# Noise-skipping Earley parsing and in-order tree extraction from shared packed parse forests

Jeremy Dohmann

November 16, 2020

# List of Algorithms

# Contents

# Acknowledgements

I would like to thank Stuart Shieber for serving as my primary supervisors for this thesis as well as Stephen Chong, Sam Gershman and Boaz Barak for serving as thesis readers.

I would also like to thank Terry Patten, Joe Campolongo, and Ceara Chewning from the SFGToolkit team at Charles River Analytics for supporting this work, serving as collaborators on our implementation, and providing feedback along the way.

Thank you to Kyle Deeds and Cameron Martel for their feedback over the years.

# Abstract

In this thesis, I identify 3 shortcomings of state of the art Earley parsing and offer a unified, end-to-end Earley parsing algorithm which remedies these shortcomings. In particular I address the following issues:

1. The Earley algorithm traditionally returns parses without any rank-ordering associated with them. In addition to not ranking the parses during parse time, the data structure used to represent parse results, the shared packed parse forest (SPPF), doesn't have any intrinsic way to extract trees in a particular order. This is a major shortcoming because, for all but the most trivial grammars and inputs, the number of potential parses for an input is intractably large, making the parser effectively useless barring some external mechanism to filter results.

2. The Earley algorithm requires that parses explain contiguous spans of tokens without any unexplained out-of-vocabulary items or tokens interceding. In other words, traditional Earley parsing parses only whole strings of tokens, as opposed to permitting parsing only discontinuous subsequences of the input tokens. This is a shortcoming in applications where the input contains spurious tokens or where the grammar is only intended to describe a subset of the input.

3. The Earley algorithm's run time is proportional to grammar size. Current state of the art methods substantially restrict the size of grammars which can be practically parsed with reasonable memory and time constraints. This is a major shortcoming because it can be difficult to represent sufficiently expressive languages for many applications using anything short of a massive grammar.

The narrative arc of this thesis can be understood as an attempt to redress the three issues above by extending the Earley algorithm to enable parse ranking and quick extraction, make it robust to noise skipping, and make parsing with massive grammars tractable.

In chapter two I address issue 1 by introducing a framework for ranking parses as a function of their intrinsic attributes. I present this approach as an extension of the work done in [41] and [21]. [41] introduces parsing as a form of deductive reasoning on a logical system defined by a context-free grammar, while [21] unifies a number of concepts from deductive parsing (e.g. recognition, assigning probabilities to parses, etc.) as performing calculations on a suitably defined semiring. Thus chapter two introduces a semiring I refer to as the 'attribute semiring' and show how Earley parsing can be used to derive the attributes of the best derivations for Earley states - where "best" is evaluated by applying a utility function $f(\cdot)$ to a state's attributes. I use the semiring formalism to show which types of attributes and utility functions can be used for this purpose.

In chapters three and four, I address the first two issues above by introducing extensions to the standard Earley parser and standard SPPF-construction algorithm to handle skipping tokens in noisy inputs as well innovations to associate each Earley state and SPPF node with the attributes of its best derivation. In chapter 3, I prove that the modified noise-skipping parsing algorithm runs in $O(w \cdot n^2)$ time where $w$ is the number of tokens allowed to be skipped between any explained tokens, also referred to as the 'skip width'. Chapter four details the construction of a novel variant of the SPPF called an ordered-SPPF which, by design, permits us to associate the best derivation attributes with SPPF nodes and makes extracting the best tree from a forest trivially easy. In that chapter, I show that construction of the ordered-SPPF can be done in $O(w^2n^3 \cdot log(wn))$ time

In chapter five, I address issue two, by showing that the ordered SPPF from chapter four can be used to not only efficiently extract the best tree in the forest but also the top $k$ parses ranked by applying the utility

function to their attributes. I show my efficient top-k algorithm can extract the top $k$ parses in $O(kn^2 \cdot log(kn)$ time.

In appendix A, I address the third issue, by showing an algorithm to filter grammars at run time in order to make large-grammar parsing more efficient.

Finally, in appendix B, I introduce an extension that allows the Earley parser to run continuously and return ranked parses in a streamed manner. This extension requires nontrivial modifications to the core logic in order to maintain memory efficiency while not losing any valid parses.

Thus, this thesis details a unified approach permitting the Earley algorithm to parse valid subsequences of noisy inputs while using massive grammars. The system detailed here is also capable of ranking the parses by customized utility functions over user-defined attributes with the constraints governing such functions detailed in chapter 2.

# Chapter 1

# Introduction

## 1.1 Motivation

### 1.1.1 Context-free Languages and Applications

Context-free languages are a formal class of language originally described by Noam Chomsky [14] and subsequently studied in great depth by linguists and computer scientists due to their usefulness in natural language processing and compilers. Parsing natural language is central to approaches in machine translation [4], natural language inference [11] , search engines [42], human computer interactions [34], and many other applications [20].

Though in recent years CFL parsers have ceded some of their supremacy to alternative grammar formalisms and syntactic representations such as dependency parsing [31] and combinatory-categorial grammars [15], many of the most prominent syntactic theories in linguistics are still rooted in context-free languages. This is primarily due to their intuitive form, broad generality, and interesting representational and computational properties which permit their application to problems far outside the world of NLP [33] [3] [36].

Among their formal properties, CFLs have been argued to be sufficiently expressive to represent nearly every syntactic phenomenon of interest in natural language [35] with limited exceptions [40].

### 1.1.2 Statistical Parsing

Avoiding the computational blow-up associated with processing the whole grammar for each parse is an urgent matter in the field of context-free grammars.

One of the most widespread ways of skirting this problem is to statistically annotate the grammar (e.g. using probabilistic context-free grammars [25] or neural networks [10]) so as to enable empirical heuristics which pursue a limited subset of potential incomplete parses at a time. A further potential limitation of statistical parsing is that it typically returns a fixed, finite number of results, pruned for maximum likelihood - while this design is crucial for avoiding the blowup associated with large grammars, it makes it so that statistical parsers never return a representation containing *all* possible parses of a sentence, only *some*.

### 1.1.3 Non-statistical Parsing and Limitations

This thesis concerns itself with enhancing Earley's non-statistical parsing algorithm to amend the three limitations laid out in the abstract. Even though probabilistic grammars and neural parsing techniques have achieved great successes, and in many ways the Earley algorithm has been sidelined by such advances, they are only useful in domains where large, tagged corpora are available. Furthermore, as noted above, non-statistical parsing is the only true solution when the user desires *all* possible parses.

### 1.1.4 Three limitations

As mentioned in the abstract there are three main limitations to Earley parsing that this thesis attempts to address. The first is the issue of sifting through the potentially exponential number parses for an input string. As noted before, this issue is addressed by statistical methods by assigning probabilities to parses and returning the top parse [9]. Other approaches involve non-deterministic beam searches in which the search for valid parses is heuristically guided by probabilities [5]. These can result in throwing out some parses and typically rely on probabilities inferred from treebank data sets.

Though highly performant, these approaches have substantially reduced flexibility when compared to traditional Earley parsing. In theory, any context-free language of interest can be constructed over a user-defined grammar, whether or not there is sufficient data available to use statistical techniques. It would be ideal if practical parsing methods had the flexibility of achieving high performance results on *any* grammar, not just on those which have received the significant resource investment of creating annotated treebanks. Furthermore, statistical reasoning is a somewhat one-dimensional approach, and it would be useful to be able to incorporate many sources of information in determining the rank of a parse, for example, presence of certain words, count of clauses of a certain type, depth of the parse tree, etc.

Thus one of the goals of high performance traditional Earley parsing can be put as follows:

**Goal one: ranking parses with the flexibility of ad-hoc grammars:**
We would like to capture the best qualities of statistical parsing, namely its ability to return optimal, small result sets in the face of highly ambiguous grammars and potentially exponentially many suboptimal parses for a given input.

We also would like to maintain the flexibility of non-statistical parsing by permitting ad-hoc, user constructed grammars, and allow ranking of grammars via highly customizable metrics.

My approach to accomplishing this goal is introducing the ability to associate each Earley state/parse tree/SPPF node with a user-programmed vector of attributes belonging to the best derivation of that state.

Another central shortcoming of standard parsing techniques is that they lack the functionality to leave tokens unexplained. This functionality would be useful, for example, in the context of parsing speech, in which people frequently utter items which are out of the vocabulary, or perform false starts and stutters which should be disregarded in valid parses. Other valid use cases of noise skipping include wanting to parse only a subset of the input. For example, highly-customized, domain-specific information extraction applications may be interested in looking for only some substructures in the data while not being interested in a fully comprehensive syntactic parse of the input. While this sort of quick-and-dirty information extraction is often handled in practice using regular expressions, or some other similarly expressive technique, it would be a tremendous boon to practitioners to enable such sorts of extraction with the expressiveness of a context-free language. In the context just mentioned, by definition we are interested only in parsing subsequences of the data and therefore need the flexibility to relax the expectation that our parser explain *every* token it encounters.

Attempting to address the noisy-parsing problem only enhances the relevance of our first goal, because, for any n-length string there are exponentially many ordered subsequences, greatly compounding the importance of efficiently ranking results.

Some parsing frameworks are better suited to handling noisy input streams than others, for example, dependency parsers are fairly robust to noise and rely less on the contiguity assumptions of standard constituent-based syntactic parsing. That said, I am focused on extending the relevance and robustness of Earley parsing in particular, thus we can state the second goal of high performance Earley parsing as follows:

**Goal two: 'noisy parsing' or, parsing non-contiguous subsequences of the input**
We would like to be able to allow skipped sequences between explained tokens, and between constituent boundaries during parsing.

This includes both permitting a parse to ignore out-of-vocabulary items as well as allowing parse results to skip in-vocabulary items as well.

This approach requires amending the Earley algorithm's core logic relaxing the constraint that tokens states have contiguous spans. Additionally, we must amend the SPPF-construction algorithm of [38] to represent the fact that nodes can have non-overlapping spans.

The final, less substantial shortcoming, which I resolve in the appendix is that Earley's time and space complexity scales with grammar size. The most recent state-of-the-art attempts [8] to enhance Earley performance on very large grammars have become outdated when compared to the increase in grammar size enabled by hardware improvements over the interceding decades. It has be shown that creating grammars encompassing the full extent of natural languages' complexity can result in grammars on the order of millions of rules [32]. In order to extend the usefulness of Earley parsing to make it competitive on state-of-the-art tasks we must devise innovations to make it practical to use such massive grammars.

Thus our final goal is the following:

**Goal three: performant parsing on massive context-free grammars**
We would like to extend state-of-the-art results on practically permissible grammar sizes.

I do so by extending the results of [8] to introduce an efficient method to dynamically filter grammars at parse time - eliminating rules which could not possibly apply to the input, greatly reducing the size of the grammar subset applied to the input and subsequently, the time overhead associated with parsing using large grammars.

### 1.2 Context-free grammars

Context-free grammars (CFGs) are an intuitive formalism used to describe the input space of a context-free language (CFL). We will not define context-free languages in this thesis, but suffice it to say that they are a useful, and very widely studied formal tool to define natural languages, as well as other sophisicated and expressive languages. Other formal classes of language include regular languages (typically represented using regular expressions or finite-state automata) and context-sensitive languages. There are actually many subtypes of CFLs as well, though they are outside the scope of this paper. The interested reader should consult [**ullman**] for more information on formal language theory as well as the formal definition of CFLs, regular languages, etc.

CFGs are a type of language description known as a rewrite system. A rewrite system consists of a symbolic vocabulary describing both the types of input tokens permissible by the language, as well as the types of syntactic structures permissible by the language. The vocabulary consists of two disjoint sets of symbols: terminals and nonterminals. Terminals describe the token literals that constitute the language (e.g. 'dog', 'cat', etc.). Some grammars describe their inputs at a higher level abstraction, using part of speech tags, semantic tags, etc. and call these symbols preterminals (e.g. 'noun', 'verb', etc.). For the purposes of the theory of parsing, and for the purposes of this thesis, preterminals and terminals are interchangeable. Nonterminals describe the constituent-types that exist in the language (e.g. 'noun phrase', 'verb phrase', 'independent clause', etc.). Nonterminals are the structural elements describing the interior nodes of a parse tree. Typically grammars have a reserved symbol, usually represented as $S$, called the sentential form. This form defines what is considered a complete, valid parse. For example, in English, you can parse a constituent to be an NP, VP, etc. but it isn't a complete sentence unless it fits a certain form, has certain elements, etc. Just like there are many ways to complete any other constituent (e.g. a noun phrase can optionally have a determinier, adjectives, etc.) so might a sentential form have many possible forms (e.g. passive sentences, sentences with prepositional modifiers, conjunctions, etc.).

The rewrite system details the structural relations relating terminals and nonterminals and provide the constraints defining what strings are or are not members of the language. The rewrite system is composed of rules containing a single left-hand side (LHS) symbol which must be a nonterminal, as well as a string of right-hand side (RHS) symbols which can be either nonterminals or terminals. The rewrite rule explains what types of elements can be interpreted as the nonterminal LHS symbol. For example:

$$\text{VP} \rightarrow \text{Verb NP NP}$$

This rule could be understood to mean "A verb phrase constituent consists of a verb, followed by a noun phrase, followed by another noun phrase". Typically the same LHS can have many rules associated with it, describing all the different ways to construct that nonterminal. The above rule may describe ditransitive verbs, but another rule may be used to describe intransitive verbs:

$$\text{VP} \rightarrow \text{Verb}$$

### 1.3 Parsing algorithms

When discussing CFLs there are two types of algorithm of interest: recognizers and parsers. Recognizers simply solve the decision problem of determining whether an input is a member of the language and whether it is not, while parsers are able to return a structural description of the input known as a parse tree, if the input is a member of the language. An example grammar describing active-voice sentences with transitive verbs is shown below, followed by a simple parse tree is shown below:

**Transitive verb grammar**

$$S := NP\ VP \tag{1.1}$$
$$NP := \text{`}dogs' \tag{1.2}$$
$$NP := \text{`}cats' \tag{1.3}$$
$$VP := V\ NP \tag{1.4}$$
$$V := \text{`}eat' \tag{1.5}$$

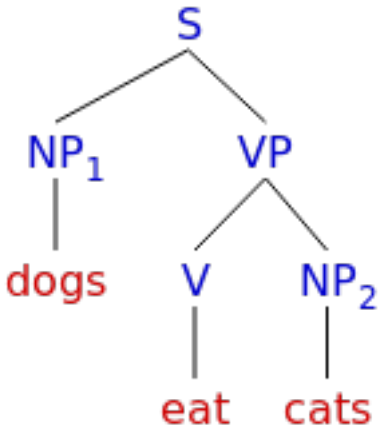



Figure 1.1: A sample parse tree for the input 'dogs eat cats' using the grammar above.

The first polynomial-time algorithm for recognizing whether a string is a member of a general context-free grammar was introduced by Jay Earley [18]. Though there were earlier algorithms capable of parsing restricted subtypes of context-free language, such as the LR parser [27], Earley's was the first successful attempt for context-free grammars containing ambiguities. Technically speaking, Earley merely introduced a recognizer. He attempted to provide an extension enabling his recognizer to return parse trees for correctly recognized inputs, but it was shown by Tomita [43] that his algorithm returned spurious results, and it wasn't until 40 years later that [38] provided a provably correct algorithm for returning *all* parses of the input without returning any spurious results.

While there exist other context-free parsing algorithms such as as the CYK algorithm [44] and the generalized left-right (GLR) parser [43], the Earley algorithm has retained popularity due to its efficiency for practical applications, the simplicity of the algorithm, and its flexibility. In particular it holds a great advantage because it avoids the GLR parser's reliance on pre-computed tables, which preclude filtering techniques such as those in [8] and [17], discussed in the appendix of this thesis.

The Earley parsing algorithm works by iterating through the tokens one by one, manipulating data structures called **states** along the way. States can be described as hypotheses about how to interpret certain spans of the input. A state corresponds to an instantiation of a rule in the RWS. It contains a LHS symbol, corresponding to the LHS of the rule it is instantiating, an array of RHS symbols of the rule, a dot '@' indicating how much of the rule has been applied instantiated thus far, and a span indicating what span of tokens the state is describing. I will represent states as follows $(LHS, RHS, [start\_index, end\_index])$ where indexing for the string starts at 1 and the end_index of a state is an exclusive bound. Thus a state spanning the entire n-length input would have bounds [1, n+1].

For example, while parsing 'dogs eat cats' using the grammar above, at the point that the first two words have been processed, the following states may be existence: $(S, [NP, @, VP], [1, 2])$, $(NP, [dogs, @], [1, 2])$, $(VP, [eats, @, NP], [2, 3])$, $(eats, [eats, @], [2, 3])$.

Taking just the first example, $(S, [NP, @, VP], [1, 2])$ can be understood as signifying that we have partially parsed a sentential form beginning at index 1 and ending at index. We have done so by using rule

1.1: ($S \rightarrow NP\ VP$). So far we have seen a complete NP, but we are still awaiting a completed VP. The state $(eats, [eats, @], [2, 3])$ is a special kind of state signifiying that we have read in a terminal symbol at index 2. There is no rule associated with this state.

In chapter two I will present Earley parsing as a form of deductive logic in which states whose LHS are a nonterminal are the theorems of the logical system, states whose LHS's are terminals are the axioms, and rules in the RWS are inference rules.

### 1.4 Parse forests

As mentioned earlier, general context-free grammars can be highly ambiguous, and as a result any particular input can have many potential parses. In fact, the number of parses can be exponential in the length of the input. In chapter five, I give an example of an expression grammar which has exponentially many parses per input. A polynomial parsing algorithm would be meaningless without the ability to store an exponentially large number of parses compactly.

[43] first introduced the shared packed parse forest (SPPF), a special data structure designed to compactly store parses. SPPFs are a dense graph structure containing a node for each state produced during the Earley parse. Each node corresponds to either a completed constituent and accompanying span, e.g. $(S, 1, 4)$, a scanned token, e.g. $(cats, 3, 4)$, or a partially completed rule, e.g. $(S \rightarrow NP@VP, 1, 2)$ [1]. A constituent node will correspond to multiple Earley states (one for each rule used to parse the span as the nonterminal - of which there may be many for highly ambiguous grammars). A scanned token node corresponds to the single unique state representing reading that terminal at the index given. Finally, the partially completed rule nodes each correspond to one unique Earley state.

Nodes are connected to each other via an intermediate data structure called a family. A constituent node will have one family for each rule deriving it. If there is more than one element before the dot in the RHS of the rule, the family consists of two nodes, one representing the element preceding the dot and another representing the rule node which results from moving the dot back one index. If there is only one element preceding the dot, the family consists solely of a node representing the element before the dot.

For example, $(S, 1, 4)$ would have an edge to a single family consisting of a pointer to the node $(VP, 2, 4)$ and the node $(S \rightarrow NP@VP, 1, 2)$.

Partially completed rule nodes may also have child families. For example, the $(S \rightarrow NP@VP, 1, 2)$ would have a single family consisting only of the single node $(NP, 1, 2)$.

Since the number of nodes is upper bounded by the number of states, and the Earley algorithm is $O(n^3)$ we are guaranteed that the SPPF is cubic in the size of the input.

As [38] showed, it is possible to correctly construct an SPPF from the Earley states by carefully constructing labeled edges from state to state during the parse. In chapter 3, I will explain how those edges are constructed in both the original implementation and my noise-skipping implementation. In chapter 4, I will show the original construction of the SPPF from [38] and discuss how to extend it to handle noise-skipping, as well as introduce an ordered SPPF data structure which uses attributes and utility functions to extract trees in order.

[1] The notation used for SPPF nodes differs from that for states. While a state would be represented as $(S, [NP, @, VP], [1, 2])$ the SPPF node is usually denoted as $(S \rightarrow NP@VP, 1, 2)$

# Chapter 2

# A novel attribute-vector semiring for ranking parses by utility

## 2.1 Abstract

Parsing is one of the most fundamental tasks in formal language theory. The first algorithm for parsing general context free languages was developed by Jay Earley in 1968 [1]. Though much of the field has been concerned with statistical methods in parsing, there is still a case to be made that developments in strictly symbolic parsing are vital to computational linguistics and other related tasks in applied computation. Many data processing tasks have sufficiently sophisticated structure to warrant CFG parsing, but they often lack the data necessary to rely solely on statistical parsing methods. Additionally, applied computation tasks almost always involve imputing structure upon highly noisy data.

This paper focuses on greedily obtaining best parses for noise-skipping Earley parsers. This task is vital in circumstances in which the input data stream is noisy and corpora are not available to guide statistical parsing algorithms, but the user still has a sense of what constitutes a "good parse". I will explore under what circumstances we can greedily compute the best parses for a constituent B using the utilities and attributes of the best parses of its sub-constituents. I will also show that the validity of this approach is sensitive to the data structure underlying the Earley chart and show that priority queues ranked in decreasing order of start index for a state can ensure correct greedy calculations of the optimal derivations as long as the grammar contains no cycles of unary productions (e.g. $A \to B$ and $B \to A$)[1]. I do so by framing Earley parsing as a form of semiring parsing and deductive logic. Two important approaches pioneered by [21] and [41].

