

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.

¹cite: chomsky formal grammars

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

³cite: what is RecEnum?

⁴cite: the history of parsing webpage

⁵cite: 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 CFGs⁸.

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 $|$ 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.

⁶cite: what are DFAs/regex

⁷cite: backrefs are RecEnum

⁸cite: use of EBNF for CFGs

2.1 S.P. Grammar-Grammar

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

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*.

$$p = \mathcal{C}_p \forall p \in \mathcal{P}. \quad (2)$$

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

$$c_p = \{e_j\}_{j=1}^m \forall c_p \in \mathcal{C}_p. \quad (3)$$

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).

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

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 non-terminals are matched against consecutive non-overlapping subsequences of the input string I .

3.1 Input Specification

$$I = \{t_i\}_{i=1}^n \mid t_i \in \Sigma \forall i \in [n]. \quad (5)$$

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 want to

In order to model a partitioning of

3.2.1 Substrings, Bookmarks, and Subsequences

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

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

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

The *substring* \bar{I}_{l_1, l_2} is the subsequence of I from indices l_1 to l_2 , inclusive. The *bookmark* \hat{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 \hat{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 .

3.2.2 Adjacent Subsequences

$$\begin{aligned} & \{\bar{I}\} \times \{\bar{I}\} \rightarrow \{\text{true}, \text{false}\} = \text{adjacent}. \\ \left\{ \begin{array}{ll} \bar{I} = \hat{I}_{l^+}, \bar{I}' = \hat{I}_{l'^+} & \Rightarrow \quad l^+ = l'^+ \Leftrightarrow \text{true}, \\ \bar{I} = \hat{I}_{l^+}, \bar{I}' = \bar{I}_{l'_1, l'_2} & \Rightarrow \quad l^+ = l'_1 \Leftrightarrow \text{true}, \\ \bar{I} = \bar{I}_{l_1, l_2}, \bar{I}' = \hat{I}_{l'^+} & \Rightarrow \quad l_2 = l'^+ + 1 \Leftrightarrow \text{true}, \\ \bar{I} = \bar{I}_{l_1, l_2}, \bar{I}' = \bar{I}_{l'_1, l'_2} & \Rightarrow \quad l'_1 = l_2 + 1 \Leftrightarrow \text{true}. \end{array} \right\} = \text{adjacent}(\bar{I}, \bar{I}'). \end{aligned} \quad (9)$$

Two subsequences \bar{I} and \bar{I}' of I are defined to be *adjacent* when $\text{adjacent}(\bar{I}, \bar{I}') = \text{true}$. As shown in (equation 9), a bookmark is adjacent to another bookmark when they occupy the same position within I . A bookmark is adjacent to a substring when it is immediately before or immediately after the substring. Two substrings are adjacent when the end of one substring is immediately before the beginning of the other.

$$\bar{I}_k^* = \{\bar{I}_q\}_{q=1}^k \mid \left\{ \begin{array}{ll} \text{adjacent}(\bar{I}_q, \bar{I}_{q+1}) & = \text{true} \forall q \in [k-1], \\ \bar{I}_1 & = \left\{ \begin{array}{l} \hat{I}_{(l^+=1)}, \text{ or} \\ \bar{I}_{(l_1=1), (l_2 \in [n])}. \end{array} \right\}, \\ \bar{I}_k & = \left\{ \begin{array}{l} \hat{I}_{(l^+=n+1)}, \text{ or} \\ \bar{I}_{(l_1 \in [n]), (l_2=n)}. \end{array} \right\} \end{array} \right\} \quad (10)$$

The *adjacency mapping* \bar{I}^* is a *contiguous* or consecutively adjacent sequence of length k of subsequences $\{\bar{I}_q\}_{q=1}^k$ of the input string I , in which the first element \bar{I}_1 is adjacent to, or contains, the first element t_1 of I , and the final element \bar{I}_k is adjacent to, or contains, the last element t_n of I . A bookmark for \bar{I}_1 or \bar{I}_k would be adjacent to t_1 or t_n , while a substring would contain t_1 or t_n . We say that \bar{I}_k^* *spans* the tokens of I .

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:

$$\begin{aligned} \text{matches}_{(\mathcal{P})} &: \{SP\} \times \mathcal{P} \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{P})}(SP, p, I) &= \{\exists p \in \mathcal{P}, c_p \in \mathcal{C}_p \mid \text{matches}_{(\mathcal{C}_p)}(c_p, I) \Leftrightarrow \text{true}\}. \end{aligned} \quad (11)$$

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 12).

$$\begin{aligned} \text{matches}_{(\mathcal{C}_p)} &: \mathcal{C}_p \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{C}_p)}(c_p, I) &= \{\exists \bar{I}_m^* \mid \text{matches}_{(e_j)}(e_j, \bar{I}_j) \forall j \in [m] \Leftrightarrow \text{true}\}. \end{aligned} \quad (12)$$

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 13).

$$\begin{aligned} &[m] \times \{\bar{I}\} \rightarrow \{\text{true}, \text{false}\} : \text{matches}_{(e_j)}. \\ \left\{ \begin{array}{ll} e_j = t \in \Sigma & \Rightarrow \quad \bar{I} = \bar{I}_{l_1, l_2}, l_1 = l_2, I_{l_1} = t \Leftrightarrow \text{true}, \\ e_j = p' \in \mathcal{P} & \Rightarrow \quad \text{matches}_{(\mathcal{P})}(SP, p', \bar{I}) \Leftrightarrow \text{true}. \end{array} \right\} = \text{matches}_{(e_j)}(j, \bar{I}). \end{aligned} \quad (13)$$

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 11).

3.3.1 Grammar Specialization

$$p^* \in \mathcal{P}. \quad (14)$$

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 4).

$$SP^* = (\Sigma, \mathcal{P}, p^*). \quad (15)$$

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*.

3.4 Implicit Adjacency

At this stage, we note two important points:

1. 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 substrings \bar{I}' , either for the same case c_p , or other cases $c'_p \in \mathcal{C}_p$.
2. 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 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 3.3.1). 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^* ⁹. We provide a graph formulation G from the set of productions P and define parsing in terms of this graph representation (section 4.1).

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

$$\Sigma = \Sigma \tag{16}$$

4.1 Graph Formulation

We will take a moment to conceptualize the act of parsing the input I from the grammar (Σ, P, p) in terms of a graph $G_{P,I}$, abbreviated as simply G . This graph is defined as follows:

- $V(G) = I$, where $V(G)$ is the vertices of G , which correspond to the consecutive tokens of I . This implies that the vertices $V(G)$ are fully ordered, and can be mapped to the integers $i \in [1, n]$, where $n = |I|$.
- The edges $E(G)$ correspond to *adjacent states* from the set of productions P . Adjacency is defined as in (section 3.2.2). **Due to the definition of adjacency, $E(G)$ may be either finite or countably infinite.**
- **A *successful parse* over the graph G is a hamiltonian path H_G over the vertices $V(G)$, in the ordering prescribed by I .**

TODO: describe how the path starts and ends!!!

⁹cite: chomsky grammars

5 Relevant Prior Art / Notes

5.1 Overview

asdf