

# Simultaneous Productions: A Fully General Grammar Specification

Danny McClanahan

2021-06-10

## Contents

<b>1</b>	<b>Motivation for a New Grammar Specification</b>	<b>2</b>
1.1	Goals . . . . .	2
1.2	No Parsing Allowed . . . . .	3
1.3	Notation . . . . .	3
<b>2</b>	<b>Definition</b>	<b>4</b>
2.1	S.P. Grammar-Grammar . . . . .	4
<b>3</b>	<b>PoP: Proof of Parsing</b>	<b>4</b>
3.1	Grammar Specialization . . . . .	4
3.2	Input Specification . . . . .	5
3.3	Matching a String . . . . .	5
3.3.1	Matching a Production $p \in \mathcal{P}$ . . . . .	5
3.3.2	Matching a Case $c_p \in \mathcal{C}_p$ . . . . .	5
3.3.3	Matching a Case Element $e_j \in c_p$ . . . . .	6
3.4	Summary of Adjacency . . . . .	6
<b>4</b>	<b>Proof of Turing-Equivalence</b>	<b>6</b>
<b>5</b>	<b>Chomsky Equivalence</b>	<b>6</b>
5.1	Chomsky Construct . . . . .	6
5.1.1	Formal Grammar Definition . . . . .	7
5.1.2	Parsing a Formal Grammar . . . . .	7
5.2	Equivalence Proof . . . . .	7
5.2.1	Construction of $\Sigma_{SP^*}$ . . . . .	7
5.2.2	Construction of $\mathcal{P}$ . . . . .	7
<b>6</b>	<b>Relevant Prior Art / Notes</b>	<b>7</b>
6.1	Overview . . . . .	7

# 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 <sup>1</sup>. However, unlike many common parsing algorithms <sup>2</sup>, an S.P. grammar can represent a recursively enumerable language <sup>3</sup>.
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 <sup>4</sup>, 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

We have noted that the above features are not shared by many commonly-used parsing algorithms <sup>5</sup>. In this paper, we will describe the S.P. *grammar-grammar*, i.e. the specification which defines any S.P. grammar.

The S.P. model was created to make grammar definitions significantly more composable by **separating the entry point for parsing/string matching from the rest of the grammar’s definition**. The hope is that this can allow for **global agreement upon a safe reliable subset of “primitive” parsers (e.g. integer and float parsers) across codebases**, while end users are still able to add custom logic on top seamlessly.

We hope to show that:

1. The S.P. grammar-grammar should be proven equivalent to Chomsky’s canonical specification of an formal language <sup>6</sup>.
2. The S.P. grammar input format should be able to represent all rungs of the Chomsky hierarchy distinctly to the *grammar-writer*, who may then pick and choose the kind of complexity they want instead of getting surprised. For example:

- regular expressions (DFAs) <sup>7</sup>,

---

<sup>1</sup>cite: chomsky formal grammars

<sup>2</sup>cite: common parsing algorithms – frequency and power? maybe cite history of parsing page?

<sup>3</sup>cite: what is RecEnum?

<sup>4</sup>cite: might need to expand on this more? need to think about what “common” means

<sup>5</sup>cite: the history of parsing webpage

<sup>6</sup>cite: chomsky formal grammars again, maybe just turing machine?

<sup>7</sup>cite: what are DFAs/regex

- EBNF syntax commonly used for CFGs <sup>8</sup>,
  - regex with backrefs (recursively enumerable?) <sup>9</sup>.
3. Furthermore, representing a grammar with S.P. **should follow strongly a “pay for what you use” policy**, i.e. having more complex features should not create needless complexity for users who only use the simpler features. **This requires creating a hierarchical-ish concept of feature complexity.**
  4. Separate the output of a parse from the algorithm that produces it: see (section 1.2). We hope that **standardizing parser output** into the fully general <sup>10</sup> PoP format as well as **sharing high-quality grammars in a modular way** can reduce what we see as a frustrating amount of mental overhead and duplicated work for everyone who wants to write any sort of compiler or interpreter. We believe this situation can be improved.

## 1.2 No Parsing Allowed

**We will not rely on the details of any specific parsing algorithm.** Indeed, we will avoid introducing any iterative algorithms at all in this paper. 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 the rest of 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 abbreviations:
  - $[n] = [1, n] \forall n \in \mathbb{N}$ .
  - $|$  should be translated as “for some” or “for which”.

---

<sup>8</sup>cite: use of EBNF for CFGs

<sup>9</sup>cite: backrefs are RecEnum?

<sup>10</sup>is this true?

## 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.

### 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 <sup>11</sup> set  $\Sigma$  and a finite set of productions  $\mathcal{P}$  defined in (equation 2). We refer to  $\Sigma$  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 PoP: Proof of Parsing

We describe a tree-like data structure “proof of parsing” (PoP) which retains sufficient information to validate that a string belongs to a recursively enumerable language, as well as the validation function “validate” applied to the data structure.