## 2.2 Definitions

**Definition: 1** Define a constituent $[A, i, j]$ to be a span of tokens $i, j$ that can be parsed as nonterminal $A$ defined relative to some grammar $G$.
**Definition: 2** Define an inference rule to be a rule stating that if some stipulated theorems $A_1, A_2, ..., A_k$ (the antecedents) are true and some side conditions are satisfied, then the consequent B is also true. A series of applications of inference rules beginning at axioms and ending in consequent B constitutes a proof of B.
**Definition: 3** Call the system of attributes $\mathbb{X}$ a set of $N$ real-valued attributes associated with a particular proof system (see section 3 for concrete details).
**Definition: 4** Associate with each application of rule $R$ an attribute vector $X_R \in \mathbb{R}^N$ drawn from $\mathbb{X}$ representing the attributes associated with applying this particular rule. For example, the log probability of this rule, the number of antecedents in the rule, the number of axioms, etc.
**Definition: 5** Associate with each proof $P$ of $B$ an attribute vector $X_{P_B} \in \mathbb{R}^N$. We will also use the notation $X_B$ when it is clear which proof of $B$ we are discussing.
**Definition: 6** Define the utility function $f(\cdot) : \mathbb{R}^N \mapsto \mathbb{R}$ to be a function mapping a vector $X \in \mathbb{R}^N$ to a single real value.
**Definition: 7** The utility $\beta(B)$ of $B$ is the value **max** $f(X_B)$ taken over all possible proofs of $B$.
**Definition: 8** The attribute vector $\hat{X}_B^R$ of the best proof of $B$ using rule $R$ is $\hat{X}_B^R =$ **argmax** $f(X_B)$ taken over all proofs of $B$ using rule $R$.
**Definition: 9** The attribute vector $\hat{X}_B$ of the best proof of $B$ is $\hat{X}_B =$ **argmax** $f(X_B)$ taken over all possible proofs of $B$.
**Definition: 10** The best proof of $B$, $\hat{P}_B$ is the (not necessarily unique) proof whose attributes are $\hat{X}_B$.
**Definition: 11** Define the pooling function $g(\cdot) : \mathbb{R}^* \mapsto \mathbb{R}^N$ to be a variadic function which takes as arguments

- The attribute vectors of the best proofs of antecedents $\hat{X}_{A_1}, ...$
- The attribute vector $X_R$ of a given application of rule R,

and produces as output the attribute vector, $\hat{X}_B^R$, of the best proof of $B$ using rule $R$

## 2.3 Introduction

The Earley parser is a parsing technique for both recognizing string membership in a (potentially ambiguous) context-free language (CFL) and for describing the possible parses (derivations) of that string under the rules of the rewrite system (RWS) description of that CFL.

[1] This requirement will be used repeatedly throughout this thesis.

[41] provides a unified framework to view the Earley parser, and general parsing algorithms as proofs in deductive logic.

The general form of an inference rule in this framework is:

$$\frac{A_1 \ ... \ A_k}{B} \langle \text{side conditions} \rangle$$

Where $A_i$ are antecedents and $B$ is the consequent, which can be $deduced$ to be true whenever its antecedents are all true and the side conditions for the inference rule are met.

To quote [41]:

> Given a grammatical deduction system, a derivation of a formula B from assumptions $A_1, ..., A_m$ is, as usual, a sequence of formulas $S_1, ..., S_n$ such that $B = S_n$ and for each $S_i$, either $S_i$ is one of the $A_j$ , or $S_i$ is an instance of an axiom, or there is a rule of inference $R$ and formulas $S_{i_1}, ...S_{i_k}$ with $i_1, ...i_k < i$ such that for appropriate substitutions of terms for the metavariables in $R$, $S_{i_1}, ..., S_{i_k}$ match the antecedents of the rule, $S_i$ matches the consequent, and the rule's side conditions are satisfied.

This definition corresponds to the usual definition of a *proof* in deductive logic, and, as shown in [41] corresponds directly to the notion of a parse tree in syntax. Thus in [41]'s framework, theorems correspond to the syntactic statement "span i,j can be parsed to nonterminal A" and proofs correspond to parse trees showing the derivation of the constituent [A,i,j].

A derivation of $B$ may be ambiguous for context-free grammars. Ambiguity arises when multiple inference rules can be used to derive B or one or more of its antecedents. Analogously, in the syntax tree parlance, ambiguity corresponds to a constituent being derivable via multiple chains of RWS rules.

In general, inference rules will take the form

$$\frac{[B, i, j] \ [C, j, k]}{[A, i, k]} \text{A} \rightarrow BC$$

Where the items of the deductive system are constitutents (i.e. span i, j can be parsed as a $B$) and the side conditions are the existence of rules in the grammar permitting the derivation, as well as constraints on the indices of the constituents.

Clearly if there are $r$ possible derivations of $[B, i, j]$ and $s$ possible derivations of $[C, j, k]$ then there are $r \times s$ derivations of $[A, i, k]$[2].

This thesis generalizes its applicability to 'noisy' parsing, where spans of tokens may be skipped in the derivation. This relaxes the side conditions of a derivation such that antecedents can have non-contiguous spans. This admits inferences of the form:

$$\frac{[B, i, j'] \ [C, j, k]}{[A, i, k]} \text{A} \rightarrow BC \wedge j' \leq j$$

Thus for example, we could merge constituents $[B, 0, 2]$ and $[C, 5, 8]$ to produce $[A, 0, 8]$, skipping 3 tokens in the process.

### 2.4 Optimality

As stated previously, the theorem $[A, i, k]$ corresponds to stating that there exists at least one parse tree of root type $A$ which parses span $i, k$. Since there may be many such parse trees we will index them as $P^j_{[A,i,k]}$. In deductive logic terms, each tree corresponds to a unique proof of the theorem $[A, i, k]$.

As defined in section 1, a parse tree (alternatively, proof) $P$ has a vector of attributes $X_P$. Since our goal is finding optimal parses, we will assume that those attributes map onto some concept that one might want to optimize during parsing, e.g. number of constituents (a.k.a. number of inference rules used), number of tokens explained (i.e. the sum of the spans for all items derived), number of tokens skipped, etc.

Following the definitions in section 1, a particular derivation rule $R$:

[2]Assuming that $A \rightarrow B \ C$ is the only rule deriving $[A, i, k]$ and there are no other contiguous sets of indices in $i, k$ for which $B$ and $C$ can be derived.

$$R \frac{[B, i, j]\ [C, j, k]}{[A, i, k]} \text{A} \rightarrow BC$$

results in a parse tree $P_{[A,i,k]}$ whose best attributes $\hat{X}^R_{[A,i,k]}$ can be calculated via applying the pooling function to the attributes of the antecedents and the inherent attributions of this rule application, $\hat{X}^R_{[A,i,k]} = g(\hat{X}_{[B,i,j]}, \hat{X}_{[C,j,k]}, X_R)$. This definition of the pooling function means that the attributes for a parse are compositional in its subtrees. The utility function $f(X_P)$ performs a ranking over all $P^j$, and as noted, we will refer to a (not necessarily unique) best parse as $\hat{P}_{[A,i,k]}$.

This chapter focuses on characterizing the systems of $g(\cdot)$ and $f(\cdot)$ which satisfy the **optimality condition**:

**Definition 1.** $f(\cdot)$, $g(\cdot)$, and the system of attributes $\mathbb{X}$ satisfy the optimality condition if and only if for all derivations of $[B, i, k]$ using rule $R$, having antecedents $[A_1, i, j_1], ..., [A_k, j_k, k]$, we can calculate the optimal parse $\hat{P}_{[A,i,k]}$, its attributes $\hat{X}^R_{[A,i,k]}$, and its utility $f(\hat{X}^R_{[A,i,k]})$, using a single application of $g(\hat{X}_{[A_1,i,j_1]}, ..., X_R$.

This problem statement is particularly attractive for Earley parsing. Since ambiguous CFGs may result in exponentially many parses for $[S, i, j]$, it would be ideal to be able to extract the best possible parse of span $i, j$ without explicitly extracting all $O(2^{|j-i|})$ parses and then ranking them post-hoc. This functionality is especially important in noise-skipping parsing, because the potential for many partial solutions to proliferate can inundate the system with parses of radically different utility values.

I will show in this paper that if you have chosen an $f$, $g$, and $\mathbb{X}$ satisfying the optimality condition, you can always determine the utility of the best parse of a constituent $[A, i, k]$ with only a constant amount of additional overhead. In chapter five I will show that you can also extract the best parse from a parse forest in constant time, and extract the $k$ best parses in $poly(k)$ time.

### 2.5 Optimality of linear utility functions

We will restrict our attention to the pooling function $g$ which simply performs vector addition over its inputs, for many reasonable systems of attributes $\mathbb{X}$ (e.g. those including log probabilities, tokens counted, tokens skipped, total, constituents, etc) this is a perfectly natural pooling function.

**Theorem 1.** *The sum pooling function $g(\cdot)$ and any function $f : \mathbb{R}^N \mapsto \mathbb{R}$ which is a linear transformation satisfies optimality.*

*Proof.* Assume constituent $B$ was derived via the following inference rule:

$$R \frac{A_1, ..., A_k}{B}$$

Optimality is satisfied if the best parse of $B$ is obtained by choosing the best parses of all the $A_i$.

We can write the attributes of the proof of $B$ achieved in this manner as $\hat{X}_B = g(\hat{X}_{A_1}, ..., X_R) = X_R + \sum \hat{X}_{A_i}$, since $g$ is the sum pooling function. Clearly $f(\hat{X}_B) = f(X_R) + \sum f(\hat{X}_{A_i})$ by the linearity of $f$. This is the best value obtainable for any proof of $B$ using this inference rule because changing any of the proofs for any of the $A_i$ would result in a smaller contribution to the sum.

More formally, switching the proof of $A_j$ from the optimal proof $\hat{X}_{A_j}$ to a suboptimal proof $\tilde{X}_{A_j}$, such that $f(\hat{X}_{A_j}) > f(\tilde{X}_{A_j})$ would yield

$$\begin{aligned} f(\tilde{X}_B) &= f(X_R) + f(\tilde{X}_{A_j}) + \sum_{i \neq j} f(\hat{X}_{A_i}) \\ &\leq f(X_r) + \sum f(\hat{X}_{A_i}) \\ &= f(\hat{X}_B) \end{aligned}$$

Where the inequality on the second line is due to the fact that $f(\hat{X}_{A_j}) > f(\tilde{X}_{A_j})$ and the equalities on the first and third lines are due to the linearity of $f$. □

Thus as long as we use a linear utility function and the sum pooling function, we can achieve optimality with no additional constraints imposed on our attributes.

### 2.6 Semiring parsing

[41]'s work on parsing as deductive reasoning was generalized in [21] to show that many useful values for derivations in deductive parsing can be calculate via semirings. A semiring $\langle K, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ is a set K over which there is defined an addition operation $\oplus$, multiplication operation $\otimes$ with identities **0** and **1** respectively, such that addition is associative and multiplication distributes over addition.

[21] showed that a number of important values relevant to parsing in the deductive framework of [41] can be abstracted into a single algorithm which calculates 'values' for each theorem in the semiring. For instance, recognizing string membership in a CFL can be treated as parsing over the boolean semiring $\langle \{TRUE, FALSE\}, \vee, \wedge, FALSE, TRUE \rangle$. He presents a number of other useful values as semirings, such as string probability: $\langle \mathbb{R}, +, \times, 0, 1 \rangle$.

For an arbitrary semiring, the semiring 'value' of a theorem $v$

$$\beta(B) = \oplus_{R \in I(B)} (k_R \otimes (\otimes_{A_i \in T(R)} \beta(A_i)) \tag{2.1}$$

Where $R$ is an inference rule deriving $B$ in the set of inference rules $I(B)$ deriving $B$ , $A_i$ is an antecedent in rule $R$, and $k_R$ is the value associated with this particular derivation.

For the string probability semiring, $k_R$ would be the probability of the inference rule $R$. For the boolean semiring $k_R$ would be TRUE if there exists such an inference rule $R$ and its side conditions are met, and FALSE otherwise.

[28] gives an excellent presentation of Goodman's parsing algorithm over semirings and extends it to include the expectation semiring, the variance semiring, and a further generalization to gradients. The interested reader should refer to both works cited above to understand the motivation behind semiring as a mathematized extension of deductive parsing.

We can present utility value calculation with the sum pooling function, as described in the previous section, as semiring parsing over the 'attribute-vector semiring': $\langle \mathbb{R}^N, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$.

'Values' in the attribute-vector semiring are an attribute vector drawn from the attribute system $\mathbb{X}$. In the attribute-vector semiring, $k_R$ from equation 1, represents the attribute vector, $X_R$, of the application of rule $R$, as defined in definition X. The addition operator corresponds to the function $argmax_{x \in \{a,b\}} f(x)$. Put formally:

$$\begin{aligned} \mathbf{a} \oplus \mathbf{b} &\triangleq argmax_{\mathbf{x} \in \{\mathbf{a},\mathbf{b}\}} f(\mathbf{x}) \\ \mathbf{0} &\triangleq \theta_{inf} \\ s.t.\ \theta_{inf} &\in \mathbb{R}^N \text{is the infimum of } f(\cdot) \end{aligned}$$

Finally, multiplication corresponds to application of $g$. Formally:

$$\begin{aligned} \mathbf{a} \otimes \mathbf{b} &\triangleq g(\mathbf{a}, \mathbf{b}) \\ \mathbf{1} &\triangleq \theta_0 \\ s.t.\ \theta_0 &\in \mathbb{R}^N \text{is the identity vector for the function g} \end{aligned}$$

Using the definitions of $\oplus$ and $\otimes$ above, we can reframe the optimality criterion in terms of semiring parsing. In particular, referring to equation 1, we see that the multiplication inside the summation calculates the attribute vector for each derivation of $B$ and the summation over all parses picks out the best such derivation.

The validity of this formulation depends on whether the definitions above constitute a valid semiring. Thus for any choice of functions $f, g$ we can check their validity by checking whether the operations above constitute a semiring. We will use this constraint to give us deeper insight into which choices of $f, g$ make utility scores greedily computable.

In order to constitute a valid semiring, **0** and **1** must be identities for their respective operations, and $\otimes$ must distribute over $\oplus$. There exists an additive identity **0** only if there is an infimum of $f$, additionally **1** is a multiplicative identity when $g$ is a function having an identity vector.

I will now derive another useful constraint on $f, g$ by using the requirement that $\otimes$ distributes over $\oplus$.

### 2.6.1 Distributivity of $\otimes$

$$\mathbf{a}_1 \otimes (\mathbf{a}_2 \oplus \mathbf{a}_3) \tag{2.2}$$
$$= g(\mathbf{a}_1, argmax_{\mathbf{x} \in \mathbf{a}_2, \mathbf{a}_3} f(x)) \tag{2.3}$$

By distributivity this must be equivalent to

$$(\mathbf{a}_1 \oplus \mathbf{a}_2) \otimes (\mathbf{a}_1 \oplus \mathbf{a}_3) \tag{2.4}$$
$$= argmax_{\mathbf{x} \in \{g(\mathbf{a}_1, \mathbf{a}_2), g(\mathbf{a}_1, \mathbf{a}_3)\}} f(x) \tag{2.5}$$

In other words, distributivity is preserved only when, for all $a_1, a_2, a_3$
$f(g(\mathbf{a}_1, \mathbf{a}_2)) > f(g(\mathbf{a}_1, \mathbf{a}_3)) \iff f(\mathbf{a}_2) > f(\mathbf{a}_3)$.

If you choose $g$ to be the sum or mean pooling functions, some valid semirings include $f$ where $f$ is a linear transformation. In fact, $f$ can be a linear transformation followed by any strictly increasing non-linearity, such as sigmoid or tanh - note that, in this case the attribute space must be restricted so that $f$ has an infimum over it. This discovery paves the way for creating learned utility functions trained via stochastic gradient descent. On the contrary, if you use the max or min pooling functions, there is no $f$ yielding a valid semiring.

Note that if the semiring is valid we know that the optimality criterion holds, but we don't know whether the optimality criterion *only* when the semiring is valid. The reason for our uncertainty is Goodman merely showed that values expressible as a semiring in his formalism can be calculated efficiently, but he did not show that *only* values expressible as a semiring were efficiently computable.

## 2.7 Preventing out-of-order processing

So far, I have been silent on the computational requirements to ensure that our utility function calculations are actually valid. The most crucial thing to note in equation 1, is that when calculating the utility $\beta(B)$ we rely on already knowing the utility of its antecedents $\beta(A_i)$.

Put in terms of inference rules, this means that if we have a derivation such as the one below:

$$\frac{[B, i, j]\ [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC$$

We must ensure that the utilities $\beta([B, i, j])$, $\beta([C, j, k])$ used to derive the utility $\beta([A, i, k])$ are the true best utilities for constituents $[B, i, j], [C, j, k]$ respectively. In computational terms, this requirement stipulates that we never derive $[A, i, k]$ using some estimate $\hat{\beta}([B, i, j])$, etc. of the utilities of its antecedents, and then rederive $[B, i, j]$ yielding a new, better utility $\beta([B, i, j]) > \hat{\beta}([B, i, j])$.

I refer to this phenomenon as **out-of-order processing** because it occurs whenever $[A, i, k]$ is processed before all derivations of its antecedents have been processed. This issue comes up because our algorithm for calculating utility scores is greedy: the utility value of $[A, i, k]$ as derived by inference rule R is set at the time it is derived. Meaning, later derivations of its antecedents don't update the resulting $\beta([A, i, j])$. Thus, this greedy algorithm fails whenever **out-of-order-processing** is possible, even if $f, g$ satisfy the optimality condition.

Framed in terms of Earley parsing, there are four distinct way an Earley state $[A, i, k]$ can be derived from uniqiue inference states:

(1) $R_1\ \dfrac{[B, i, j]\ [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC$

(2) $R_1\ \dfrac{[B, i, j]\ [C, j', k]}{[A, i, k]}\ \mathrm{A} \rightarrow BC \wedge j' \neq j$

(3) $R_2\ \dfrac{[D, i, j]\ [C, j, k]}{[A, i, k]}\ \mathrm{A} \rightarrow DC$

(4) $R_3 \dfrac{[D, i, j]\ [E, j, k]}{[A, i, k]}$ A $\rightarrow DE$

That is, $[A, i, k]$ can be generated by two identical rules with different amounts of noise skipped (1 vs. 2), or via different rules consisting of the same last element (compare 1 and 3), or via completely different rules (compare 1 and 4).

An Earley state stores the start index, the end index, and the rule from which it was derived. Thus unique states would be generated for examples 1, 3, and 4 above, while 1 and 2 would be considered identical states.

The conflict resolution scheme used in this work is that when a unique Earley state has been generated twice, we determine which derivation had the highest utility and set its attributes to those belonging to that derivation. Thus, when using a utility function penalizing skipped tokens, we would save the attributes from derivation 1 over those from 2.

Although the identity of each unique $[A, i, k]$ state is tied to the rule used to generate it, each of those states is agnostic of the rules used to generate its antecedents. Otherwise each state itself would be an entire derivation tree, obviating the polynomial sized storage of the shared-packed parse forest representation.

This erasure of antecedents' derivation history is where trouble with **out-of-of order processing** can arise. Consider what happens if we generate three states and time $t$:

- $[A, i, k],\ A \rightarrow BC$, utility = 10
- $[A, i, k],\ A \rightarrow DC$, utility = 11
- $[A, i, k],\ A \rightarrow DE$, utility = 12

Imagine we then rederive $[C, j, k]$ at a later time $t' > t$ resulting in a new, best-derivation, that if incorporated into states 1-3 above would result in the following new states:

- $[A, i, k],\ A \rightarrow BC$, utility = 13
- $[A, i, k],\ A \rightarrow DC$, utility = 14
- $[A, i, k],\ A \rightarrow DE$, utility = 12

When rederiving $[C, j, k]$ we don't notify all downstream states in the SPPF that their derivations need to be updated - if we did we would asymptotically increase the run time of the parsing algorithm. Thus our Earley states for $[A, i, k]$ would never get updated and we would still believe that

$[A, i, k], A \rightarrow DE$, utility = 12 is its best derivation.

This problem results in a state (uniquely determined by its LHS, RHS, and span) having an incorrect utility score associated with it whenever it is derived for the last time before one (or more of its antecedents) receives its final max utility value (and attributes).

Since the Earley chart is backed by a list of data structures $ADT_k$ indexed by the end index of the states contained in them states are processed in order of their ending index. Thus we can restrict definition of **out-of-order processing** to refer only to when an item's rightmost antecedent is rederived (with a higher utility score) after the consequent state has been derived for the last time. Note that this conclusion is invalid if we have unary cycles and empty productions, thus we will ignore such grammars and assume such constructs have been removed by reducing grammars to Chomsky-normal Form.

Thus our working problem statement is:

**Out-of-order processing** occurs when a state
($[A, i, k], A \rightarrow BC$, utility = u, attributes = $X$) is created by a rule:
and then at some later time, $[C, j', k]$ is processed again via a unique derivation to yield a state $[C, j', k]$ with new utility value such that rederiving:
ought to yield
($[A, i, k], A \rightarrow BC$, utility = $u'$, attributes = $X$ where $u' > u$

### 2.7.1 Impact of data structure on out-of-order processing

Whether out-of-order processing occurs depends on what kind of data structure we use for each entry $L_i$ in the Earley chart.

As mentioned, Earley states are stored in abstract data structures $ADT_k$ indexed by their end index and each index $k \in [0, |input|]$ is processed sequentially until all states in $ADT_i$ have been exhausted.

The factor determining whether out-of-order processing occurs is the order in which states are removed from the each $ADT_k$. I will now show that using a stack or queue produces out-of-order processing, while it can be avoided by using a prioritiy queue sorted in order of decreasing start index, with ties broken by a special topological sort over a small subset graph of the grammar.

### 2.7.2 Stack ADTs

Consider the simple grammar:

**Grammar 1**

$$S := DC \tag{2.6}$$
$$C := AB \tag{2.7}$$
$$A := a \tag{2.8}$$
$$D := d \tag{2.9}$$
$$B := bbb \tag{2.10}$$
$$B := bb \tag{2.11}$$

Applied to input "dabbb" there is one parse with no skipping (using rules 2.6-2.10), and two with a single token skipped (using rules 2.6-2.9 and 2.11).

Crucially, both parses use the following inference rule to complete C:

$$\frac{[A, 1, 2]\ [B, 2, 5]}{[C, 1, 5]}\ \mathrm{C} \rightarrow AB$$

Let's consider what happens when we process $ADT_k$ when using a stack.

On the eve of processing $ADT_5$, it would like look something like: $[(b, 4, 5)]$.

After popping $(b, 4, 5)$ and then pushing states completed by that scan, we would have the following stack:

$$(B, [b, b, @], [2, 5])$$
$$(B, [b, @, b], [2, 5])$$
$$(B, [b, b, b, @], [2, 5])$$
$$(B, [b, b, @], [3, 5])$$

We would then pop the top state (which has a single token skipped) and use it to complete C using CFG rule 2.7 and then push the completed state back onto the stack, resulting in the same stack as above but with the top element replaced with $(C, [A, B, @], [1, 5])$, which has a single token skipped.

We would then pop the top state again, completing S (with a single token skipped via rule 2.6), replacing $(C, [A, B, @], [1, 5])$ with $(S, [C, D], [0, 5])$. Next we pop S and store it as a completed sentential form. We then pop the next state, which would fail to scan and be discarded.

