherwise} & \Rightarrow & \text{IDK}. \end{array} \right\} \tag{17}$$

The functions F_{-} and F^{+} provide "less than" and "greater than" operators which can compare any two subsequences of I, but may return an IDK result.

3.3 Matching a Production

We construct the predicate to answer the matching question for a production $\text{matches}_{(\mathcal{P})}$ recursively, by defining the "matches" function over multiple separate domains:

$$\begin{aligned} \text{matches}_{(\mathcal{P})} &: \{SP\} \times \mathcal{P} \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{P})}(SP, p, I) &= \{\, \exists\, p \in \mathcal{P}, c_p \in \mathcal{C}_p \,|\, \text{matches}_{(\mathcal{C}_p)}(c_p, I) \Leftrightarrow \text{true}\}. \end{aligned} \tag{18}$$

A production $p \in \mathcal{P}$ *matches* an input string I when any of its cases $c_p \in \mathcal{C}_p$ match I as defined in (equation 19).

$$\begin{aligned} \text{matches}_{(\mathcal{C}_p)} &: \mathcal{C}_p \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{C}_p)}(c_p, I) &= \{\, \exists\, \bar{I}^{*}_{m} \,|\, \text{matches}_{(e_j)}(e_j, \bar{I}_j)\, \forall\, j \in [m] \Leftrightarrow \text{true}\}. \end{aligned} \tag{19}$$

A case $c_p \in \mathcal{C}_p$ *matches* an input string I when there exists an adjacency mapping $\bar{I}^{*}_{m}$ of length $m = |c_p|$ which maps each case element e_j to a subsequence $\bar{I}_j$ such that every case element matches its assigned subsequence from the adjacency mapping as defined in (equation 20).

$$\begin{aligned} [m] \times \{\bar{I}\} \rightarrow \{\text{true}, \text{false}\} &\quad : \text{matches}_{(e_j)}. \\ \left\{ \begin{array}{lcrcl} e_j = t \in \Sigma & \Rightarrow & \bar{I} = \bar{I}_{l_1, l_2}, l_1 = l_2, I_{l_1} = t & \Leftrightarrow & \text{true}, \\ e_j = p' \in \mathcal{P} & \Rightarrow & \text{matches}_{(\mathcal{P})}(SP, p', \bar{I}) & \Leftrightarrow & \text{true}. \end{array} \right\} &\quad = \text{matches}_{(e_j)}(j, \bar{I}). \end{aligned} \tag{20}$$

A case element e_j matches an input subsequence $\bar{I}$ when e_j is a token $t \in \Sigma$, in which case $\bar{I}$ is a length-1 substring of I containing the single token t, or when e_j is a production p', in which case the subsequence $\bar{I}$ must match the production p' as defined in (equation 18).

3.4 Grammar Specialization

$$p^* \in \mathcal{P}. \tag{21}$$

As described in (section 3.3), an S.P. grammar SP alone is not sufficient information to unambiguously parse a string – a single production must also be specified. Therefore to get an *executable grammar*, we select a single "top" production $p^* \in \mathcal{P}$, corresponding to the *start symbol* found in Chomsky grammars (section 5).

$$SP^* = (\Sigma, \mathcal{P}, p^*). \tag{22}$$

The tuple SP^* formed from the selection of p^* is referred to as a *specialized grammar*. For this reason, we may also refer to a grammar SP (without having chosen any p^* yet) as an *unspecialized grammar*.

3.5 Summary of Adjacency

At this stage, we note a few important points:

1. An adjacency mapping $\bar{I}^*_k$ essentially represents a parse tree **TODO: HOW??? CROSS-SERIAL DEPS???**

2. A production p may match a finite or countably infinite number of subsequences $\bar{I}$ of I, not just one. So, if we say p matches $\bar{I}$ for some case c_p, it may still match other subsequences $\bar{I}'$, either for the same case c_p, or other cases $c'_p \in \mathcal{C}_p$.

3. We have not yet described a method to **actually construct an adjacency mapping $\bar{I}^*$ for a given specialized grammar and input**. That is out of scope for this paper.

4 Proof of Turing-Equivalence

5 Chomsky Equivalence

We have described an S.P. grammar $SP = (\Sigma, \mathcal{P})$ (section 2.1), and we have *specialized* the grammar into $SP^* = (\Sigma, \mathcal{P}, p^*)$ by selecting a production $p \in \mathcal{P}$ (section 3.4). We have described the conditions under which p^* is said to successfully match an input string I consisting of tokens from Σ (section 3.3).

We attempt to directly reduce the canonical specification of a formal grammar (often attributed to Noam Chomsky) into specialized or executable form SP^* [10].

5.1 Chomsky Construct

The definition of a "formal grammar" we copy from Noam Chomsky as follows [11]:

5.1.1 Formal Grammar Definition

$$\begin{aligned}
\Sigma&\text{: terminal symbols.} && (23)\\
N&\text{: nonterminal symbols.} && (24)\\
S&\text{: start symbol.} \in N && (25)\\
P&\text{: productions.} && (26)\\
P&=\{(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*\}. && (27)\\
G&=(N,\Sigma,P,S). && (28)
\end{aligned}$$

5.1.2 Parsing a Formal Grammar

asdf

5.2 Equivalence Proof

We will perform a Cook-Levin-style reduction from a Chomsky grammar $G = (N, \Sigma, P, S)$ (section 5.1) into a specialized S.P grammar $SP^* = (\Sigma, \mathcal{P}, p^*)$ (section 3.4) [12].

5.2.1 Construction of Σ_{SP^*}

$$\Sigma_{SP^*} = \Sigma_G. \tag{29}$$

The alphabet Σ is exactly the same in both S.P. and the Chomsky formulation.

5.2.2 Construction of $\mathcal{P}$

$$asdf \tag{30}$$

6 Relevant Prior Art / Notes

6.1 Overview

asdf

[10] cite: chomsky grammars
[11] cite: chomsky!!!
[12] cite: cook-levin!