### 3.1 Grammar Specialization

$$p^* \in \mathcal{P}. \quad (5)$$

An S.P. grammar  $SP$  alone is not sufficient information to unambiguously parse a string – a specific production must also be provided (we have defined it to be this way). Therefore to get an *executable grammar*, we select a single “top” production  $p^* \in \mathcal{P}$ . This is vaguely reminiscent of the *start symbol* found in Chomsky grammars (section 5).

---

<sup>11</sup>The finiteness of  $\Sigma$  and  $\mathcal{P}$  is an entirely arbitrary limitation we apply for the purposes of this paper.

**TODO:** this mechanism is the same way we should use for specifying precedence of non-top productions when a parse is completely ambiguous in a subtree!!!

$$SP^* = (\Sigma, \mathcal{P}, p^*). \quad (6)$$

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*.

### 3.2 Input Specification

$$I = \{t_i\}_{i=1}^n \mid t_i \in \Sigma \forall i \in [n]. \quad (7)$$

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

**TODO: we should not distinguish the act of parsing a finite vs infinite stream!**

### 3.3 Matching a String

We represent the conditions necessary to match a production  $p \in \mathcal{P}$  recursively, by defining the “matches” function over multiple separate domains:

1.  $\text{matches}_{(\mathcal{P})}$ : the top-level function that matches against a single production  $p$ .
2.  $\text{matches}_{(\mathcal{C}_p)}$ : matches against a single case  $c_p$  from a production  $p$ .
3.  $\text{matches}_{(e_j)}$ : matches against a single case element  $e_j$  from the case  $c_p$ .

#### 3.3.1 Matching a Production $p \in \mathcal{P}$

$$\begin{aligned} \text{matches}_{(\mathcal{P})} &: \mathcal{P} \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{P})}(p, I) &= \{\exists c_p \in \mathcal{C}_p \mid \text{matches}_{(\mathcal{C}_p)}(c_p, I) \Leftrightarrow \text{true}\}. \end{aligned} \quad (8)$$

A production  $p \in \mathcal{P}$  *matches* an input string  $I$  when **any** of its cases  $c_p \in \mathcal{C}_p$  match  $I$  via  $\text{matches}_{(\mathcal{C}_p)}$  as defined in (equation 9).

#### 3.3.2 Matching a Case $c_p \in \mathcal{C}_p$

$$\begin{aligned} \text{matches}_{(\mathcal{C}_p)} &: \mathcal{C}_p \times \{I\} \rightarrow \{\text{true}, \text{false}\}. \\ \text{matches}_{(\mathcal{C}_p)}(c_p, I) &= \{\text{TODO: ???}\}. \end{aligned} \quad (9)$$

**TODO:**  $>$  = context-sensitive can be modelled as coroutines, compared to the immediate function call of DFA/CFGs?

### 3.3.3 Matching a Case Element $e_j \in c_p$

$$\left\{ \begin{array}{l} e_j = t \in \Sigma \Rightarrow \bar{I} = \bar{I}_{l_1, l_2}, l_1 = l_2, I_{l_1} = t \Leftrightarrow \text{true}, \\ e_j = p' \in \mathcal{P} \Rightarrow \text{matches}_{(\mathcal{P})}(SP, p', \bar{I}) \Leftrightarrow \text{true}. \end{array} \right\} = \text{matches}_{(e_j)}(j, \bar{I}). \quad (10)$$

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 8).

## 3.4 Summary of Adjacency

At this stage, we note a few 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 subsequences  $\bar{I}'$ , either for the same case  $c_p$ , or other cases  $c'_p \in \mathcal{C}_p$ .

## 4 Proof of Turing-Equivalence

**TODO: do reduction from lambda calculus!!**

## 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 3.1). We have described the conditions under which  $p^*$  is said to successfully match an input string  $I$  consisting of tokens from  $\Sigma$  (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  $SP^*$  <sup>12</sup>.

### 5.1 Chomsky Construct

The definition of a “formal grammar” we copy from **TODO: cite!!!** as follows <sup>13</sup>:

---

<sup>12</sup>cite: chomsky grammars

<sup>13</sup>cite: chomsky!!!

### 5.1.1 Formal Grammar Definition

$\Sigma$ : terminal symbols. (11)

$N$ : nonterminal symbols. (12)

$S$ : start symbol.  $\in N$  (13)

$P$ : productions. (14)

$P = \{(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*\}$ . (15)

$G = (N, \Sigma, P, S)$ . (16)

### 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 3.1)<sup>14</sup>.

### 5.2.1 Construction of $\Sigma_{SP^*}$

$$\Sigma_{SP^*} = \Sigma_G. \quad (17)$$

The alphabet  $\Sigma$  is exactly the same in both S.P. and the Chomsky formulation.

### 5.2.2 Construction of $\mathcal{P}$

$$asdf \quad (18)$$

## 6 Relevant Prior Art / Notes

### 6.1 Overview

asdf

---

<sup>14</sup>cite: cook-levin!