The resulting stack at this point is

$(B, [b, b, b, @], [2, 5])$
$(B, [b, b, @], [3, 5])$

The issue here, which is unique to the noise-skipping context, is that when a state is derived which has an identical production and span as one that already exists, it doesn't get added back to the ADT to be processed again. Therefore when we pop $(B, [b, b, b, @], [2, 5])$ we would re-complete $(C, [A, B, @], [1, 5])$ and update its attribute values, but it doesn't get pushed to the stack again to rederive $(S, [D, C, @], [0, 5])$, leaving $(S, [D, C, @], [0, 5])$ with permanently incorrect tokens-skipped value of 1.

We would next pop $(B, [b, b, @], [3, 5])$ and derive
$(C, [A, B, @], [1, 5])$ a third time, once again, not pushing it back to the stack to update the derivation of $(S, [D, C, @], [0, 5])$ leaving it with an incorrect best possible parse which skips one token.

#### 2.7.3 Priority queue ADTs

The proof of validity for certain priority queue ADTs is subtle.

As noted, out-of-order processing happens when, while processing $ADT_k$ a state is derived:

> and then at a later time, before $ADT_k$ is exhausted, its rightmost antecedent is rederived using the same rule it was derived with the first time.

In the previous subsection, $(S, [D, C, @], [0, 5])$ was derived via rule 6, yielding a certain value for the best derivation of S, and then C was rederived later on using the same rule. Meaning that 1) C was processed via rule 7 to yield S, 2) S was processed and removed from the list and stored as a completed sentential form, then 3) C was dervied via rule 7 *again* yielding a new best derivation of S. At this point, however, S's utility is not updated because the state $(C, [A, B, @], [1, 5])$ already existed.

I will now show that with a suitably chosen comparator function, a priority queue ADT will resolve out of order processing. I will first show that ranking states in decreasing order of start state fails if we break ties using the order in which states were added (FIFO) if our grammar has unary productions. After that I will show a more robust tie-breaking method that succeeds.

#### 2.7.4 Priority queue sorted by start index

For a given derived symbol A, there are two cases, either $A$ is derived from a rule with multiple terms on its RHS, or it is derived from a unary rule. Note that per the stipulations of the previous section, we exclude grammars with unary cycles, but permit unary rules in general.

In the first case,

$$R\,\frac{[B_1, i, i'], ..., [B_n, i'', j][C, j', k]}{[A, i, k]}\,\mathrm{A} \rightarrow B_1 ... B_n C$$

We know that $B_i$ are not empty and therefore $j' > i$. In which case, we will never have $[A, i, k]$ processed before the final time $[C, j, k]$ is processed because the ordering predicate of the priority queue will ensure that all possible derivations of $[C, j, k]$ will occur before $[A, i, k]$ is derived.

Therefore, if we have no unary productions we can simply rank by start index and don't even need a tie-breaking scheme. If, however, we admit unary productions (even if they're not cyclic) we need a more advanced tie-breaking strategy to prevent out-of-order processing.

Consider the case of a unary production:

$$R\,\frac{[C, i, k]}{[A, i, k]}\,\mathrm{A} \rightarrow C$$

The only way for out-of-order processing to occur is if 1) $[C, i, k]$ is derived via some rule $R$, 2) it is then processed to derive $[A, i, k]$, then 3) its antecedent in rule $R$ is rederived, causing a rederivation of $[C, i, k]$

after $[A, i, k]$ was already derived from it. This is only a problem if $[C, i, k]$ is derived via rule $R$ twice, once before $[C, i, k]$ derives $[A, i, k]$ and once after.

Thus, our priority queue must have the property that if $[C, i, k]$ is derived by rule $R$ with antecedent $[B_r, j', k]$, then the antecedent must never be placed on the queue again after $[C, i, k]$ is popped.

Note that the first ordering predicate resolves cases where $j' \neq i$, thus we only need to consider the case in which $j' = i$, that is $[C, i, k]$ is derived from $B_r$ using the unary rule:

$$R \frac{[B_r, i, k]}{[C, i, k]} \mathrm{C} \rightarrow B_r$$

I must now show that it is possible to construct a grammar in which $[B_r, i, k]$ derives $[C, i, k]$, $[C, i, k]$ is popped off the queue to complete $[A, i, k]$, then $[B_r, i, k]$ is placed back on the queue.

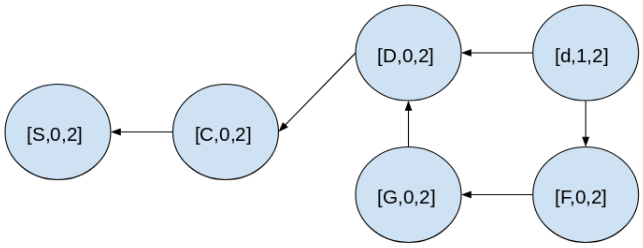
To show this, consider that all states completed on $ADT_k$ have $\alpha_k$ as their right corner terminal. Using this observation we can use the grammar to construct a DAG relating the states stored in $ADT_k$, in particular there will be an edge from the state scanning $\alpha_k$ ($[\alpha_k, k-1, k]$) to whichever LHS's are completed by the scan. Additionally, for any two nonterminals $A, B$ in the grammar, there is an edge from a state having LHS $B$ to a state having LHS $A$ if there exists a rule $A \rightarrow \alpha B \rightarrow^* \alpha_i, ..., \alpha_k$, with $\alpha$ potentially empty and where $*$ indicated transitive closure. This formulation means that $A$ derives $\alpha_i, ..., \alpha_k$ via direct application of a rule with $B$ as its rightmost term on the RHS.

We are now ready to show a grammar that uses unary productions to create out-of-order processing when we use random tie-breaking in our priority queue. Consider the grammar below

**Grammar 2**

$$S := C \tag{2.12}$$
$$C := D \tag{2.13}$$
$$D := d\ d \tag{2.14}$$
$$G := F \tag{2.15}$$
$$F := d\ d \tag{2.16}$$

While processing input $d\ d$ we can construct the following graph for the states in $ADT_2$:




The states adjacent to $[d, 1, 2]$ will be completed first, followed by the direct neighbors of those states in some arbitrary order.

For example, we add the first generation of completed states $[D, 0, 2], [F, 0, 2]$. We then dequeue $[D, 0, 2]$ and add $[C, 0, 2]$, to get $[F, 0, 2], [C, 0, 2]$. Subsequent steps yield $[C, 0, 2], [G, 0, 2]$. We will soon fall prey to out-of-order processing, because we dequeue $[C, 0, 2]$, then rederive its antecedent $[D, 0, 2]$ via $[G, 0, 2]$. After rederiving $[D, 0, 2]$, we will dequeue $[S, 0, 2]$ for the first time. $[S, 0, 2]$ will never be processed again, because the second derivation of $[C, 0, 2]$ is via the same rule, and therefore doesn't result in $[C, 0, 2]$ being enqueued again with its new utility value. Thus $[S, 0, 2]$ gets processed once via the shortest path from $[d, 1, 2]$ and then never gets notified that a new path with different attributes is discovered.

### 2.7.5 Topological sorting

The issue plaguing the previous solution was that it is possible for a state $S$ to be reachable by paths of different length, resulting in its successor $S'$ to be processed once before $S$ see all of its possible derivations.

This problem suggests the very straightforward fix of breaking ties via topological sort because clearly topologically sorting graphs such as that above, results in all antecedents of a node being processed before it is processed - the exact circumstance which causes out-of-order processing.

Because there are no unary cycles in our grammar, there exists a global topological sort of nonterminal symbols derivable by unary productions that is valid for all right-corners $\alpha_k$. This means that, no matter what terminal scan begins $ADT_k$ there is a unique topological sorting which will tell us what order to perform the completions in thereafter. Thus we don't need to perform a new topological sorting every completion step - a single topological sort before parsing even begins will suffice! This can be shown via a proof by contradiction.

If there are no unary cycles in the grammar, then we can construct a graph $G$ in which each node is a nonterminal and there is a directed edge from symbol B to A iff there is a unary production deriving A from B. We will also say that we can construct a graph $G_k$ which contains all the states completable from $ADT_k$ with edges pointing from state $A$ to be $B$ if $A$ completes $B$. Note that $G_k$ has $\alpha_k$ as its source node, and $G_k/\alpha_k$ is a subgraph of $G$.

**Theorem 2.** *The $G$, $G_k$ constructed in the manner above have the following properties:*

1. *G is acyclic and has a valid topological ordering*
2. *There exists some topological ordering over the symbols in G that is valid for all graphs $G_k$ created by extending G to have right corner symbol $\alpha_k$ as source node*
3. *Any topological ordering of G is valid for all $G_k$ extended by right corner symbol $\alpha_k$*

*Proof.* Item one above is trivial. By definition there are no unary cycles so the graph of all unary derivations is acyclic.

For item two, assume that there exist two distinct symbols $\alpha_j$ and $\alpha_k$ for $j < k$. Assume that at time $j$ there is a topological sorting of all states completable by $[\alpha_j, j, j+1]$. Call the corresponding graph $G_j$. Additionally, assume that there is a topological sorting for the states completable by $\alpha_k$. Call the corresponding graph $G_k$. Assume by contradiction that the two topological sortings are inconsistent with one another, that is there exist two symbols such that $B \prec A$ in one graph and $A \prec B$ in another.

Each $G_j$, and $G_k$ has a unique source, and all interior nodes are nonterminals, therefore any path through $B$ to $A$ in $G_j$ and any path from $A$ to $B$ in $G_k$ must pass through only nodes and edges which are contained in $G$. Thus in order for $A \prec B$ in one graph and $B \prec A$ in the other, there must be a cycle in $G$, which by assumption is not the case.

For item three, as noted above $G_j/[\alpha_j, j, j+1]$ is a subgraph of $G$. Therefore a topological ordering over $G$ is also a valid topological ordering of $G_j/[\alpha_j, j, j+1]$. It is trially true that $[\alpha_j, j+, j+1]$ is topologically first in $G_j$ since it is the unique source. Therefore the topological ordering of $G_j$ is the topological ordering of $G_j/[\alpha, j, j+1]$ with $[\alpha, j, j+1]$ inserted at the front. □

This proof shows that we can simply topologically sort the nonterminal symbols by unary production when preprocessing the grammar, and use the value of each symbol in that topological sort as the tie breaker for the priority queue.

### 2.7.6 Queue ADTs

I will omit the proof that standard queues experience out-of-order processing because it is a trivial extension of the same proof for priority queues with arbitrary tie-breaking.

# Chapter 3

# Noise-skipping Earley parsing over the attribute-vector semiring

### 3.1 Abstract

In the previous chapter, I introduced a novel attribute-vector semiring defined by a domain of real-valued attributes, a utility function $f(\cdot)$, and a pooling function $g(\cdot)$. Using this semiring, I showed that context-free parsing systems can associate with every constituent the set of attributes and utility score belonging to its best derivation. In that chapter, I not only demonstrated the existence of validity of that semiring, but showed that the underlying chart in the Earley algorithm must be a priority queue whose sorting predicate is a state's start index, with ties broken by a topological sort on the graph of unary productions.

While this chapter provided useful guarantees about attribute-vector utility functions, it didn't provide a complete computational description of how to integrate the system into the Earley algorithm, nor did it address the modifications to the Earley algorithm necessary to handle skipping tokens during the parse.

This chapter will be self-contained. Anybody with an understanding of context-free grammars but no background in parsing will be able to learn the traditional Earley algorithm [18], Elizabeth Scott's algorithm for constructing share -packed parse forests [38], and the innovations I introduced to enable noise-skipping and to track attribute-vectors and utility scores.

First, I will briefly sketch the 3 phases of the Earley algorithm and the manner in which its data structures are populated to permit parse forest extraction. Next, I will show the modifications necessary to the core algorithm in order to track parses over skipped portions of text and the additional data structures needed to ensure metadata for optimal parses can be maintained. Then I will discuss four specific parse tree attributes - tokens explained, constituents encountered, noise skipped, and tokens skipped - all of which are suitable candidates for use in a utility function.

Finally, I will discuss time and space complexity. The algorithms to construct the ordered SPPF and extract its trees will be discussed in chapters 4 and 5.

### 3.2 Introduction

In 1969, Jay Earley developed the first polynomial-time algorithm to parse general context-free grammars [18]. Previous algorithms had been developed for specific kinds of CFG and accomplished cubic or subcubic time on those special classes of grammar. Some of these early developments are covered in [18]. Earley's publication marked the first cubic time algorithm for general CFGs and marked a substantial development for both its excellent theoretical bounds and its fast performance in practice. Earley's algorithm has been enduringly popular as well because it is particularly well-suited to easy construction of shared packed parse forests [38] and because its flexibility with respect to the grammar's representation makes parsing with massive grammars practical when used in conjunction with good filtering schemes see [8] as well as the appendix. As explained in [8], Earley's algorithm is particularly amenable to on-the-fly preprocessor filtering because it doesn't use costly a priori computation of LR tables or shift/reduce tables like other fast algorithms such as GLR.

### 3.3 Earley's algorithm

#### 3.3.1 States and charts

**States**

Earley's algorithm was originally introduced and is most succinctly described in [18]. An excellent exposition can be found in chapter 30 of [30] as well as source code for an OOP implementation in Java. [30] strays very little from Earley's original notation and concepts so I will use their presentation. Their notation is also an intuitive extension of that used in the deductive logic presentation used in the previous chapter.

Earley's algorithm is part of a general family of parsers known as chart parsers. The atomic unit of information used in Earley's algorithm is the *state*. A state consists of a start index $i$, and end index $j$, a constituent representing the $LHS$ of a rule, an array representing the $RHS$ of that rule, and a dot indicating how much of the rule has been confirmed up to token $j$. Thus a state represents a hypothesis that symbol $LHS$ will be found starting at index $i$, tokens $[i, j)$ have been parsed as the elements of the RHS preceding the dot, and a hypothesis that the remainder of the RHS will be found starting at index $j$.

Example states are given below, where @ is used to indicate the dot:

1. $(root,\ [@,\ S],\ [1,\ 1])$
2. $(S,\ [\alpha,\ @,\ S,\ \beta],\ [1,\ 2])$

3. $(S,\ [\alpha,\ S,\ \beta,\ @],\ [1,\ 6])$

Item 1 above represents a prediction that the rule $root \rightarrow S$ will be completed beginning at index 1. The location of the dot indicates no tokens have been consumed.

Item 2 tells us we are predicting a constituent of type $S$ beginning at index 1. Additionally, token $\alpha$ was scanned at index 1. In order to complete the parse, another constituent $S$ must be parsed starting at token 2 followed by a token $\beta$.

Item 3 shows a completed state. It indicates that the tokens 1 through 5 were parsed as an $S$ via rule $S \rightarrow \alpha\ S\ \beta$. Paying close attention to the indices we can see that the middle $S$ was parsed from by matching 2 through 4 (this would be indicated by indices $[2, 5]$).

We will see when we introduce noise-skipping it may be possible for $S$ to match span $[2, 5]$ but skipped the middle token (3), or that tokens were skipped between the end of $S$ and the $\beta$ scanned at index 5 when $(S,\ [\alpha,\ S,\ D,\ \beta],\ [1,\ 6])$ was completed.

**Charts**

During parsing, states are placed into a list of data structures $ADT_k$, called a chart, where each index $k$ in the chart stores the states whose end index is $k$. Since end indices are exclusive upper bounds, the list of $ADT_k$ will always be longer than the input string by 1.

$ADT_k$ can be any dynamically sized data structure and can either accumulate and maintain all states ever encountered whose end index is $k$, or can be added to and modified in-situ such that a state is popped off (dequeued, polled, etc.) as it is processed, resulting in processing for token $k$ being complete when $ADT_k$ is empty.

For quick look up, the chart typically contains a hash table mapping symbols in the vocabulary to a set of states in the chart which have that symbol immediately preceding their dot. Furthermore, a chart will keep track of every state that has ever been added to it, so that if a state gets added once (due to some derivation) and then added again (via a new, different derivation) it is not added to the chart to be processed again, but rather merged with all previous instances of that state. In [38], this is done by merging two types of state metadata called the reduction and precessor lists. In our noise-skipping variant we will also merge the attributes for the constituent derived by each state.

In this work, a state is considered identical to another previously seen state if their LHS, rule, dot index, and span $i, j$ are all identical.

Though in standard Earley parsing it is irrelevant what data structure you use for $ADT_k$, as shown in the previous chapter, correctly evaluating best attributes for a constituent requires a priority queue sorted by start index with ties broken via topological sort over the grammar.

### 3.3.2 Scanning, prediction, and completion

The input sentence is processed one index at a time starting by populating $ADT_1$ with a set of starting states (I will only consider a single unique starting state $(root,\ [@,\ S],\ [1,\ 1])$. For each index $k \in [1, n+1]$, we dequeue states from $ADT_k$, and process each state. Each state can be processed via the application of three possible operations: scanning, prediction, or completion. Each state will only ever be eligible for one of these operations. The operations, if successful, may result in the creation of new states to be added either onto $ADT_k$ or $ADT_{k+1}$. Once all states have been dequeued from $ADT_k$, the loop variable increments and the next $ADT$ is emptied.

Once all $ADT$s in the chart have been processed, under most conventions, the parse is considered successful if there is a completed state indicating a start state was completed over range $[1, n+1]$.

A high-level sketch of this algorithm is below:

**Algorithm 1** Earley algorithm

```
1: Chart ← ADT[n + 1] fill the chart with n+1 ADTs
2: start_state ← (root, [@, S], [1, 1])
3: Chart[1].enqueue(start_state)
4: for k ← 1 to n + 1 do
5:     ADT_k ← Chart[k]
6:     next_state ← ADT_k.dequeue()
7:     ApplyOperation(Chart, next_state)
   return (root, [S, @], [1, n + 1]) ∈ Chart[n + 1]
8: procedure APPLYOPERATION(Chart, state)
9:     if state.dot_at_end() then
10:        Complete(Chart, state)
11:    else if state.symbol_after_dot is terminal then
12:        Scan(Chart, state)
13:    else
14:        Predict(Chart, state)
```

### Scanning

Scanning is performed on states who have a terminal following their dot, e.g. **St** $:= (S, [\alpha, S, @, \beta], [j, k])$. Performing a scan on **St** would involve checking whether the token at index 5 matches the terminal symbol following the dot (namely $\beta$). If it matches then a new state indicating the consumption of that token at index $k$, **St'** $:= (\beta, [\beta, @], [k, k + 1])$, will be placed on $ADT_{k+1}$. Scanning is the simplest operation as it requires a simple check to see if the next element of the input matches the element expected after the dot, either creating the state described above if it succeeds, or failing to do so and moving on to the next state in $ADT_k$ or onto the next $ADT_{k+1}$.

This looks as follows:

**Algorithm 2** Earley algorithm; scanning

```
1: procedure SCAN(Chart, state)
2:     idx ← state.start_idx
3:     symbol ← state.symbol_after_dot
4:     if input[idx] = symbol then
5:         st ← (symbol, , [symbol, @], [idx, idx + 1])
6:         Chart[idx + 1].enqueue(st)
```

### Prediction

Prediction is the nonterminal analogue of scanning. It is performed while processing states of the form **St** $:= (A, [B, @, C, \beta], [j, k])$, where the symbol following the dot is a nonterminal. A state such as **St** can be thought of as hypothesizing that a constituent of type $C$ will be found starting at index $k$.

In order to perform this processing, we must find every rule $R$ in the grammar containing $C$ as its LHS and initialize a state in chart $k + 1$ for each such rule hypothesizing the derivation of $C$ via rule $R$ starting at index $k$.

For example if our grammar had the rules $C \rightarrow D\ E$ and $C \rightarrow \delta\ E$, then two new states would be placed onto $ADT_k$

**St'** $:= (C\ , [@, D, E], [k, k])$
**St'** $:= (C\ , [@, \delta, E], [k, k])$

Note that the operations described above create entirely new states, but they don't move the dots of existing states in order to represent the fact that we have updated a hypothesis about the identity of some span of text. The job of creating such successor states belongs to the completion operation.

It is important to note that predicted states are placed on the same $ADT_k$ as the state that generated them. This is the primary mechanism by which a chart entry will grow while looping. Pseudocode is below:

**Algorithm 3** Earley algorithm; prediction

```
1: procedure PREDICT(Chart, state)
2:     idx ← state.end_idx
3:     symbol ← state.symbol_after_dot
4:     for rule ∈ Grammar[symbol] do
5:         st ← (rule.LHS, @ ∘ rule.RHS, [idx, idx])
6:         Chart[idx].enqueue(st)
```

**Completion**

Completion is performed when processing a state whose dot occupies the final position in its array. For example **St'** := $(B,\ [B,\ @],\ [j,\ k])$ would be a candidate for completion.

Completing the state above would proceed by searching $ADT_j$ (corresponding to the start index of the completed state) for states whose element immediately following the dot matches the LHS of **St'**. For example completion of **St'** could find the following states in $ADT_j$:

$$\textbf{St1} := (S,\ [\alpha,\ S,\ @,\ B],\ [i,\ j])$$
$$\textbf{St2} := (C,\ [@,\ B,\ D],\ [j,\ j])$$

We would then produce successor states whose dots have been advanced one entry, and whose end index has been set to the end index of the state we are completing ($k$ in this example). This would produce:

$$\textbf{St1'} := (S,\ [\alpha,\ S,\ B,\ @],\ [j,\ k])$$
$$\textbf{St2'} := (C,\ [B,\ @,\ D],\ [j,\ k])$$

The successor states are always pushed onto the $ADT$ currently being emptied (because their end index is now set to be the same as the end index of the state we are completing), therefore they too will be processed during this round of the algorithm.

By convention, the algorithm is said to have succeed when a sentential form is completed, e.g. when $(root,\ [@,\ S],\ [1,\ 1])$ is completed to $(root,\ [S,\ @],\ [1,\ n+1])$

Pseudocode is below:

**Algorithm 4** Earley algorithm; completion

```
1: procedure COMPLETE(Chart, state)
2:     end_idx ← state.end_idx
3:     start_idx ← state.start_idx
4:     LHS ← state.LHS
5:     for incomplete_state ∈ Chart[start_idx] whose symbol after dot is LHS do
6:         st ← clone(incomplete_state)
7:         st.increment_dot
8:         st.end_idx ← end_idx
9:         Chart[end_idx].enqueue(st)
```

### 3.4 Extending Earley to construct parse forests

The astute reader may have noticed that this algorithm doesn't constitute a parsing algorithm but rather a recognition algorithm. This is so because the algorithm of the previous section can only tell whether a sentential form has been found, but it doesn't specify a mechanism for reconstructing the parse tree(s) deriving the sentence. At the time of writing Earley believed he had introduced a correct algorithm to extend

his recognizer into a parser [18], but it was discovered in 1985 [43] that Earley's algorithm for parse extraction generates spurious parses. As noted in [38], there have been many other formulations of the Earley parser that resulted in spurious parses, including that presented in the reference textbook implementation of [30]. It wasn't until 40 years after Earley's original paper that [38] showed a provably correct algorithm for building a parse forest data structure known as a binarized shared packed parse forest, which succeeds in creating a cubic-sized data structure capable of storing exponentially many parses. Scott's SPPF is an unordered data structure and presents no algorithm for extracting the trees in any particular order. Other works [37] have studied performing operations on the SPPF to eliminate ambiguities, but to my knowledge chapter five of this thesis represents the first time anybody has developed a method for extracting trees from the parse forest in a ranked-order in polynomial time.

I will omit Scott's formulation of the Earley algorithm as it uses a much different notation and vocabulary from this work. Instead, I will indicate how its operations integrate into the description of the Earley algorithm so far. Scott's core contribution leaves the actual procedure of the Earley algorithm in place, but adds data structures to track the derivational relationships between states. In particular, she relies on two relations: the "predecessor" relation and the "reduction" relation.

In addition, to demonstrating how define and keep track of these relations, she shows that you can construct an SPPF from this metadata. I will defer discussion of how exactly to construct the SPPF to the next chapter.

### 3.5 Predecessor/reduction

Adapting the presentation in [38], we will say that each Earley state has two dictionaries: the predecessor dictionary $pred : \mathbb{N} \mapsto$ **List**(State) and the reduction dictionary $red : \mathbb{N} \mapsto$ **List**(State) which map integer indices to lists of states.

Modifying the predecessor and reduction dictionaries of a state occurs only during the completion phase. There are two cases, either we are completing a state whose LHS is a terminal or completing a state whose LHS is a nonterminal:

1. While completing a state whose LHS is a terminal $t := (a_j, [a_j, @], [j, j+1])$ we search for states whose symbol following the dot is $a_j$, $predecessor\_state := (A, [\alpha, @, a_j, \beta], [i, j])$, whose dot can be advanced by one. This proceeds by producing a successor state $successor\_state := (A, [\alpha, a_j, @, \beta], [i, j+1])$. At this point we will add $predecessor\_state$ to the list in index $j$ of $successor$'s predecessor dictionary. This can be written in pseudocode as follows: $successor.pred[j].append(incomplete\_state)$[1]. [38] frames this process as creating a labeled edge with label $j$ from $predecessor$ to $predecessor\_state$ in a graph of predecessor relations.

2. While completing a state whose LHS is a nonterminal $nt := (B, [\beta, @], [j, k])$, we search for states whose symbol following the dot is $B$, $predecessor\_state := (D, [\delta, @, B, \mu], [i, j])$, whose dot can be advanced by one. This proceeds by producing a successor state $successor := (D, [\delta, B, @, \mu], [i, k])$. At this point we create a reduction edge labeled $j$ from $successor$ to $nt$. If $\delta$ is non-empty, then we also add a predecessor edge from $successor$ to $predecessor\_state$ labeled $j$.

The general relationship is that predecessor relationships point from a state $successor$ to the state $predecessor\_state$, where $successor$ and $predecessor\_state$ are identical except for the dot in $successor$ being moved forward one index relative to $predecessor\_state$'s. Reduction relationships point from a state $successor$ to a state $nt$ if $successor$ was produced from $predecessor\_state$ by derivation of the nonterminal on the LHS of $nt$.

If, at any point, a state is created that is identical to a state that has already been created before, its pred/red dictionaries are merged.

At this point we can fill out the completion method presented earlier:

[1] [38] actually has the edge pointing in the opposite direction: $predecessor\_state.pred[j].append(successor)$, though I believe this is a typo

**Algorithm 5** Earley algorithm; completion with predecessor/reduction

```
1: procedure COMPLETE(Chart, state)
2:     end_idx ← state.end_idx
3:     start_idx ← state.start_idx
4:     LHS ← state.LHS
5:     for predecessor_state ∈ Chart[start_idx] whose symbol after dot is LHS do
6:         successor ← clone(incomplete_state)
7:         successor.increment_dot
8:         successor.end_idx ← end_idx
9:         Chart[end_idx].enqueue(sucessor)
10:        if LHS is a nonterminal then
11:            successor.red[start_idx].append(state)
12:            if predecessor_state.dot_index > 0 then
13:                successor.pred[start_idx].append(incomplete_state)
14:        elsesuccessor.pred[start_idx].append(incomplete_state)
15:
```

### 3.6 The noise-skipping Earley parser

Now that we have established the technical foundation for the traditional Earley algorithm I will define the "noise-skipping" problem and some crucial terminology. The remaining subsections will concern themselves with the modifications necessary to generalize the algorithm described in the last two sections to the noise-skipping problem as well as how to calculate and maintain attribute vectors for best derivations for each state, as defined in chapter two.

Noise skipping parsing is a form of parsing that allows rules to be completed over noncontiguous spans of text. Whereas before we required that states and their successors be directly aligned with one another, during noise skipping we allow some (potentially bounded) number of tokens to intervene. A simple example is parsing the grammar $S \rightarrow A\ B\ C$. Whereas under traditional parsing we can only parse the literal string "A B C", a noise skipping parser can parse "A d B C", "A d d d d B d d C", and even "A B C C" where we recognize the parse which skips the first C in favor of the second C as constituting a valid parse. This discontinuity modifies the behavior of all three of the operations in the Earley algorithm and also requires some additional overhead to maintain efficiency.

Additionally, even though ranking parses and extracting them in order from parse forests is also a problem of technical interest in traditional Earley parsing, concern about this problem is particularly acute in the noise-skipping context because the relaxation of adjacency constraints can lead to a proliferation of possible parses.

There are two distinct types of skipping one might be interested in: skipping out of vocabulary items (which I refer to from hereon out as **OOV-skipping**) and skipping items in the vocabulary (which I refer to from hereon out as **token-skipping**). If you want to ignore OOV items entirely and don't care about how many were skipped, where in your parse they occur, etc., the former could be accomplished by simply removing OOV items from your input altogether. The second one is more challenging as it cannot be handled by preprocessing alone, and therefore requires extending the Earley algorithm. Both types of skipping may come up in a variety of language contexts. For instance, during spontaneous speech people might utter OOV utterances such as 'uh' or 'um', and also utter items in the vocabulary which you would otherwise like to ignore, for instance false-starts or stammering - thus it is valuable to address both types of skipping.

I will parametrize the number of OOV items or number of tokens one would like to allow a valid parse to skip by the letter $w$, though in theory you can leave $w$ equal to the input size.

Additionally, we will always maintain the important an invariant that the start and end indices of a state are the first and last tokens explained in that span. Thus we will never have a state's span start with skipped items, or have it end with skipped items.

### 3.6.1 Preprocessing noise

The approach laid out in this paper performs a preprocessing step to tag all OOV items. You will see when I present the protocol for **token-skipping** that this is actually unnecessary, but by pre-tagging all noisy spans, we can allow our forwards scans to skip those indices entirely, which is much cheaper than enqueuing states onto indices in the chart which will never yield any results. The following sections will assume we have preprocessed our input such that we have an $O(1)$ way to check whether an index $i$ is the start of a sequence of OOV items, a map to return $j$, the end of that span, given $i$ as well as the reverse mapping. I will also assume that every time we complete a state having start and end index $[i', j']$ we cache the number of OOV items occurring in that span (this will be used in our practical attribute vector example that tracks amount of noise skipped).

### 3.6.2 Scanning and prediction

Scanning and prediction are nearly identical to their standard Earley parsing variants save for three main differences.

Recall that scanning occurs by taking states expecting a terminal $s := (A, [\alpha, @, a, \beta], [i, j])$ and checking whethere the $j^{th}$ element of the input string, $a_j$, matches $a$. Recall, additionally, that prediction occurs by taking states expecting a nonterminal $s := (A, [\alpha, @, B, \beta], [i, j])$ and spawning a state of the form $(B, [@, ...], [j, j])$ for each rules $B \rightarrow ...$ whose LHS is the symbol following $s$'s dot.

Both methods share the common feature that they attempt to find a substring starting at index $j$ of the input that matches the symbol after the dot (in the case of scanning, its a literal symbol, in the case of prediction its an entire nonterminal). Extending standard Earley to noise skipping simply relaxes the constraint that the matching substring must begin at $j$. Instead, we calculate a list of valid indices $j'_1$,..., tabulating the number of tokens/OOV items and then perform a new prediction/scan at each index $j'$. A naive approach would be fairly trivial, but I will present a more sophisticated approach that uses the precomputation mentioned in the previous subsection, maintains the indexing invariant established earlier, and avoids placing states on indices corresponding to OOV indices in the input (all of which yield substantial benefits when presented with data having long spans of OOV items).

**Algorithm 6** Noise-skipping Earley algorithm; forward indices for scans/predictions

```
1:  procedure GETFORWARDINDICES(state, j)
2:      oov_index_map ← precomputed
3:      j' ← j
4:      oov_skipped ← 0
5:      forward_indices ← {j' : 0}          ▷ predicted indices j' and # OOV items between j and j'
6:      w ← Maximum allowable skipped tokens
7:      if state.dot_index = 0 then return forward_indices
8:      else if state.dot_index > 0 then
9:          if j' ∈ oov_index_map then
10:             oov_skipped += oov_skipped[j'] + 1 − j'
11:             j' ← oov_index_map[j']
12:     for i = 1; i ≤ w ∧ i < |input|; i += 1 do
13:         j' += 1
14:         if j' ∈ oov_index_map then
15:             oov_skipped += oov_skipped[j'] + 1 − j'
16:             j' ← oov_index_map[j']
17:         if j' < |input| then
18:             forward_indices[j'] = oov_skipped
        return forward_indices
```

Thus the first modification to noise-skipping scanning/prediction occur via producing the states as usual, but creating one unique state for each of the indices generated by 'GetForwardIndices' and caching the amount of noise skipped in range $[j, j']$. The second modification is that in addition to each state main-

taining the number of OOV items and number of tokens skipped separately, they will also store $j' - j$, the total length of the span skipped. The purpose of doing so will be made apparent later on.

Note that prediction will result in states whose indices are $[j', j']$ and scanning will result in states whose indices are $[j', j' + 1]$ which will be stored on $ADT_{j'}$ and $ADT_{j'+1}$ respectively.

The final modification is not directly related to noise skipping but rather is a safe guard to prevent out-of-order processing, ensuring the validity of parsing over the semiring. Each newly produced state is assigned a topological sort number to assist in tie breaking for the priority queue underlying $ADT_k$. The topological sort number is determined by the precomputed topological sort of the grammar described in chapter 2.

### 3.6.3 Completion

Completion requires more substantial modifications for two reasons: 1) noise-skipping changes the conditions governing which states $q$ can have their dot moved via completion of a state $s$, 2) changes are required for the predecessor/reduction labeling scheme of [38], 3) completion corresponds to the stage of semiring parsing at which attribute vectors are merged via the pooling function $g(\cdot)$ and when new best-utilities are computed via $f(\cdot)$.

As mentioned previously, completion occurs when a state $s := [\Gamma, [\gamma, @], [j', k])$ is processed. Applying the completion operation to s searches for all states of the form $q := (D, [\alpha, @, \Gamma, \beta], [i, j])$ whose symbol after the dot matches the LHS of state s. From q, the new successor state $q := (D, [\alpha, \Gamma, @, \beta], [i, j])$ is produced. Note that we have relaxed the traditional Earley algorithm's constraint that $j = j'$ and we must now search chart entries $ADT_j$ for $j' - j < |w|$. Accomplishing this works by a mechanism very similar to that used in $GetForwardIndices$ but runs it in reverse. The algorithm assumes the existence of an easy way to look up $ADT_k$ entries by the symbol after their dot (computable via a dictionary).

**Algorithm 7** Noise-skipping Earley algorithm; reverse indices for completions

```
1:  procedure GETCOMPLETED(state, LHS)                ▷ LHS is the left hand side of state
2:      completed_states ← {}
3:      i ← s.start_index
4:      backward_indices ← GetBackwardIndices(s, i)
5:      for j ∈ backward_indices do
6:          completed_states.extend(Chart[j][LHS])    ▷ Searching states by the symbol after their dot
        return completed_states
7:  procedure GETBACKWARDINDICES(s, j)
8:      oov_index_reverse_map ← precomputed
9:      j' ← j
10:     backward_indices ← [j']
11:     w ← Maximum allowable skipped tokens
12:     for i = 1; i ≤ w ∧ i < |input|; i += 1 do
13:         j' −= 1
14:         if j' ∈ oov_index_reverse_map then
15:             j' ← oov_index_reverse_map[j']
16:         if j' < 0 then return backward_indices
17:         backward_indices.append(j')
        return backward_indices
```

The key difference between the forward and backward index methods is that in the backward case we no longer need to keep track of noise skipped in the range because we can assume it has already been cached from the corresponding call to 'GetForwardIndices' that put state 's' on $ADT_j$ to begin with.

The other primary algorithmic changes relate to the labeling conventions of reduction and predecessor edges mentioned in the previous section. As noted previously, when processing state $s$ that has been completed we create a successor state $successor$ from $predecessor\_state$ if $incomplete\_state$ was anticipating the LHS of $s$ after its dot. If $predecessor\_state$'s dot was not at the start position then we add a predecessor edge from $successor$ to $predecessor\_state$ with label $j$ (unless $s$ is a terminal in which case we always add

the edge). If $s$'s LHS was a nonterminal then we create a reduction edge from $successor$ to $s$ with label $j$ where $j$ is the start index of $s$ and the end index of $predecessor\_state$. As I am sure you can predict, we are going to relax the constraint that $j$ be an index common to $s$ and $predecessor\_state$, which is going to change how the labels work.

This can be illustrated using the states below:

$$\begin{aligned} s &:= (C, [\alpha, @], [j', k]) \\ predecessor\_state &:= (A, [\beta, @, C, \delta], [i, j]) \\ successor\_state &:= (A, [\beta, C, @, \delta], [i, k]) \end{aligned}$$

In the noise-skipping case, the start index of $s$ is some $j' \geq j$, the end index of $predecessor\_state$. Again we handle the cases of nonterminals and terminals differently.

1. If $s = (a, [a_{j'}, @], [j', j' + 1])$ has a terminal as its LHS then we still label the predecessor edge from $successor = (A, [\beta, a, @, \delta], [i, j' + 1])$ to $predecessor\_state = (A, [\beta, @, a, \delta], [i, j])$ with label $j$ (where $j$ is the end index of $predecessor\_state$)

2. If $s = (C, [\alpha, @], [j', k])$ has a nonterminal as its LHS we label the reduction edge with $j'$and the predecessor edge with $j$.

It may be surprising that the nonterminal case is modified from the original algorithm while the terminal case is not. The reason for this is subtle and has to do with the differences between how terminal and nonterminal nodes are constructed in the SPPF [38]. It will become clear in chapter four why this labeling scheme is correct.

### 3.6.4 A word on attribute calculation

As noted, one of the aims of this algorithm is to compute best-attribute vectors with utility functions defined over the semiring of chapter 2. There are four crucial components to doing so correctly:

1. Maintain the correct topological sort values for the states

2. Resolve conflicts when freshly deriving state $S'$ which is identical to a state $S$ produced earlier in the parse.

3. Calculate the attributes for each state (we have actually been showing how to do so for two useful attributes: OOV items skipped and tokens skipped). There isn't really a general form for how to do so as it will depend on the nature of the attribute.

4. Merge attributes during completion

I have already discussed how to generate the topological sort values. For conflict resolution, when merging two copies of identical states (drawn from different derivations), we create a new $S''$ whose attributes are $argmax_{\mathbf{x} \in \mathbf{a}, \mathbf{a}'} f(x)$ where $a, a'$ are the attributes of $S$ and $S'$ respectively. Additionally, we merge all predecessor/reduction lists.

I have already shown for example how to calculate the OOV items skipped and tokens skipped during scanning and prediction. I will now show how to correctly the "OOV items skipped" attribute during completion:

We will leverage the cached data we have on the number of OOV items skipped in each $[i, j]$ range encountered during $GetForwardIndices$.

1. If state $nt := (B, [\beta, @], [j', k])$ is undergoing completion, where $B$ is a nonterminal, and we are yielding a new successor state $p := (D, [\delta, B, @, \mu], [i, k]$ from its predecessor $q := (D, [\delta, @, B, \mu], [i, j]$, then we look up the number $N$ of OOV items between $j, j'$. This is the rule-specific value for this attribute. We can then pool this value with the corresponding values for $q$ and $nt$ via the pooling function $g(\cdot)$.

2. The case for the LHS of $nt$ being a terminal is otherwise the same except $nt$ would have the value 0 for its number of OOV items skipped.

### 3.6.5 Completion metadata

There is a final bit of accounting we will use which will be use in the next chapter when we build the SPPF.

We will store a **completion metadata** map which maps tuples $(nt, q)$ to the vector of attributes $X_p$, where $p$ is a successor of $q$ via completion of $nt$. Note that though $p$ is uniquel determined by $(nt, q)$, the relationship is not bijective since multiple tuples $(nt', q')$ could result in a $p$.

It turns out we only need to store the mapping $(nt, q) \mapsto X_p$ the first time it is encountered in order to maintain the optimal attributes associated with deriving $p$ by way of $(nt, q)$. This is so because if no out-of-order processing occurs, as remedied by chapter 2, then $nt$ and $q$ are never rederived once $p$ is derived for the first time. This means that we know exactly what the best derivations of $nt$ and $q$ are when we first make the mapping $(nt, q) \mapsto X_p$. Therefore, any derivation of $p$ with a different best attribute vector $X_p$ must occur via a different $(nt', q')$

## 3.7 Complexity

## 3.8 Time complexity

I will first show a brief description of Earley's proof of his algorithm's run time, and mention the minor adjustments necessary to apply it to the noise-skipping algorithm. Original proofs for Earley's run time can be found in [18] and [19]. The presentation of this proof is adapted from [2] to fit the notation used in this chaper.

**Theorem 3.** *The run time of the standard Earley algorithm is $O(n^3)$ where $n$ is the length of the input.*

*Proof.* It is easy to see that there are at most $O(|G|)$ possible dotted rules - (LHS, RHS, dot) triplets, where $|G|$ is the grammar size, because for every production with $p$ elements on its RHS there are $O(p)$ possible locations to put a dot. Summing the length of each rule over all of the rules is the exact definition of grammar size.

We can show the run time by bounding the maximum number of states possible in $ADT_k$. Since each state in $ADT_k$ has the same end index, they can be uniquely identified by their start index $i \leq k$ and their dotted rule (of which there are $O(|G|)$ unique possibilities). This gives a bound of $O(|G| \cdot k)$ possible unique states in $ADT_k$.

Every state in $ADT_k$ can have either a scan, prediction, or completion applied to it.

Scanning takes constant time because it simply requires checking the $k^{th}$ token of the input string. Thus scans on $ADT_k$ contribute $O(|G| \cdot k)$ operations.

Prediction over all states in $ADT_k$ can result in no more than $|G|$ new unique states, because there will be at most unique state for each production, the number of which is bounded by $|G|$. In addition to the total overhead of adding $|G|$ unique states, we also have the overhead of processing all $O(|G| \cdot k)$ states in $ADT_k$. This results in $O(|G| \cdot k + |G|) = O(|G| \cdot k)$

Completing a state with start index $i$, in the worst case will result in advancing every state in $ADT_i$. Thus if there are $O(|G| \cdot k)$ states in $ADT_i$, for $i \leq k$, then completing all the states in $ADT_k$ can result in $O(|G|^2 \cdot k^2)$ operations.

Thus run-time is dominated by completions. If we sum the contributions of completing the states in $ADT_k$ over all $k \leq n$ we get $O(|G|^2 \cdot n^3)$ operations. □

Note that introducing token noise skipping doesn't change the number of possible unique states on any $ADT_k$ because we haven't changed anything about the states themselves, just the way they relate to each other during the algorithm. The main differences induced by allowing skipping $w$ tokens between explained tokens are the following:

1. Each scan now takes $w$ operations because we must search $w$ tokens. This contributes $O(|G| \cdot w \cdot k)$ overhead
2. Prediction takes $O(|G| \cdot k + |G| \cdot w)$ operations because each of the $|G|$ new unique states can be placed on any of $w$ possible charts. The number of states on $ADT_k$ is still bounded by $|G| \cdot k$.
3. When completing a state with start index $i$ we now run the risk of completing all the states in charts $ADT_{i-w}, ..., ADT_i$. This means each completion for a state in $ADT_k$ may result in as many as $O(|G| \cdot k \cdot w)$ operations, meaning that all completions in $ADT_k$ could take $O(|G|^2 \cdot k^2 \cdot w)$ steps.

The modified run time of completions still swamps the overhead of scanning and prediction, resulting in an overall run time of $O(w \cdot n^3)$, thus in the general case, of $w = n$, we get a quartic run time.

I would like to note that my algorithm, per the previous chapter, also uses priority queue sfor each $ADT$. Since there are $O(n^2)$ states and each one is enqueued and dequeued just once, subbing in a priority queue adds $n^2 \cdot log(n)$ overhead, leaving the overall $O(w \cdot n^3)$ run time unchanged.

### 3.9 Space complexity

Earley noted that for just performing recognition it is possible to achieve $O(n^2)$ memory consumption but in order to extract parses one requires $O(n^3)$. As is shown in [38]'s algorithm for SPPF construction, this is so because reconstructing the parse forest requires remembering every completion used in the algorithm in order to produce a successful parse. Concretely, we end up producing an edge in the predecessor/reduction lists for every completion.

Since our modification completes a multiplicative factor $w$ more completions, our memory consumption must also be $O(w \cdot n^3)$

**Algorithm 8** Noise-skipping Earley algorithm

$topological\_ordering \leftarrow$ Map each nonterminal symbol to its number in the topological sort described in chapter 1
$oov\_index\_map \leftarrow$ Map the start index of an OOV span to its end index
$oov\_index\_reverse\_map \leftarrow$ Map the start index of an OOV span to its end index
$unique\_states \leftarrow$ Keep track of all unique states produced so far in order to properly merge multiple derivations of the same state
$w \leftarrow$ Bound on the number of tokens a parse can skip
$Chart \leftarrow ADT[n+1]$ fill the chart with n+1 $ADT$s
$start\_state \leftarrow (root,\ [@,\ S],\ [1,\ 1])$
AddToChart($start\_state, Chart[1]$)
**for** $k = 1$ to $n + 1$ **do**
    $ADT_k \leftarrow Chart[k]$
    $next\_state \leftarrow ADT_k.dequeue()$
    ApplyOperation($Chart, next\_state$)
**return** $(root,\ [S,\ @],\ [1,\ n+1]) \in Chart[n+1]$
**procedure** ADDTOCHART(state, ADT)
    **if** $state \in unique\_states$ **then**
        $prior\_state = unique\_states[state.hash]$
        $state.preds = state.preds \bigcup prior\_state.preds$
        $state.reds = state.reds \bigcup prior\_state.reds$
        $Attrs = \{state.attributes, prior\_state.attributes\}$
        $state.attributes = argmax_{x \in Attrs}(utility\_function(x))$
    **else**
        $unique\_states[state.hash] = state$
    $ADT.enqueue(state)$

**Algorithm 9** Noise-skipping Earley algorithm; cont'd

```
procedure APPLYOPERATION(Chart, state)
    if state.dot_at_end() then
        NoisyComplete(Chart, state)
    else if state.symbol_after_dot is terminal then
        NoisyScan(Chart, state)
    else
        NoisyPredict(Chart, state)
procedure NOISYSCAN(Chart, state)
    indices ← GetForwardIndices(state, state.end_idx)
    for idx ∈ indices do
        idx ← state.start_idx
        symbol ← state.symbol_after_dot
        if input[idx] = symbol then
            st ← (symbol, , [symbol, @], [idx, idx + 1])
            AddToChart(st, Chart[idx + 1])
procedure NOISYPREDICT(Chart, state)
    indices ← GetForwardIndices(state, state.end_idx)
    for idx ∈ indices do
        symbol ← state.symbol_after_dot
        for rule ∈ Grammar[symbol] do
            st ← (rule.LHS, @ ∘ rule.RHS, [idx, idx])
            AddToChart(st, Chart[idx + 1])
procedure NOISYCOMPLETE(Chart, state)
    end_idx ← state.end_idx
    start_idx ← state.start_idx
    LHS ← state.LHS
    predecessor_states ←GetCompleted(state, LHS)
    for predecessor_state ∈ predecessor_states do
        successor ← clone(predecessor_state)
        successor.increment_dot
        successor.end_idx ← end_idx
        AddToChart(successor, Chart[end_idx])
        if LHS is a nonterminal then
            reduction_label ← state.end_idx
            successor.red[reduction_label].append(state)
            if predecessor_state.dot_index > 0 then
                predecessor_label ← predecessor_state.end_idx
                successor.pred[predecessor_label].append(predecessor_state)
        else
            label ← state.start_idx
            successor.pred[label].append(predecessor_state)
```

# Chapter 4

# Building an ordered shared packed parse forest

### 4.1 Abstract

In the previous chapter, I introduced the standard Earley algorithm and Scott's [38] predecessor and reduction relations she uses to construct the parse forest. I then introduced novel extensions to the algorithm to enable it to a.) handle skipping noise and b.) maintain a vector of attributes (and the accompanying utility score) belonging to the optimal possible derivation of each state. Chapter 2 defined the types of attributes and utility functions for which the algorithm in chapter 3 is valid. In this chapter I will show that by associating each state with the attributes and utility scores of their best derivation, we can construct a data structure I call an **ordered SPPF** which has the property that the best parse can be extracted in constant time and the $k$ best parses can be extracted in $poly(k)$ time. This property is extremely useful because even though standard SPPF's are polynomial in the size of their input, the number of trees they represent can be exponential, therefore it would not be enough to simply extract all the parse trees and then rank them post-hoc.

Before we are ready to discuss in-order parse extraction from the **ordered SPPF** I will first introduce Scott's original algorithm for constructing a standard SPPF and I will show the modifications necessary to handle noise-skipping as well as the modifications necessary to make an SPPF ordered. The next chapter will take up the subject of in-order parse extraction in polynomial time.

### 4.2 Introduction

In the previous chapter I introduced the predecessor and reduction relations which are computed at run time in order to enable construction of the full parse forest for an Earley parser. In this section, I will described how exactly those relations are used by [38] to build an SPPF corresponding to all and *only* the parses possible for the input. The interested reader should consult [38] and [37] directly to get a detailed account of how SPPFs look - for our purposes I will describe them simply as a generalization of binary trees in which parent-child relations are represented with directed hyperedge from parent node $u$ to a set of children $w, v$ called a family. Additionally, nodes can have multiple incoming and outgoing hyperedges and there can be cycles in the graph. An simple SPPF is shown below for an expression grammar parse over input '1 + 1 + 1 + 1'. Each node with text is an SPPF node, either corresponding to a symbol and a span or an incomplete rule and a span. The circular nodes represent families. Families contain pointers to 1 or 2 child SPPF nodes. If a node has multiple families it can be realized via multiple distinct combinations of direct children. For example, the first and second families of (E,0,5) correspond to a right associative and left associative parse, respectively.

In the following section, I will present the modifications necessary to handle skipped noise and in the subsequent section I will show how to make the SPPF an ordered SPPF.

#### 4.2.1 The BuildTree procedure

[38] describes a recursive function BuildTree which, given an SPPF node $u$ and an Earley state $p$, populates $u$ with children, each of which is an SPPF corresponding to a derivation anchored at $p$ such that all possible subtrees anchored at $p$ are stored in $u$. In this way, calling it on a root node and sentential state will produce a complete SPPF for the input. The pseudocode below describes the BuildTree function of [38] in a notation consistent with the conventions of the previous chapter. For SPPF nodes, I use a notation $(A, i, j)$ where $A$ is either a symbol in the vocabulary or a partially completed rule, $i$ is the start index, and $j$ is the end index. Every time a new SPPF node is produced by the procedure below, we check a global store to see if it has been created already and if so we modify the existing copy directly, otherwise we create a new node altogether. The pseudocode represents this using the following shorthand $v \leftarrow (a, i, i+1)$.

Just as elsewhere in this thesis, I use capital letters to indicate nonterminal, lowercase to represent terminals, and greek letters to represent sequences of symbols in the vocabulary. $Family(\{w, v\})$ indicates the 'family" data structure described in [38] which represents a directed hyperedge from a parent to the set a set of child SPPF nodes $w$ and $v$.

In brief, this algorithm begins at the root state and recursively builds a tree for the current state $p$ and node $u$. There are four cases for the form the current state :

1. If there is only one item before $p$'s dot:
    (a) If that item is a terminal, create a leaf node for it and add it to $u$'s children

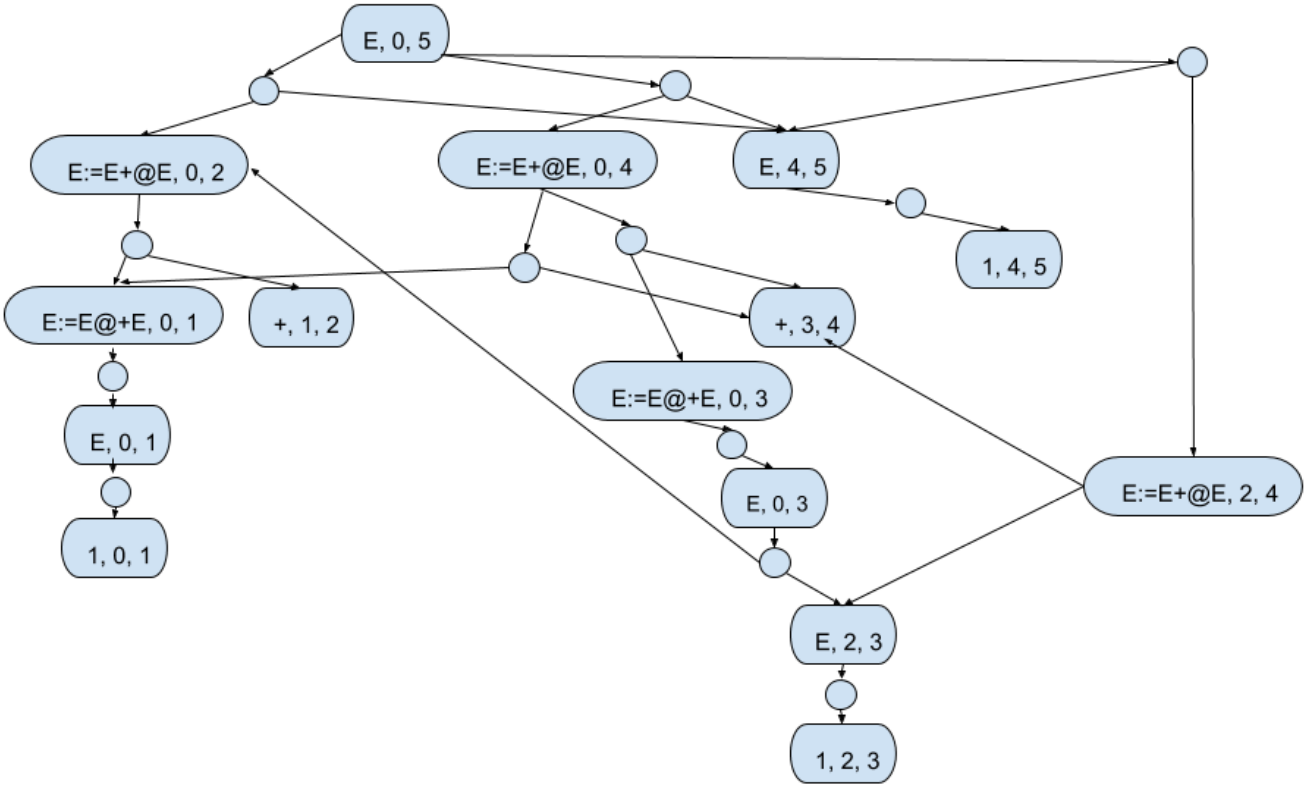



Figure 4.1: .

(b) If that item is a nonterminal, create a new internal node $w$ corresponding to the nonterminal and repeatedly call BuildTree on $w$ and each state deriving that nonterminal (accessible via the reduction relations for $p$). The reason we call BuildTree for every state is that since $v$ corresponds simply to a symbol $C$ and a span, there will be a different state for each rule parsing span $i, j$ whose LHS is $C$, each of which will have different children, and therefore require different SPPF nodes.

2. If the there are multiple items before $p$'s dot:

(a) If the immediately preceding symbol is a terminal, create a leaf node for it as well as an internal node for the its predecessor state, call BuildTree on that predecessor node and for every derivation of the predecessor state (accessible via the predecessor relations for $p$). Add a hyperedge from $p$ to the family consisting of the two nodes produced.

(b) If the immediately preceding symbol is a nonterminal, iterate through all derivations of the preceding symbol $C$ and create a different internal SPPF node for all derivations having different start/end indices (accesible via the reduction relations for $p$). Call BuildTree on each pair of Earley state and corresponding SPPF node. Additionally, iterate through all predecessor states ($p$

but with the dot moved to be before $C$). There will be different predecessor states corresponding to each different possible start index of $C$. Create SPPF nodes for those predecessor states via BuildTree. Finally, for each pair of SPPF nodes for reduction/predecessor pair, add a hyperedge (a.k.a. "family") from $p$ to that pair.

In this way, each family of a node $u$ corresponds to a way of deriving the symbol preceding the dot and the predecessor state corresponding to move $p$'s dot backwards by one index. This is why the family is a singleton if there is a single element preceding the dot, and why it is a tuple otherwise. Since it is possible to derive nonterminal symbols preceding the dot in multiple ways, via a different rule whose LHS is $C$ (and with different indices), we need to loop over possible way to derive the symbol before the dot (using the reductions) and every subsequent way to derive the predecessor state resulting from moving the dot backwards by one index. At the start of each invocation to BuildTree, we check whether an Earley state has been processed before to ensure we don't enter an infinite loop.

**Algorithm 10** Scott's SPPF construction algorithm

```
1:  procedure BUILDTREE(u, p)
2:      if p has been processed then
3:          return
4:      if p = (A, [a_i, @, β], [i, i + 1]) then
5:          v ← (a_i, i, i + 1)
6:          u.addChild(Family({v}))
7:      else if p = (A, [C, @, β], [i, j]) then
8:          v ← (C, i, j)
9:          for C_derivation ∈ p.reds[i] do
10:             BuildTree(v, C_derivation)
11:             u.addChild(Family({v}))          ▷ We construct a unique family for each state deriving C
12:     else if p = (A, [α, a_{j−1}, @, β], then
13:         v ← (a_{j−1}, j − 1, j)
14:         for predecessor_state ∈ p.predecessors(j − 1) do
15:             w ← (A := α@a_{j−1}β, i, j − 1)
16:             BuildTree(w, predecessor_state)
17:         u.addChild(Family({w, v}))
18:     else if p = (A, [α, C, @, β], [i, j]) then
19:         for label ∈ p.reds.labels do          ▷ Label is the start index of a derivation of C
20:             for C_derivation ∈ p.reds[label] do
21:                 v ← (C, label, j)
22:                 BuildTree(v, C_derivation)
23:                 for predecessor_state ∈ p.preds[label] do
24:                     w ← (A := α@Cβ, i, label)
25:                     BuildTree(w, predecessor_state)
26:                 u.addChild(Family({w, v})      ▷ We construct a unique family for each state deriving C
```

### 4.3 Noise-skipping SPPFs

In the previous chapter I introduced modifications to the labeling relations necessary to enable noise-skipping. In particular I noted that when completing a state $s$ with start index $j'$, if we produce a successor state $p$ from predecessor state $q$, with end index $j < j'$, by moving $q$'s dot forward, then we use $j'$ as the label for the reduction and predecessor edges, from $p$ to $s$ and $q$ as opposed to the end index of $q$, $j$. This modification only applies when there are more than symbols before the dot for $p$.

This suggests that the logic of the function above only needs to be changed for the third and fourth branches, in particular we have to take every instance where we've assumed that node $v$'s start index is identical to $w$'s end index. This happens on lines 13, 16, and 26 in the pseudocode below: On line 14, we now search over all predecessor relations, some of which may be labeled with $k < j - 1$. This occurs

**Algorithm 11** Noise-skipping SPPF construction

```
1: procedure NOISYBUILDTREE(u, p)
2:     if p has been processed then
3:         return
4:     if p = (A, [a_i, @, β], [i, i + 1]) then
5:         v ← (a_i, i, i + 1)
6:         u.addChild(Family({v}))
7:     else if p = (A, [C, @, β], [i, k]) then
8:         v ← (C, i, k)
9:         for C_derivation ∈ p.reds[i] do
10:            NoisyBuildTree(v, C_derivation)
11:            u.addChild(Family({v}))          ▷ We construct a unique family for each state deriving C
12:    else if p = (A, [α, a_j', @, β], [i, j' + 1]) then
13:        v ← (a_j', j', j' + 1)
14:        for j ∈ p.preds.labels do
15:            for predecessor_state ∈ p.preds[j]) do
16:                w ← (A := α@a_j'β, i, j)
17:                NoisyBuildTree(w, predecessor_state)
18:        u.addChild(Family({w, v}))
19:    else if p = (A, [α, C, @, β], [i, k]) then
20:        for j' ∈ p.reds.labels do                ▷ Label is the start index of a derivation of C
21:            for C_derivation ∈ p.reds[j'] do
22:                v ← (C, j', k)
23:                NoisyBuildTree(v, C_derivation)
24:                for j = max(i, j' − w); j ≤ j' do
25:                    for predecessor_state ∈ p.preds[j] do
26:                        w ← (A := α@Cβ, i, j)
27:                        NoisyBuildTree(w, predecessor_state)
28:                u.addChild(Family({w, v})        ▷ We construct a unique family for each state deriving C
```

when skip noise between $k$ and $j-1$ by moving the dot on a state $q$ whose end index is $k < j-1$. We also modify the fourth branch to account for the fact that now reduction edges have labels $j'$ potentially different than the labels, $j < j'$, of their predecessor states they worked with to produce the successor state. Since a predecessor state can be completed by any state whose $j' > j$ where $j'$ and $j$ don't skip more than the allowable tokens $w$, we use the loop constraints on line 24.

#### 4.3.1 Correctness of the labeling relations

In the previous chapter I noted the following generalization for how to produce the predecessor/reduction relations and how to label them during completion for the noise-skipping parser.

1. If $s = (a, [a_{j'}, @], [j', j'+1])$ has a terminal as its LHS then we still label the predecessor edge from $successor = (A, [\beta, a, @, \delta], [i, j'+1])$ to $predecessor\_state = (A, [\beta, @, a, \delta], [i, j])$ with label $j$ (where $j$ is the end index of $predecessor\_state$)

2. If $s = (C, [\alpha, @], [j', k])$ has a nonterminal as its LHS we label the reduction edge with $j'$and the predecessor edge with $j$.

As noted, in case 1, where the symbol preceding the dot in $successor$ is a terminal, uses $j$ as its label instead of $j'$. This case is handled in the third branch of NoisyBuildTree: since the state corresponding to the scanned terminal always has exactly the same start index, if we used $j'$ as the label all of our predecessor states would be stored under the same label. Though this isn't necessarily a problem, in theory, we would no longer be able to intuit the end index of $w$ from the label alone. Thus, the basic logic is that, since the

scanned token always has the same bounds, but the predecessor state doesn't, it is more parsimonious to base the label on the predecessor state's boundaries than the scanned token's.

In case 2, both the start index of $s$ (referred $C\_derivation$ in NoisyBuildTree) and the end index of $predecessor_s tate$ are different. The labeling procedure used above, where the reduction edge bears the label of its corresponding state's start index $j'$ and the predecessor edge bears the label of the corresponding state's end index $j$ ensures that each state used in a predecessor/reduction pair only occurs once in $p$'s predecessor/reduction lists. Furthermore, the loop on line 24 ensures we construct a family for every $C\_derivation/predecessor\_state$ that could've led to a derivation of $p$ - in particular, those where $j' - j \leq w$ and $j > i$.

### 4.4 Rank-ordering

While the modifications needed for building SPPFs robust to noise skipping were relatively minor - namely the changes to the labeling relations from the previous chapter, and the addition of extra looping mechanisms to check predecessor's whose end indices aren't the start index of the reduction state - the modifications necessary to produce an ordered-SPPF are somwhat more substantial - relying both on the introduction of the completion metadata mentioned at the end of last chapter and modification to the simple "set" architecture of the Family data structure of [38].

Before detailing the innovations necessary to make an SPPF ordered, I will briefly define an ordered SPPF.

In an ordered SPPF, each node $u$ has associated with it, a best Earley state $p$, where best is determined by applying to the utility function to the attribute vector of its best derivation. Thus $p$ represents the Earley state whose best derivation is better than that of all other Earley states associable with $u$. This is a non-vacuous requirement because an SPPF node $u$ can either correspond to a nonterminal and a span, or an incomplete rule and a span. In the former case, any rule whose LHS was that nonterminal could derive the node, meaning there will be one Earley state for each such derivation.

Additionally, we recursively require that all of $u$'s families $Family(\{w, v\})$ have SPPF nodes who also have a best Earley state associated with each. Finally, we require that the child list of $u$ is ordered such that all of its child families are listed in decreasing utility of the best parse possible from deriving $u$ via $w$ and $v$.

#### 4.4.1 Creating families

A node $u$'s family consists of either one or two SPPF nodes. There is only one SPPF node $v$ in the family if there is only one symbol before $u$'s dot. If there is more than one symbol preceding the dot for $u$, then there will be one family node corresponding to the predecessor state and another corresponding to the reduction state.

As noted above, each node in a family $w, v$ will have a best Earley state associated with it, call them $p_w, p_v$ whose *best* derivation is the best of all possible derivations of all other Earley states associated with $w$ and $v$. If you recall, in the previous chapter we stipulated that while completing a state $s$, every time we produce a successor state $successor$ from a $predecessor\_state$, we will map the pair $(s, predecessor\_state)$ to the attributes of the resulting $successor$, and due to our choice of $ADT$, as explained in chapter 2, we could always be certain that those attributes were the attributes of the best possible derivation of $successor$ via $s$ and $predecessor\_state$.

Thus, while creating family $F$ consisting of the set $w, v$ we can simply set the attributes to be the result stored in the completion metadata store for tuple $p_w, p_v$. While creating a family from a singleton state $v$, we can simply use the attributes of $v$ directly to set the best attributes for that family. See algorithm 12.

#### 4.4.2 Adding children

In order to add children to an ordered SPPF node we must choose an underlying data structure for the sorted child list. A sensible choice is a max-heap. This is so because we need to be able to maintain sort-order dynamically, and, as we will see, be able to handle modifying the priority of already existing heap elements. Thus the addChild function referenced in the NoisyBuildTree pseudocode is as simple as constructing the family using the constructors above and adding the family a to a max-heap whose ordering predicate is the utility of the family.

**Algorithm 12** Ordered Family construction

```
1: procedure FAMILY(v)
2:     this.members = {v}
3:     this.states = {v.state}
4:     this.attributes = {v.state.attributes}
5:     this.utility = utility(this.attributes)
6: procedure FAMILY(w,v)
7:     this.members = {w, v}
8:     this.states = {w.state, v.state}
9:     this.attributes = completions_metadata(this.states)
10:    this.utility = utility(this.attributes)
```

#### 4.4.3 Updating families

Finally, we must be sure to update families when better attributes become available for them. In the pseudocde for NoisyBuildTree, we always simply added new families to the child set of each node, but now we must update the attribute values of families if a less optimal copy of the family already existed in the child list. Thus a necessary modification to our code is that instead of merely adding the new family to the child set of the current family, we check to see if a family with same signature exists, extract it, merge it with the new one using the UpdateFamily procedure below and store the result in the child list. As a reminder, a family is uniquely identified by the names of its members (either a symbol or a rule with dot) plus their indices.

**Algorithm 13** Updating families

```
1: procedure FAMILY(u,child_nodes)
2:     f' ← Family(child_nodes)
3:     for f ∈ u.children do
4:         if f = f' then
5:             if f'.utility > f.utility then
6:                 u.children.remove(f)
7:                 u.children.add(f')
```

### 4.5 Correctness

Below is pseudocode for our complete method OrderedNoisyBuildTree Note that the UpdateFamily method only updates a family's attributes when that family is reinstantiated with better attributes. One might be concerned that it could be possible for a child node $w$'s best attributes could be updated after its parent $u$ has updated the families containing $w$ for the last time.

There are two ways this could happen:

1. While deriving SPPF node $u$, we produce a child state $w$ (or $v$), and we never encounter the best state deriving $w$ and $v$ before Family({w,v}) is added to $u$ for the last time. This would occur, for example, if the loops on line 29/30, or the loops on line 20/21 of OrderedNoisyBuildTree terminated before encountering all possible states deriving $w/v$.

2. While deriving SPPF node $u$, we produce child state $w$. While building the tree anchored at $w$ using a suboptimal state $p'_w$, we build a child $u'$, which during its thread of execution results in an edge pointing back to $w$ (in other words there is a loop in the SPPF that's encountered before all derivations of $w$ are completed).

The first case never happens because the predecessor/reduction lists will contain all states deriving SPPF nodes $w/v$. This means that before $u$ leaves scope, the loops on 29/30 and 20/21 will encounter every state. This is guaranteed by the correctness of the parsing algorithm - it is guaranteed to find all possible derivations permissible on the input.

The second case could only ever happen if there were a unary cycle in the grammar, which we declared invalid for the use of utility functions in chapter 2. This is so because the existence of a path $u \rightarrow w \rightarrow u' \rightarrow^* w$ where w has the same symbol and same indices in both occurrences would mean that there is a unary cycle in the grammar beginning and ending with the LHS symbol of $w$.

### 4.6 Time complexity

In [38] and [39] it is proven that the standard SPPF construction algorithm is $O(n^3)$ and takes $O(n^3)$ space. I do not refer the reader to the proofs in the original material because they are confusing and difficult to map directly onto the terminology of this thesis. An alternative proof of this statement is laid out below.

**Theorem 4.** *Scott's SPPF construction algorithm 'BuildTree' has $O(n^3)$ time and space complexity.*

*Proof.* As noted in the previous chapter and shown in [18], [19] and [2], there are $O(|G| \cdot n^2)$ states produced by the Earley algorithm.

The BuildTree procedure produces one SPPF node for each Earley state, meaning that there are $O(n^2)$ invocations to BuildTree. The worst-case run-time is then determined by the time it takes to loop over the predecessor/reduction edges for each node.

There will only ever be one unique predecessor per predecessor edge label $j'$. The reason for this is that a predecessor points from a state $p = (A, [\alpha, a, @, \beta], i, j)$ to a predecessor $q = (A, [\alpha, @, a, \beta], i, j')$. Per the definition of a state, a state is uniquely defined by its LHS, its RHS, its dot location, its start index, and its end index. Since a predecessor to $p$ is merely a state $q$ with the dot index moved backwards by one, the LHS, RHS, dot location, and start index are all fixed, the only possible place of variation is the end index. Thus once we specify the predecessor edge label $j'$ emanating from state $p$ there will only ever be a single unique state $q$ pointed to by that edge.

There will, however, be as many as $O(|G| \cdot n)$ total reduction edges emanating from a state $p$ (summing across all labels, as we will see).

The logic for this follows straightforwardly from the proof at the end of the previous chapter. A state $p = (A, [\alpha, C, @, \beta], i, j)$ will have a reduction edge to, at most, every state capable of deriving a $C$ which ends at index $j$. That is, we will have one for all of the states of the form $C\_derivation = (C, [\delta, @], j', j)$. Note that there will be one such state $C\_derivation$ for every unique rule used to derive $C$ between indices $j'$ and $j$ and there may be any index $j'$ between $i$ and $j$, thus worst-case there are $O(|G| \cdot n)$ such states. You can get the same result by noting that $C\_derivation$ would've been on $ADT_j$ during parsing, but any particular $ADT_j$ is bound to have $O(|G| \cdot n)$ states on it, thus there could only be $O(|G| \cdot n)$ possible states $C\_derivation$ completing $p$.

Thus, in the fourth branch of the if-statement on lines 19-26 of algorithm 10, when we loop over all reductions, we will complete $O(n)$ steps to that loop. Once we have specified a start index, $label$, for the state $C\_derivation$ deriving $C$ between $label$ and $j$, we will have uniquely determined the predecessor state whose start index is $i$ and whose end index is $label$. Thus the loop on line 23 only actually occurs once.

Thus the overall run time is $O(n^3)$ as shown in [38] □

The run time of the noise-skipping ordered-SPPF construction algorithm differs in only 3 ways:

1. A state $C\_derivation$ deriving $C$ as a reduction of $p = (A, [\alpha, C, @, \beta], i, k)$ could've existed on $ADT_{k-w}, ..., ADT_k$ thus there are $O(|G| \cdot w \cdot k)$ possible states. Alternatively, the state is uniquely determined by the rule used to derive $C$, the start index $j'$ and the end index of the state which must fall between $k - w$ and $k$.

2. When enumerating the predecessors corresponding to a particular reduction, rather than there being a unique predecessor for $C\_derivation$ with end index $j'$, there may be one for each $j' - w, ..., j'$, because the predecessor state $predecessor\_state = (A, [\alpha, @, C, \beta], i, j)$ may have any end index in with $j' - j \leq w$.

3. At every step of the algorithm, rather than performing constant-time operations, we are modifying a priority queue.

The first two points mean that there are now $O(w^2 n^3)$ total steps in the algorithm. But the third point means that we modify a priority queue each time, yielding a run time of $O(w^2 n^3 \cdot log(w^2 n^3)) = O(w^2 n^3 \cdot log(wn))$. Note that this is asymptotically worse than the $O(w \cdot n^3)$ run time of the parser.

**Algorithm 14** Noise-skipping ordered SPPF construction

```
1:  procedure OrderedNoisyBuildTree(u, p)
2:      if p has been processed then
3:          return
4:      if p = (A, [a_i, @, β], [i, i+1]) then
5:          v ← (a_i, i, i+1)
6:          if Family({v}) ∈ u.children then
7:              UpdateFamily(u, {v})
8:          else
9:              u.addChild(Family({v}))
10:     else if p = (A, [C, @, β], [i, k]) then
11:         v ← (C, i, k)
12:         for C_derivation ∈ p.reds[i] do
13:             NoisyBuildTree(v, C_derivation)
14:             if Family({v}) ∈ u.children then
15:                 UpdateFamily(u, {v})
16:             else
17:                 u.addChild(Family({v}))
18:     else if p = (A, [α, a_j', @, β], [i, j'+1]) then
19:         v ← (a_j', j', j'+1)
20:         for j ∈ p.preds.labels do
21:             for predecessor_state ∈ p.preds[j]) do
22:                 w ← (A := α@a_j'β, i, j)
23:                 OrderedNoisyBuildTree(w, predecessor_state)
24:         if Family({w, v}) ∈ u.children then
25:             UpdateFamily(u, {w, v})
26:         else
27:             u.addChild(Family({w, v}))
28:     else if p = (A, [α, C, @, β], [i, k]) then
29:         for j' ∈ p.reds.labels do
30:             for C_derivation ∈ p.reds[j'] do
31:                 v ← (C, j', k)
32:                 OrderedNoisyBuildTree(v, C_derivation)
33:                 for j = max(i, j' − w); j ≤ j' do
34:                     for predecessor_state ∈ p.preds[j] do
35:                         w ← (A := α@Cβ, i, j)
36:                         OrderedNoisyBuildTree(w, predecessor_state)
37:                 if Family({w, v}) ∈ u.children then
38:                     UpdateFamily(u, {w, v})
39:                 else
40:                     u.addChild(Family({w, v}))
```

# Chapter 5

# In-order k-best parse extraction

### 5.1 Abstract

In the previous chapter I described how to use the results of the noise skipping Earley parser to produce an ordered SPPF which would be well-suited to fast, in-order extraction of the $k$ best trees. The only ordering imposed in the previous chapter was that each for each node in the forest $u$ with family set $\mathbb{F}$, each family $f \in \mathbb{F}$ has a vector denoting the attributes of the best possible subtree derivable from $f$ and that all families in the family set are ordered by the utility of their best subtree.

In this chapter, I will first show a motivating example of why in-order, k-best extraction is useful even for very simple grammars, then I will provide 3 algorithms: $O(n)$ top-1 tree extraction, brute force k-best extraction, and $kn^2 \cdot log(kn)$ k-best extraction.

This work presents a major contribution to this field as it is the only work addressing polynomial time trees from a parse forest built using utility functions. This is in part because work in the field has primarily focused on dealing with optimality on the front-end, that is, designing the parsing algorithm or the forest building algorithm itself to extract best trees, or work using approximation methods to generate approximate $k$-best result sets.

The model presented in this paper is much more robust than such alternatives because it presents a provably exact way to extract best trees from a static, non-modified forest object. Thus, rather than the design parameter "k" determining the shape of the results from the beginning of the parse, my method allows the same SPPF object to be queried many times with different $k$'s and always give correct results. Additional benefits include the fact that modifying the utility function is straightforward and simply requires a new parse of the input, but otherwise no changes to the algorithm.

### 5.2 Worked example

Consider the simple expression grammar below:

$$E := E + E$$
$$E = 1$$

This toy grammar is an excellent example in for our setting because the use case of expression grammars is ubiquitous and it produces highly ambiguous parses due to associativity of addition. Ignoring noise-skipping for now, the number of parses for an expression in the language is $C_n$ were $n$ is the number of 1s and $C_n$ is the $n^{th}$ Catalan number, which is exponential in $n$ - by the time we've reached 25 1s we are already past $10^{13}$ possible parses. The blow up in the number of parses is even more astounding when one permits noise-skipping, as the number of parses then becomes $C_n + \binom{n}{1}C_{n-1} + \binom{n}{2}C_{n-2} + ....$

An example SPPF for the input '1 + 1 + 1' is below (reproduced from chapter 4). There are four possible parses over the span 0-5: one that skips no tokens and is left-associative, one that skips no tokens and is right-associative, one that skips the second '1' and the second '+', and final parse that skips the second '1' and the *first* '+'.

In the figure, families are represented as unlabeled circular nodes containing pointers to their member nodes. You can extract a parse by recursively choosing one family per non-leaf node and all nodes from each chosen family. A node having multiple families indicates that there are multiple ways to realize that node. For example the first family of (E,0,5) indicates it can be realized by either a) skipping the second 1 and second + b) by skipping no tokens and using left-association or c) by skipping no tokens and using right-association. The two family nodes underneath (E:=E+@E,0,4) indicates that the first E can either include the first three tokens '1 + 1' or could just include the first token '1'. You can recognize that there are four possible parses by counting the number of ways to choose exactly one family for each node. There are three possible branches at the root node, one of which leads to (E:=E+@E,0,4) which has two possible branches. In the ordered-SPPF setting each node would have a set of attributes corresponding to the best possible subtree for that node and each family would have the attributes corresponding to merging the best possible subtrees for its member nodes. When expanding a node $n$ and choosing between the best family $f$ and a suboptimal family $f'$, I will show that it is possible to calculate the the attributes of $n$ resulting from choosing $f'$ over $f$ and using the best derivations of its members.

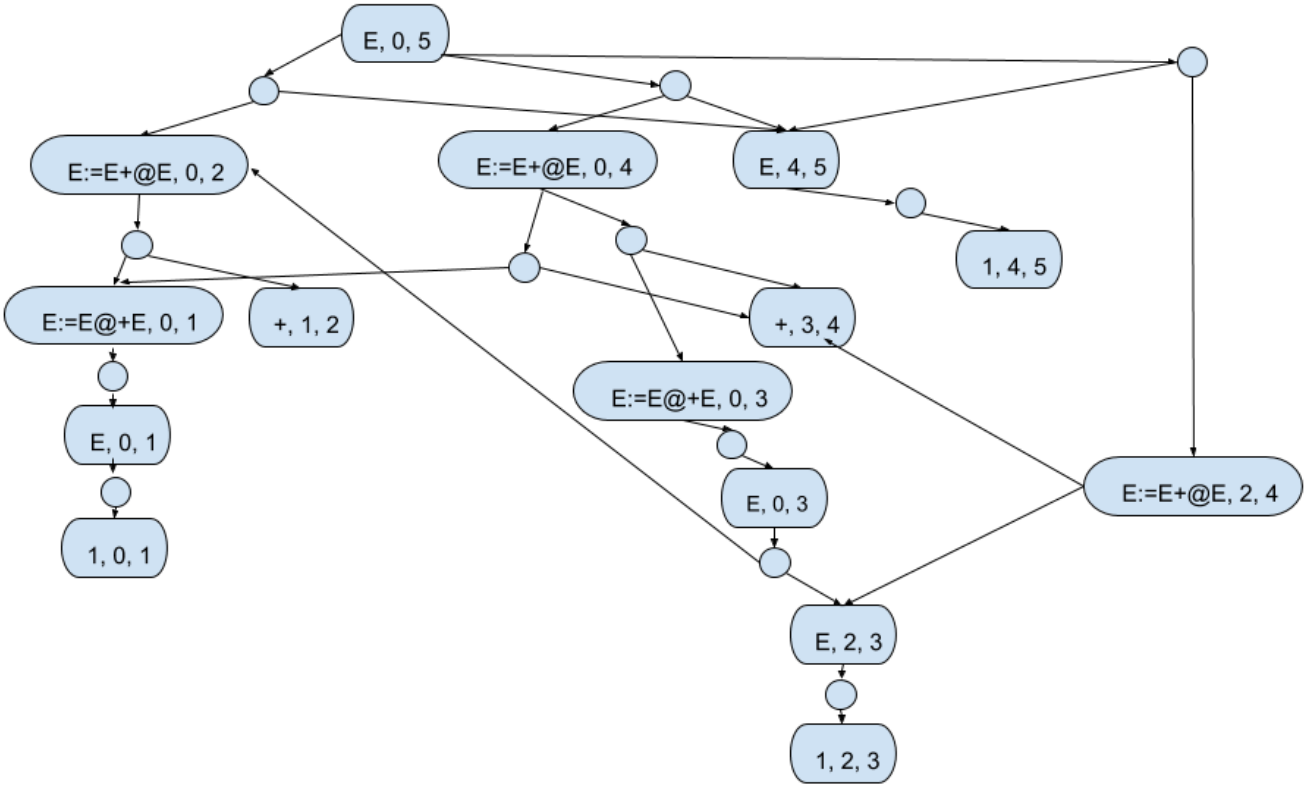



Figure 5.1: The four possible parses for '1 + 1 + 1' are represented in the SPPF above.

Extracting trees in order of some well-chosen utility function would aid us greatly in sorting through the madness because it obviates the (obviously intractable) need to extract all the exponentially many trees for our expression grammar and rank them post-hoc. In the noise-skipping case, something as simple as ranking trees in order of tokens explained would give all $C_n$ complete parses before any of the lesser ones. Furthermore, a utility function equal to the different between the depths of the right and and left hand sides of a tree could make the $+$ symbol right associative.

[37] offers solutions to this problem which removes parses from the tree, and other parsing algorithms such as LR parsers can omit certain dispreferred parses from the result set a priori, but neither of those solutions have the generality of the utility functions from the previous chapter. Thus, it is apparent that in order in-order tree extraction which is polynomial in the number of trees desired is absolutely essential for making general context-free grammars workable. Especially so if we would like to have relatively general utility predicates, have an easily tunable "k", and be able to reparse with a new utility function easily (this last property would even enable to possibility of learning our utility function).

### 5.3 Algorithms

As mentioned in the abstract, the key property of the ordered SPPF enabling in-order extraction is that each family in a node's family set has a vector denoting the attributes of the best possible subtree derivable from it. As discussed in the previous chapter, maintaining this property is possible so long as there no loops

in the forest. An additional computational property aiding in-order extraction is that each nodes list of families is presorted in decreasing order of utility by a priority queue.

This mode of organization lends itself to a straightforward and fast algorithm for extracting the (not necessarily unique) "best" tree from the forest. I call this the O(n) best tree algorithm

### 5.3.1 O(n) best tree algorithm

This algorithm is O(n) in the sense because it runs solely in the time requires to print a tree, which will be of size $O(n)$ since the trees are binary trees with $n$ leaves.

Due to the ordering properties of the SPPF the O(n) best tree algorithm is very simple:

**Algorithm 15** O(n) best tree extraction

```
1: procedure BESTTREE(root)
2:     t ← Tree(root)
3:     if |root.children| > 0 then
4:         best_family ← root.children.peek()
5:         for child ∈ best_family.members do
6:             t.addChild(BestTree(child))
```

Where $Tree(root)$ is an invocation of a constructor that produces a tree-node description from an SPPF node. The nature of the constructor is a design decision based on what form you want your parse trees to take.

### 5.3.2 Brute force in-order extraction

This algorithm consists of walking the entire forest, building each and every tree and sorting the results by utility. If $d$ is the number of trees, this takes $O(d \cdot log(d))$ time because we need to extract each tree (which is possible in time proportional the number of trees) then sort them.

In the case of the expression grammar this run time is $O(n \cdot e^n)$. Though this run time is pretty abysmal, it has relatively low overhead, and for inputs with few parses it is a viable solution. Additionally, if you would like ALL the trees in ranked order, then this algorithm is far superior to the one in the next section.

Overall the algorithm has a very similar flavor to many reinforcement learning algorithms which use the abstraction of an optimal policy $\pi$ and calculate the cumulative expected reward of taking (potentially) suboptimal action $a$ at time $t$ and then following that optimal policy thereafter.

Similarly in the brute force extraction algorithm, we build each tree via a recursive thread of execution. Along a particular thread of exectuion, we maintain the attributes of the parse we would get if we only took optimal paths from thereon out, and then every time we take a suboptimal family $f$, we generate each derivation of its component members $d$ (for which the best derivation of $d$ may be even more suboptimal than the best derivation of $f$) and determine the attributes that would result from choosing $d$ and taking best derivations from thereon out. Since this is applied recursively, we build the attribute vectors from the leaf nodes up (for whom there is only one derivation, and therefore their attribute value is certain) leading us to get a correct attribute vector for each.

Walking the forest can be done via a simple recursive procedure in which every time we encounter an SPPF node with multiple families, we spawn a new recursive call (producing a different subtree) for each family.

Since the attributes associated with each SPPF node are those of the best possible derivation, they are based on the assumption that all descendant child nodes will be derived with their optimal family. Determining the attributes associated with each (suboptimal) tree, therefore, requires associating each thread of execution with the attributes that derivation would have if only 'best' child families were taken for the remainder of execution and then, every time a suboptimal family is taken, determining how that choice alters the attributes of the best derivation.

This functionality is accomplished via the AlternativePathValue() method. AlternativePathValue takes the attributes of the current tree if only best families were chosen from the current node on, $current\_attributes$, additionally takes the attributes of the best family $best\_attributes$, as well as the attributes of the alternative family we are recursing on $alternative\_attributes$. The implementation can be specific to the attributes used

for your parser. For example, in the noise-skipping, attributes such as noise skipped and tokens skipped can't be calculated by simple addition and subtraction of values if the spans of different families are non-identical. In general, however, if the pooling function was the sum pooling function, one can simply use vector addition and subtraction. The pseudocode below assumes only standard vector addition and subtraction.

While constructing the trees for SPPF node $u$, recursing on a child family $f$ will produce different possible subtrees for each of the members $\{w, v\}$ of the family. In order to construct the trees of $t$ we need to find all the ways to take a derivation of each of the children. If children one has two derivations and child two has five derivations, this results in 10 unique trees for $u$. Note that this corresponds to taking the cartesian product of the derivation sets of child one and two. Additionally, just as we had to merge the attributes of descendants in a derivation using a pooling function, likewise we need to calculate the attributes for a tree by merging the attributes of its subtrees. Note that the attributes derived from the family using AlternativePathValue represent the best possible attributes for the overall parse if $f$ is chosen over the best family, this doesn't give us the attributes for the sub-optimal subtrees derivable from this family, of which there may be many. Thus, for any given combination of subtrees correspdonding to $f$, we must calculate the attributes of the resulting overall tree if those subtrees are chosen as the children of $u$.

Again, the calculation of the attributes is somewhat subtle, but relies on similar logic to that in AlternativePathValue(). MakeChildLists() takes in the node whose child subtrees we are deriving, $node$, the possible subtrees of the child family's component members, $family\_member\_derivations$, and the attribute vector of the best parse using family $f$, $best\_attributes\_f$. We then, for each derivation calculate the attributes of the overall best tree using that subtree.

#### 5.3.3 Continuation passing style (CPS) k-best extraction

This algorithm is capable of extracting the $k$ best trees in time proportional to $k$ as opposed to the total number of trees $d$. Though it has much higher overhead per tree returned than the brute force method, if $d \gg k$ then this method must be employed.

The algorithm is a somewhat unusual one, but it is essentially similar in character to the brute force algorithm, except it employs continuation passing style to explore execution paths and only evaluates execution paths in order, ranking unevaluated threads of execution in decreasing order of utility - eliminating the need to generate *all* trees in the forest. These constraints alone won't give polynomial performance, however, because if we don't intelligently queue up threads of execution then we could end up queueing all possibilities, which would still result in run time proportional to the forest size, thus it is necessary that we only queue up a polynomial number of unexplored paths in the duration of our execution.

That said, we will still invoke similar concepts as in the previous algorithm, such as the ability to calculate the best possible attributes accessible via choosing a certain family $f$ at time $t$ and then proceeding to only choose best families from thereon out. Additionally, we will use the AlternativePathValue and MakeChildLists methods from earlier.

Our ability to only queue a subset of the forest's trees relies on the fact that ambiguities (represented by different families underneath a node $u$) are ordered in terms of best possible utility accessible by completing the parse from thereon out. This allows us to, while generating a tree from family $f$, queue up alternative derivations using family $f'$ over $f$ before queueing up alternatives choosing $f''$ if $f'$ has a better optimal outcome than $f''$.

I will now introduce and define some terminology, show the algorithm, then prove its correctness and upper bound its run time.

**Formal definitions**

**Definition: 12** A **tree** $t$ is a single parse instance taken from an SPPF. Trees from an SPPF are binary trees where each node is either a rule node or a symbol node. A symbol node $(S, i, j)$ has a symbol from the vocabulary and a span. The symbol node is either a leaf node and corresponds to a terminal or it is an interior node with a nonterminal symbol. In the latter case the node has at most two children (one child in the case of a unary rule, two otherwise) corresponding to the rule used to expand it in the parse, $S \rightarrow \alpha\beta\gamma$, where the left child is always $(\alpha\beta@\gamma, i, j')$ and the right child is always $(\gamma, j', j)$.

**Definition: 13** We can describe a tree $t$ by listing the SPPF nodes involved and indicating which family was chosen for each node. If each node is given an index $n^0, ..., n^j$. Then we will indicate the family chosen

**Algorithm 16** Brute force forest extraction

```
1: procedure BRUTEFORCEFOREST(node, best_attributes)
2:     result_list ← []
3:     if |node.families| = 0 then
4:         tree ← Tree(node)
5:         result_list.add(tree)
6:     else
7:         best_family ← node.families.peek()
8:         for f ∈ node.families do
9:             best_attributes_f ← AlternativePathValue(best_family, f, best_attributes)
10:            family_member_derivations ← []
11:            for child ∈ f.members do
12:                child_derivations ← BruteForceForest(child, best_attributes_f)
13:                family_member_derivations.add(child_derivations)
14:            MakeChildLists(node, family_member_derivations, best_attributes_f, results_list)
15:    return results_list
16: procedure ALTERNATIVEPATHVALUE(best_family, alternative_family, best_attributes)
17:                                                    ▷ Assume additive pooling function
18:    return best_attribute ⊖ best_family.attributes ⊕ alternative_family.attributes
19: procedure MAKECHILDLISTS(node, family_member_derivations, best_attributes_f, result_list)
20:    derivations ← CartesianProduce(family_member_derivations)
21:    for child_list ∈ derivations do
22:        tree ← Tree(node)
23:        derivation_attributes ← best_family_attributes
24:        for child ∈ child_list do
25:            if utility(child.attributes) < utility(best_family_attributes) then
26:                derivation_attributes+ = (child.attributes ⊖ best_family_attributes)
27:            if utility(derivation_attributes) < utility(best_family_attributes) then
28:                tree.attributes ← derivation_attributes
29:            else
30:                tree.attributes ← best_family_attributes
31:            result_list.add(tree)
```

to expand each $n^j$ as $f^j_{i_j}$. The indices $i_j$ are ordered such that family $f^j_{i_j}$ results in a better subtree for $n^j$ than $f^j_{i'_j}$ if $i'_j > i_j$.

**Definition: 14** A **tree expansion** *of an SPPF node* **n** is the tree resulting from creating a tree node for $n$ as well as tree expanding one of its child families $f$ chosen according to *some* selection policy

**Definition: 15** The **set** of **tree expansions** *of an SPPF node* **n**, $\mathbb{T}_n$ is the set of all possible trees derivable from $n$. Each two distinct SPPF nodes $n_1$ $n_2$ have completely distinct tree expansion sets, though two trees $t_1 \in \mathbb{T}_{n_1}$ and $t_2 \in \mathbb{T}_{n_2}$ may share some number of subtrees. This is obvious because technically a leaf node is a subtree and any two parse trees parsing the same tokens will share, at the very least, the same leaves.

**Definition: 16** A **tree expansion** *of a family* **f** results in up to two subtrees generated by expanding the SPPF node members of the family.

**Definition: 17** A **thread of execution, T** *anchored at family f* is the process, resulting in up to two subtrees, of expanding family $f$ under the policy of always expanding the best family for a node $u$ when multiple child families exist. I will denote the tree resulting from, using $[\![\mathbf{T}(f)]\!]$

**Definition: 18** An alternate family set $\{f^j_{i'_j}\}$ is a way of representing a tree, which specifies the set of suboptimal families taken for each node $n^j$. An alternate family set can be read as: "when you get to node $j$ take family $f^j_{i'_j}$ instead of $f^j_0$. We will usually say that this alternate family set is relative to the best tree

which takes all of the best families for each node $n^j$. Note that by taking a different family for node $n^j$ you may end up producing a tree which includes SPPF nodes which aren't in the best tree at all. This doesn't matter for the representation, which just focuses on which suboptimal families were chosen. The purpose is that the alternate family set represents a set of non-optimal choices taken during execution.

**Definition: 19** A **task, S** is a 6-tuple $(\mathbf{n}^j, \mathbf{f}^j_{i_j}, \mathbf{f}^j_{i'_j}, \mathbf{t}_{n^j}, \textbf{root\_attributes}, \textbf{alt\_family\_set})$ consisting of a primary family $\mathbf{f}^j_{i_j}$, an alternate family $\mathbf{f}^j_{i'_j}$, an origin SPPF node, $\mathbf{n}^j$, an origin tree $\mathbf{t}_{n^j}$, the attribute vector of the root node **root_attributes** resulting from finishing the task, and the alternate family set corresponding to the tree that will result from this task **alt_family_set**. A task can be thought of as a continuation, in the continuation-passing style sense, for a thread of execution and will ultimately result in a single tree. When generating a tree $t$ along a thread of execution of a thread of execution **T**, and an alternative for a family $f$ is encountered, a task can be queued up representing the result of replacing $[\![\mathbf{T}(\mathbf{f}^j_{i_j})]\!]$ with $[\![\mathbf{T}(\mathbf{f}^j_{i'_j})]\!]$ in $t$. Crucially, using root_attributes, we can determine the utility of the tree that was would result from completing task $S$ before even executing it, thus we can lazily execute tasks in order of utility. Additionally, a new child task will always have the alternate family set of the task that created it plus the family $f'$ whose alternate it is exploring.

- $\mathbf{n}^j$ is an SPPF node, found along the present thread of execution, which contains an ambiguous choice of families $\mathbf{f}^j_{i_j}$/$\mathbf{f}^j_{i'_j}$ the alternatives of which we would like to explore. As noted above, the task **S** will evaluate the expansion of the alternative family $\mathbf{f}^j_{i'_j}$ and graft the result in the context otherwise resulting from $[\![\mathbf{T}(f^j_{i_j})]\!]$. I call this grafting process $[\![\mathbf{T}(f^j_{i_j})]\!][t_{n^j}] \leftarrow [\![\mathbf{S}]\!]$ because we swap the results of the task into the tree produced by $\mathbf{T}f^j_{i_j})$ at subtree $t_{n^j}$

- $\mathbf{f}^j_{i_j}$ is the family being used to generate node $n^j$ in the thread of execution which was running when the task was created.

- $\mathbf{f}^j_{i'_j}$ is the alternative to $\mathbf{f}^j_{i_j}$ that will be expanding by evaluating **S**.

- $\mathbf{t}_{n^j}$, is the subtree into which task's result $[\![\mathbf{S}]\!]$ will be grafted. It corresponds to the context of $n^j$'s expansion in the original thread of execution **T** which spawned **S**

- **root_attributes** is the vector of attributes associated with the $[\![\mathbf{T}(f^j_{i_j})]\!][t_{n^j}] \leftarrow [\![\mathbf{S}]\!]$

- **alt_family_set** is a set of families representing all the suboptimal families taken to produce the tree resulting from the task. I will show later that the alt_family_set uniquely identifies a tree.

We don't necessarily need to store f, we simply need to pass in the differential value of choosing f' over f - something which can be calculated by calling AlternativePathValue at the time the task is produced.

**Useful theorems**

I will now show some useful properties that will help us derive an efficient algorithm for lazily extracting trees in order.

First, I will show that you can uniquely identify a suboptimal tree off a given thread by enumerating the families it chooses which differ from those chosen by the optimal tree on that thread, and additionally that this mapping is one-to-one

**Lemma 1.** *For a given thread (specified by a root node $n^0$ with choice of family $f^0_{i_0}$, see 5.3) any suboptimal tree $t_i$ in the tree expansion set for $n^0$ can be uniquely specified by listing the suboptimal families it chooses for each of its nodes (note that even if no suboptimal families are chosen then that empty set denotes the best possible tree in the expansion set). This was defined as the 'alternate family set' in the previous section.*

*Additionally no two alternate family sets map to the same tree. Thus there is a bijection between alternte family sets $\{f^j_{i'_j}\}$ and trees in the tree expansion set of $n^0$.*

*Proof.* First I will show that a set of alternate families $\{f^j_{i'_j}\}$ uniquely specifies a tree $t_i$ in the tree expansion set of $n^0$. This is very clearly true because by definition the best tree in the thread will be produced by taking $f^0_i$ for the first family and then taking optimal families for all children thereafter.

In general, when expanding some tree $t_i$ we take an optimal family $f^j_0$ for node $n^j$ unless otherwise specified, so in order to specify how to produce $t_i$ it suffices to say which families besides $f^0_i$ and $\{f^j_0\}$ we took because by default those families which are left unspecified will be the best family available for each node. It is important to note that there will be no ambiguity because no node will ever be encountered more than once in the same tree (this is due to the fact that we've excluded unary cycles). Additionally, families are a unique data structure belonging to a single node. Thus when we specify an alternate family set it is completely unambiguous which nodes in the tree we are referring to.

For the reverse direction, we must show that no two alternate family sets $\{f^j_{i'_j}\}$ can map to the same tree.

Since the mapping between family sets and nodes in the tree is unambiguous and since no node can appear twice, it suffices to show that no two distinct families for $f^j_i$ and $f^j_{i'}$ for node $n^j$ could result in the same subtree anchored at $n^j$. This is trivially true by the definition of an SPPF, where two distinct families will always have a distinct combination of SPPF node members and thus will result in unique pairs of child subtrees. □

We can generate other, less-optimal, elements of tree set $\mathbb{T}_{n^0}$ by either changing $f^0_{i_0}$ to $f^0_{i'_0}$ for $i' > i$ (that is, by changing the family used for $n^0$) or by changing the families $\{f^j_{i_j}\}$ used to expand descendant nodes $n^j$ to $\{f^j_{i'_j}\}$ where $i'_j > i_j$ (that is, changing the families used for one or more of $n^{0\prime}s$ descendants). Again, this corresponds to the notion of the alternate family set above, which can be viewed as an instruction manual for how to make suboptimal trees: "when you get to node $n^j$ take $f^j_{i'_j}$ instead of $f^j_0$, e.g.

Another important lemma is below

**Lemma 2.** *Assume the tree expansion set of $n^0$ has more than 1 tree. The utility of trees in the expansion set of $n^0$ is monotonically decreasing in the number of families by which those trees differ from the best tree for $n^0$. That is any tree resulting from alternate family set $\mathbb{F}_s$ is always worse than a tree resulting from an alternate family set $\mathbb{F}'_s$ if $\mathbb{F}'_s \subset \mathbb{F}_s$.*

*Proof.* This is a trivial consequence of our definitions of the utility function and pooling function of chapter 1. The utility function is monotonic over the pooling function so taking two suboptimal families will always result in a greater diminishment to the utility than taking only a single suboptimal family. □

This leads to two useful corollaries:

**Corollary 1.** *If we are producing the best tree for a node $n^0$ then the next best tree differs by only one family.*

*Proof.* This is a trivial consequence of 2. □

**Corollary 2.** *If we are producing the $i^{th}$ best tree and the largest alternate family set so far for a tree $t_{i'}$, $i' < i$ has $k$ elements, then the alternate family set of $t_i$ has at most $k+1$ elements.*

*Proof.* This is a trivial consequence of 2. □

**Corollary 3.** *If we have a list of the top $k$ trees for node $n$, $\{t_1, ..., t_k\}$. The alternate family set of tree $t_{k+1}$ will be equivalent to the alternate family set of one of the trees $t_i$ with the exception of either swapping one family $f^j_{i_j}$ for another $f^j_{i'_j}$ or by adding a suboptimal family for a new $j$, $f^j_{i_j}$. That is, it differs by at most one from the existing alternate family sets so far.*

*Proof.* This is a consequence of monotonicity 2.

Assume by contradiction that $t_{k+1}$'s alternate family set differs by at least two families from all other alternate family sets for tree $t_1, ..., t_k$. By monotonicity we could produce a better tree by removing one of those families from the set, and by the uniqueness of the alternate family set representation, this would result in a new tree $t'_{k+1}$ that is better than $t_{k+1}$ but isn't one of the trees in $\{t_1, ..., t_k\}$. Thus violating the assumption that $t_{k+1}$ is actually the $k+1^{st}$ best tree. □

**Suboptimal trees**

In order to understand how to use tasks to generate the set of tree expansions $\mathbb{T}_n$ for SPPF node $n$ it is crucial to understand the ways in which node $n$ could expand to different trees and the structural relationships giving rise to the different types of ambiguities.

Let's say we are on a thread of execution for node $n^0$ using family $f^0_{i_0}$, $\mathbf{T}_{f^0_{i_0}}$, which will result in tree (a) of figure 5.3. This tree parses 8 tokens (leaf nodes). It has two branches both of which have left-association. Assume left-association is preferred for this example grammar.

Furthermore, let's say that this tree was produced by we choosing family $f^j_i$ for each SPPF node along the route, $n^j$.

Let's consider all the ways that we could generate the tree set of $n^0$ assuming we are along a thread which would result in (a) of figure 5.3.

1. We could choose to modify just the family used for $n^0$. That is we could choose $f^0_{i'_0}$ over $f^0_{i_0}$, where $i'_0 > i_0$. This is shown in (b).
2. We could modify a single descendant node's family. That is we could choose $f^j_{i'_j}$ over $f^j_{i_j}$, where $i'_j > i_j$ for some $j > 0$. This is shown in (c).
3. We could modify two descendant nodes' families along the same branch. This is shown in (d), where switching $n^1$'s family from $f^1_{i_1}$ to $f^1_{i'_1}$ has given rise to a new child node $n^3$ and we have subsequently chosen a suboptimal family for $n^3$ as well $f^3_{i'_3}$.
4. We could modify two descendant nodes' families on different branches, where the shortest path to the closest common ancestor of the modified nodes is 1. This is shown in (e).
5. We could modify arbitrarily many nodes along multiple branches where the shortest path to the closest common ancestor of the modified nodes is 1. This is shown in (f), where we have chosen suboptimal families for both $n^1$ and $n^3$ (along the same branch) and for $n^2$ (along a different branch).
6. We could also modify sets of arbitrary cousins (i.e. where the shortest path to the common ancestor is longer than 1).This is shown in (g), where we have chosen suboptimal families for both $n^5$ and $n^4$. If we had a larger tree we could've shown three distinct subtrees, or four distinct subtrees all having a node modified. In general there can be any number of distinct subtrees with any number of suboptimal families chosen therein.

Clearly the first case can be handled by simply creating a new task $\mathbf{S} = (n^0, f^0_{i_0}, f^0_{i'_0}, t, root\_attributes)$, where $t$ is whatever context tree contains $n^0$ (in this case we can assume $n^0$ is the root and have $t$ be the empty node $[n^0]$, and root_attributes are calculated using the AlternativePathValue method from the previous section. We will refer to these as 'twin' tasks in the next section.

The second case can be handled by simply enqueuing a task $\mathbf{S}_0 = (n^j, f^j_{i_j}, f^j_{i'_j}, t, root\_attributes)$ where $t$ is the rest of (a) in figure 5.3 except for the subtree anchored at $n^j$. We will refer to these as 'child' and 'descendant' tasks in the next section. Thus (c) would result from a child task while producing (a).

The third case can be handled by the transitive closure of adding tasks for ambiguous children. For example, in figure (d) we would first enqueue $\mathbf{S}_1 = (n^1, f^1_{i_1}, f^j 1 i'_1, t_1, root\_attributes)$ then, when $S_1$ is executing we would add $\mathbf{S}_3 = (n^3, f^3_{i_3}, f^3_{i'_3}, t_3, root\_attributes)$, where $t_3$ would be the tree in (c) (or (d)) minus $n^3$. It is important to note that because of the lemma, it is okay that we don't enqueue $\mathbf{S}_3$ before $\mathbf{S}_1$ because we know that $\mathbf{S}_3$ will result in a worse overall tree than $\mathbf{S}_1$ so we don't need to process it first. We wil refer to these as 'child' and 'descendant' tasks in the next section, where a descendant task has spun off another descendant task. Thus, (d) will result as a child task while producing (c) which was itself a child task from (a).

The fourth case can be accomplished by, when a task starts, anchored at node $n^j$, we back up the tree and queue up tasks for every family of every sibling and grand-aunt to $n^j$. For example, in figure (e), we would first enqueue $\mathbf{S}_1 = (n^1, f^1_{i_1}, f^j 1 i'_1, t_1, root\_attributes)$ then, when $S_1$ begins execution we would add $\mathbf{S}_2 = (n^2, f^2_{i_2}, f^2_{i'_2}, t_2, root\_attributes)$, where $t_2$ would be the tree in (c) and (e) minus $n^2$. It is important to

note again, that because of the lemma it is okay that we don't enqueue $\mathbf{S}_2$ before $\mathbf{S}_1$ because $\mathbf{S}_2$ modifies two families, so it will always be worse than a tree that just modifies $n^1$. We will refer to these as 'sibling' and 'aunt' tasks in the next section. Thus (e) will result as a sibling task while producing (c), which was in turn a child task from (a).

The fifth case, which modifies two or more nodes along two or more paths, is a generalization of the third and fourth cases and will . In the example in (e), we will first create task corresponding to choosing an alternative derivation of $n^1$, then a task for choosing an alternate derivation of $n^3$, then finally a task for choosing an alternate derivation of $n^2$. Once again, the order that we have added them in is fine because a task which modifies just $\{n^1, n^3\}$ is better than one which modifies all three $\{n^1, n^3, n^2\}$ so it is correct to finish the first two tasks before enqueueing the third. Thus, (f) will result as an aunt task while producing (d) which was itself a child task from (c), which was a child task from (a).

Finally, the sixth case can be handled by enqueuing a child task for $n^5$ while executing the thread in subfigure (a), then creating a cousin task for $n^4$ at the time that we being executing the task for $n^5$. Note that the existence of cousin tasks means that we essentially must queue up a task corresponding to all $O(n)$ other SPPF nodes every time we being executing a new task.

Now that I have explained the ways to describe all the ways for trees in a given tree expansion set to differ from one another, we are ready to give an algorithm which captures the intuitions of the foregoing discussion.

**Algorithm**

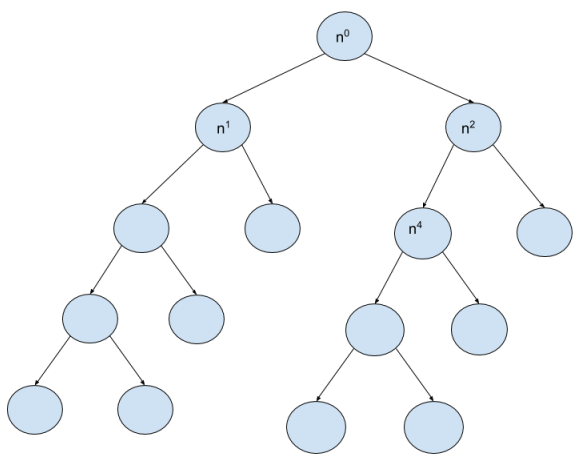
The essence of the algorithm is extremely simple. We will use theorem 2 to show that we can build a queue of pending tasks incrementally. In particular, because of monotonicity and 2 it will always be the case that the $i^{th}$ best tree has an alternate family set that differs from the largest previously seen alternate family set of trees $t_1, ..., t_{i-1}$ by at most one family, and thus we can ensure it will be on the queue at time $i$ by queueing all appropriate tasks for tasks $t_1, ...$ when they are processed.

In particular, we will maintain a queue of tasks and a set of completed trees. When we begin executing a new task with alternate family set $\mathbb{F}_s$, we will check every non-direct-ancestor node (including itself) and create a new task corresponding to instead choosing the suboptimal families of those nodes. For each such suboptimal family, we will add that family to the alternate family set of that task. Note that if a node $n^j$ was already using suboptimal family $f^j_{i_j}$ on this thread of execution, we will add a new task only for suboptimal families $f^j_{i'_j}$ where $i'_j > i_j$. We do this because the families are ordered, thus if we were already on a task where we are using $f^j_{i_j}$ then choosing families with index less than $i_j$ will result in tasks which have already been completed (because their trees will be better than the tree we are currently making).

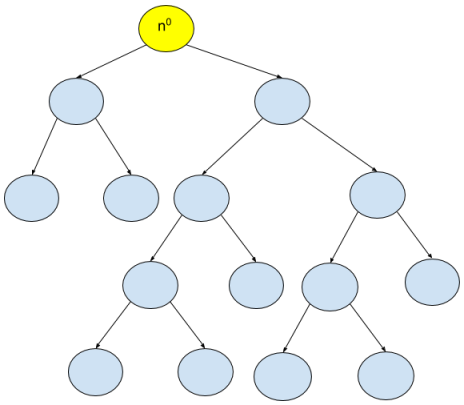
We will filter to ensure no tasks with duplicate alternate family sets will be added to the task queue. Per 1, this will ensure that no tree gets produced twice.

The reason why we only add tasks for non-direct-ancestors is because adding alternate tasks for direct-ancestors may result in the node about which the current thread is anchored ceasing to exist. Additionally, due to monotonicity you can accomplish the same thing by first having a task for the ancestor and then adding one for the descendant, because the task wherein only the ancestor is suboptimal will always be better than one where both the ancestor and descendant are suboptimal.

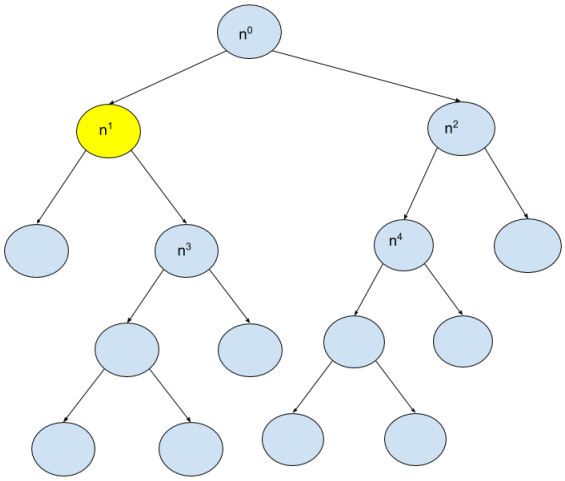
The pseudocode below has the method 'IterativeLoop' responsible for executing tasks in order, a method to execute a task and produce the tree from it, as well as queue up all tasks for all relative nodes. Major tree operations have been abstracted over such as searching for all other non-direct-ancestor nodes in the tree and adding a task for all its suboptimal families. The book-keeping necessary to do this will be contingent on your implementation of the tree data structure. I assume the existence of a map from all non-ancestor nodes to the subtree containing them, as well as the family used to generate them.



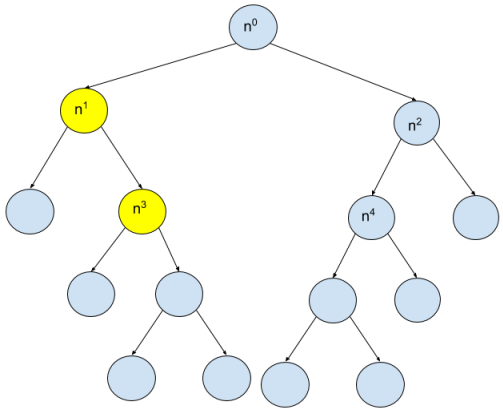

(a)




(b)

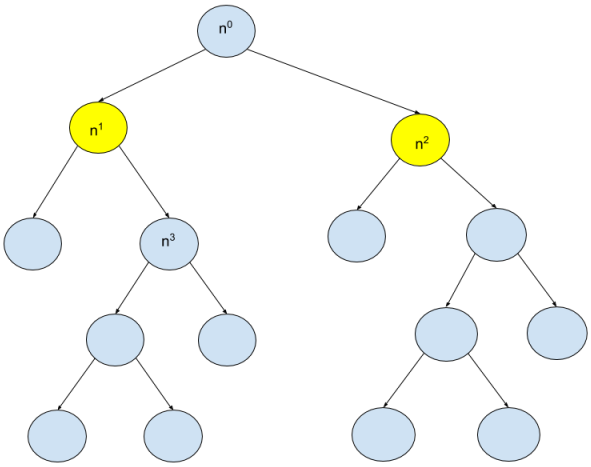



(c)

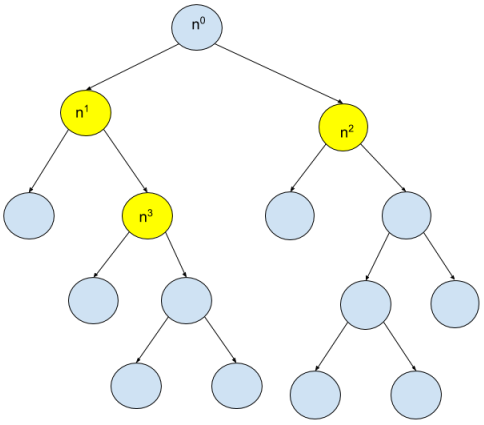



(d)

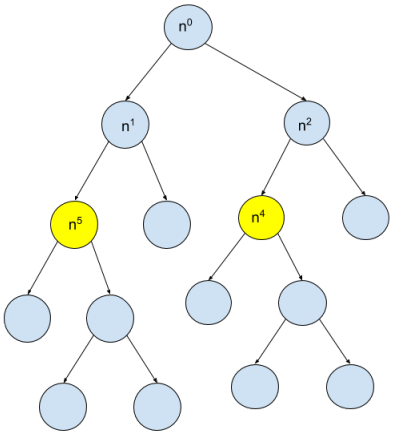
Figure 5.2: The six different ways to derive suboptimal trees along a thread of execution.




(e)




(f)




(g)

Figure 5.3: Cont'd: the six different ways to derive suboptimal trees along a thread of execution.

**Algorithm 17** CPS k-best extraction

```
1: procedure ITERATIVELOOP(root_node, best_attributes)
2:     completed_trees ← {}                                          ▷ Ordered set
3:     origin_tree ← Tree(root_node)
4:     best_family ← root_node.families.peek()
5:     root_task ← (best_family, best_family, best_attributes, root_node, origin_tree, {})
6:     task_queue ← []
7:     task_queue.append(root_task)
8:     while |task_queue| > 0 ∧ |completed_trees| ≤ k do
9:         next_task ← task_queue.pop()
10:        next_tree ← Submit(next_task)
11:        completed_trees.add(next_tree)
12:    return completed_trees
13: procedure SUBMIT(task)
14:    parent_tree ← copy of origin_tree (where the result will be grafted)
15:    tree_root pointer to parent_tree's root
16:    best_subtrees ← {}
17:    for child ∈ task.f′ do
18:        child_tree = Tree(child)
19:        if child.isLeaf() then
20:            child_tree.attributes = child.attributes
21:            parent_tree.children.add(child_tree)
22:        else
23:            best_family ← child.families.peek()
24:            child_tree.attributes ← best_child_family.attributes
25:            task.Recurse(best_child_family, root_attributes, child_tree)
26:            best_subtrees[best_child_family] ← child_tree
27: procedure RECURSE(family, best_attributes, parent_tree)
28:    for child ∈ family do                                         ▷ Expand family
29:        child_tree = Tree(child)
30:        if child.isLeaf() then
31:            child_tree.attributes ← child.attributes
32:            parent_tree.children.add(child_tree)
33:        else
34:            best_child_family = child.families.peek()
35:            child_tree.attributes = best_family.attributes
36:            this.Recurse(best_child_family, best_attributes, child_tree)
37:            best_subtrees[best_child_family] ← child_tree
```

**Algorithm 18** CPS k-best extraction; add subtasks

**procedure** MAKEDESCENDANTTASKS(best_subtrees, family, best_attributes, parent_tree)
  **for** $child \in family$ **do** ▷ Create descendant tasks
    **if** $!child.isLeaf()$ **then**
      $child\_tree = Tree(child)$
      **for** $alt\_family \in child.families[1:]$ **do**
        $best\_family = child.families.peek()$
        $alt\_fam\_attributes = AlternativePathValue(best\_family, alt\_family, best\_attributes)$
        $original\_subtree = best\_subtrees[best\_family]$
        $task\_parent\_tree = clone(original\_subtree).remove\_children()$
        $descendant\_task \leftarrow (alt\_family, best\_family, alt\_fam\_attributes, child, parent\_tree)$
        $task.descendant\_task\_queue.add(descendant\_task)$
**procedure** TWINTASKS
  **for** $twin\_family \in origin\_node.families[1:]$ **do**
    $best\_family = origin\_node.families.peek()$
    $twin\_fam\_attributes = AlternativePathValue(best\_family, twin\_family, root\_attributes)$
    $task\_parent\_tree = clone(origin\_subtree).remove\_children()$
    $alt\_family\_set \leftarrow task.alt\_family\_set \cup twin\_family$
    $twin\_task \leftarrow (twin\_family, best\_family, twin\_fam\_attributes, origin\_node, task\_parent\_tree, alt\_family\_set)$
    $task\_queue.add(twin\_task)$ ▷ Only if $alt\_family\_set$ hasn't been seen before.
**procedure** OTHERTASKS
  **for** $n^j \in non\_ancestor\_nodes$ **do**
    $subtree \leftarrow non\_ancestor\_nodes[node]$
    $f^j_{i_j} \leftarrow non\_ancestor\_nodes[node]$
    $i_j \leftarrow node.families.index\_of(f^j_{i_j})$
    **for** $alt\_family \in node.families[i_j + 1:]$ **do**
      $best\_family = node.families.peek()$
      $alt\_fam\_attributes = AlternativePathValue(best\_family, alt\_family, root\_attributes)$
      $task\_parent\_tree = clone(subtree).remove\_children()$
      $alt\_family\_set \leftarrow (task.alt\_family\_set - f^j_{i_j}) \cup twin\_family$
      $alt\_task \leftarrow (alt\_family, best\_family, alt\_fam\_attributes, n^j, task\_parent\_tree, alt\_family\_set)$
      $task\_queue.add(alt\_task)$ ▷ Only if $alt\_family\_set$ hasn't been seen before.

**Correctness**

It suffices to show the that the $i^{th}$ call to submit will always return the $i^{th}$ best tree (if there are $i$ or more trees available), because then we will be ensured that after $k$ steps, we will have the $k$ best trees in order.

*Proof.* Trivially, we know that the first task will produce the best tree, $t_0$.

By corollary 1 we know that the next best tree will differ in only one family and thus will have alternate family set $\{f^j_{i_j}\}$. Clearly this will be enqueued while producing $t_0$.

Due to lemma 1 we know that we will never produce the same tree twice and therefore at time $i$ the queue will never contain a task corresponding to a tree produced at time $t_j$, $j < i$.

Additionally, according to corollary 3, we know that if we have produced trees $t_i$ then the next best tree $t_{i+1}$'s alternative family set $\mathbb{F}_{i+1}$ differs from that of some tree $t_j$, $j < i$, $\mathbb{F}_j$ by at most one family. Either a) it differs by the addition of a family for a new node (and thus will be one larger), or b) it differs by swapping an existing family of a node (and thus will be the same size).

In either case, we are guaranteed that when we processed $t_j$, we created a task corresponding to $\mathbb{F}_{i+1}$ either via an arbitrary relative task (in the first case) or a twin task. Thus we know that the task for $t_{i+1}$ is on the queue at time $i + 1$ and since no redundant tasks for trees $t_1, ..., t_i$ are on the queue at this time, that task is the next item on the queue.

Note that if there is no next tree, the queue must be empty and therefore the loop will terminate early. □

**Complexity**

As we have shown previously due to the uniqueness of the alternate family set representation 1, we will never produce the same tree twice or enqueue redundant tasks. This means that we will call submit exactly $k$ times.

But, during execution of a task we will produce a tree, which takes $O(n)$ overhead, visit $O(n)$ SPPF nodes and enqueue a task for all of its alternate families, of which there may be $O(n)$. Thus each call to submit takes $n^2$ steps and also adds potentially $n^2$ nodes to the queue which has complexity $n^2 \cdot log(k \cdot n)$. Thus we get an overall complexity of $kn^2 \cdot log(kn)$.

# Appendices

# Appendix A

# Parsing very large grammars[1]

[1]This chapter was published in ACM Southeast 2020, coauthored by Kyle Deeds

## A.1 Abstract

All non-statistical context-free parsers are plagued by a worst case parse time, $O(|G| \times n^3)$, linear in the size of the grammar, $|G|$, which in many applications can cause major slowdowns [8]. In this paper, we pursue general purpose pre-processing intended to speed up the run time of context-free parsers by selecting a smaller fragment of the grammar prior to parsing. Our work provides an ad-hoc method to perform sub-grammar selection, increasing the practicality of parsing with very large context-free grammars. We present an algorithm we call 'Terminal-tree filtering' (TTF), a new variant of the 'b-filtering' algorithm presented in [8], which finds the same filtered set of grammar rules as the 'b-filter' for a given input, but which achieves new state-of-the-art run time and grammar size performance within in this problem space. Our work boasts remarkable performance boosts across a range of workloads, including a 10-20 fold speedup on a real world English grammar of size $|G| \approx 115,000,000$. Furthermore, we achieve far greater scale, parsing sentences using grammars $\approx 10$ times larger than those previously studied. The TTF algorithm filters quickly enough to provide feasible parsing times on our example English grammar. On this very same test suite [8] 's algorithms, which are the broadest, most recent contributions to the field, demonstrate unacceptable run times. We also provide a theoretical analysis which elucidates under which conditions worst-case behavior and best-case behavior can be expected, and show that even under worst-case performance, variants of the TTF still show some empirical advantages to the b-filter.

## A.2 Introduction

### A.2.1 Context-free Languages and Applications

Context-free languages are a formal class of language originally described by Noam Chomsky [14] and subsequently studied in great depth by linguists and computer scientists due to their usefulness in natural language processing and compilers. Parsing natural language is central to approaches in machine translation [4], natural language inference [11] , search engines [42], human computer interactions [34], and many other applications [20].

Though in recent years CFL parsers have ceded some of their supremacy to alternative grammar formalisms and syntactic representations such as dependency parsing [31] and combinatory-categorial grammars [15], many of the most prominent syntactic theories in linguistics are still rooted in context-free languages. This is primarily due to their intuitive form, broad generality, and interesting representational and computational properties which permit their application to problems far outside the world of NLP [33] [3] [36].

Among their formal properties, CFLs have been argued to be sufficiently expressive to represent nearly every syntactic phenomenon of interest in natural language [35] with limited exceptions [40].

### A.2.2 Context-free Grammars and Scalability

In addition to their expressiveness and parsing guarantees, CFLs are made actionable by their capacity to be expressed as a context-free grammar (CFG), an intuitive formalism that describes the unbounded productions of a language as a finite rewrite system containing rules of the form:

$$\text{A} \rightarrow \text{C} \mid \text{BC} \mid ....$$

This rule would be read as "A expands to C, or B then C, or...".

An important result is that all CFGs can be reduced to Chomsky-normal form [13]. A grammar is said to be in Chomsky-normal form if all rules either contain two nonterminals, a single terminal or the empty string on the RHS.

The mapping between CFLs and CFGs is many to one. Additionally, in contrast to regular languages (a.k.a regular expressions, or finite state automata), the task of finding the *provably minimal* CFG that expresses a CFL is undecidable [23]. These two properties make the task of reducing grammar size without reducing expressiveness an open problem. Additionally, grammar reduction is of utmost importance in large-scale parsing applications because algorithms quickly become intractably slow and memory intensive as grammar sizes grow [8].

The main reasons for this intractability are that 1.) the run time of parsing algorithms is proportional to the grammar size and 2.) bottom-up dynamic programming algorithms, such as the Earley algorithm waste a huge proportion of their memory pursuing dead ends during parse time whenever the number of

grammar rules not applicable for the parsed input is much greater than the number of applicable rules - a property which, intuitively, should scale with grammar size in many applications.

#### A.2.3 Statistical Parsing

Avoiding the computational blow-up associated with processing the whole grammar for each parse is an urgent matter in the field of context-free grammars.

One of the most widespread ways of skirting this problem is to statistically annotate the grammar (e.g. using probabilistic context-free grammars [25] or neural networks [10]) so as to enable empirical heuristics which pursue a limited subset of potential incomplete parses at a time. A further potential limitation of statistical parsing is that it typically returns a fixed, finite number of results, pruned for maximum likelihood - while this design is crucial for avoiding the blowup associated with large grammars, it makes it so that statistical parsers never return a representation containing *all* possible parses of a sentence, only *some*.

#### A.2.4 Non-statistical Parsing and Limitations

This paper concerns itself with enhancing non-statistical parsing techniques to skirt the computational blow-up associated with large grammars. Even though probabilistic grammars have achieved great successes, they are only useful in domains where large tagged corpora are available. There are many problem domains in which syntactic parsing is desirable but annotated data sets aren't available and the grammars needed to describe the problem are too large to be feasibly processed by the former state of the art filtering algorithms featured in [8]. Furthermore, as noted above, non-statistical parsing is the only true solution when the user desires *all* possible parses.

Experiment 3 presents one such use-case from an industry collaboration with Charles River Analytics, which achieved meaningful run time improvements on a domain-expert designed English grammar for use in the cybersecurity domain.

### A.3 Related work

Within the non-statistical domain, there are two main methods by which to improve parse times and reduce the memory overhead associated with pursuing dead-ends in the grammar: grammar filtering [8] and guided parsing [6]. Guiding is a generalization of filtering, which was first suggested to the community in the pursuit of linear-time parsing [26].

Filtering is a pre-processing method in which, prior to parsing input $s_i$, an algorithm finds a subset $G_{s_i}$ of the original grammar $G$ such that all possible parses of $s_i$ under $G$ are admissible under $G_{s_i}$. This process can either proceed by strategies which preserve the expressiveness of the underlying CFL or by modifying the expressiveness of the CFL to be a sublanguage which generates $s_i$ but potentially narrows its scope to omit some other elements of the original language.

Guiding is a somewhat more sophisticated, parsing-time method, in which, at every step of the parsing algorithm during which nondeterminism could be introduced, instead of exploring all possible rules at once, use some procedure (called an 'oracle' in [6]) to eliminate any rules which certainly cannot be applied.

Filtering is a degenerate form of guiding, where the same nondeterminism reduction criterion is applied at every step: don't expand a rule in $G$ unless its also in $G_{s_i}$. As mentioned in [6] however, guided parsing is algorithm specific, as different parsing algorithms encounter nondeterminism at different steps in their algorithm, and manage nondeterminism in different ways. For this reason, this paper focuses solely on grammar filtering as a more general approach to run-time reduction for context-free parsing algorithms.

There are a number of papers exploring guiding and filtering strategies for parsing algorithms which broadly follow the principles of this paper: use some criteria relating the input $s_i$ to the rules in $G$ in order to exclude some rules during preprocessing or prevent their application during parse time. These "guides" or "oracles' can take the form of any number of complex processes such as preparsing using a different grammar formalism, constructing an extension of the original language using automata, or filtering the grammar using easily precomputed information about the input. Please see [6], [8] for applications to CFG parsing and [7], [1] for similar principles applied to Tree Adjoining Grammars.

The broadest, most recent catalogue of *filtering* strategies for context-free grammars is presented in [8]. [8] offers a number of filtering protocols anchored by three mutually independent algorithms to find subgrammars, as well as a fourth algorithm they call the 'make-a-reduced-grammar' algorithm - a procedure relying on the removal of rules containing unproductive and unreachable symbols, which can be found in

many introductory texts [23] and which merely reduces the size of the CFG, not the expressiveness of the CFL it represents.

The paper's algorithms include **b-filtering**, which relies on eliminating rules containing terminal symbols which are not in $s_i$, **a-filtering** which uses adjacency criteria to eliminate all rules containing adjacent symbols, which generate terminal symbols that are not adjacent in $s_i$, and finally **d-filtering**, described further in [6], which relies on the construction of a pair of finite state automata defining regular languages that are supersets of the original CFL in question.

In this paper we introduce run-time improvements to the b-filtering algorithm, ignoring both a-filtering and d-filtering because they rely on adjacency information within the input string and so would not generalize to noise-skipping parsers, an important variant of parsing algorithms for real world applications; additionally the d-filter incurs overhead to produce its regular language supersets of $G$ that [8] concedes may suffer from a combinatory explosion for sufficiently large grammars (though the exact formal properties of this technique are left unexplored by the authors). Furthermore, b-filtering is generally a precursor step to a and d-filtering, so all subsequent filtering strategies are bottle-necked by its run time. An important limitation of b-filtering to note is that it only is able to filter out rules containing a terminal, also called a lexicalized rule. This is especially a problem for any grammars which are already in Chomsky-normal form (CNF) [13] as they tend to have a minimal number of lexicalized rules. CNF is used a preprocessing step for the CYK algorithm, so it is likely that our filtering method, and the ones presented in [8] would have limited effectiveness for speeding up CYK parsers.

We now describe b-filtering as presented in [8], our b-tree filtering algorithm, and empirical and theoretical results showing its improvement over the original.

### A.4 Definitions

Context-free languages (CFLs) are a formal classification of language within the Chomsky-hierarchy of languages [12]. They are typically defined either as the set of all string sets recognizable by pushdown automata (PDA) (where each automaton defines a language), or they are defined as the set of strings generable by a context free grammar.

A context-free grammar (CFG) is a 4-tuple $G \equiv (V, P, T, S)$, where $V$ is a Vocabulary of string symbols appearing in the grammar, $T$ is a set of terminal strings which can appear as the tokens in strings belonging to the language, $P$ is a list of productions (or rewrite rules - hence the term rewrite system used to describe this style of representation) of the form $A \to \alpha\beta\gamma...$ where the left-hand side (LHS) is a single nonterminal (formally, an element in $V/T$) and the right-hand side (RHS) is a sequence (possibly empty) of elements in $V$, either terminals or nonterminals. Finally, $S$ is a single element in $V$ called the sentential form, or start symbol. Any string accepted by the language must be generable by a series of productions starting at $S$.

A CFG's size $|G|$ is define as the sum of the lengths of the RHSs for all productions in $P$. A symbol $\alpha \in V$ is unreachable in $G$ if there is no series of productions starting at $S$ which contains $\alpha$. A symbol $\beta \in V$ is unproductive if no series of productions $\beta \to ...$ results in a string containing only terminals from $T$. A rule is useful if it contains no unproductive or unreachable symbols.

The process of determining whether a given string $s$ is in a CFL is called recognizing, while the process of finding all possible derivations is called parsing. A derivation is described as a parse tree. The set of all parse trees for a string $s$ is called a parse forest.

Finally, parsers are algorithms which take grammars $G$ and input strings $s$ and return parse forests. This paper focuses on the Earley parser, an algorithm whose run time, as with many other general CFL parsing algorithms, is $O(|G| \times |s|^3)$ [8].

See [38] for more details on parse forests, [23] for more details on CFLs and formal languages more broadly, and [41], [29], [43] for more information on CFL parsers.

### A.5 B-filtering

b-filtering, when applied to a grammar $G$ and input $s$ works by removing all rules that contain terminals in their RHS that aren't featured in the input $s$. This method is correct because any rule $P$ with terminal $\alpha$ can only ever successfully be applied to a string containing $\alpha$. Thus, when parsing $s$ it is safe to remove $P$ if $\alpha \notin s$.

Consider the grammar *G* below applied to the input $a\ b$:

$$S \rightarrow A\ B \tag{A.1}$$
$$S \rightarrow C\ B \tag{A.2}$$
$$A \rightarrow a \tag{A.3}$$
$$B \rightarrow b \tag{A.4}$$
$$C \rightarrow c \tag{A.5}$$

The b-filtering strategy would eliminate the final rule, leaving us with just rules 1-4.

The downside to Bouiller and Sagot's algorithm is that it requires that we check every rule to see whether it contains any terminals not contained in the input. In particular, we will show in section 6 that its best and worst case run time is $\Theta(|P| \times |T|)$, where $|T|$ is the number of terminals in the vocabulary. For very large grammars, this can be an incredibly time consuming and wasteful process - especially if the vast majority of rules contain terminal symbols excluded by the filtering rule. An ideal solution would be one which only touches those productions which we would like to include in our subgrammar - such a solution would approach the lower-bound performance threshold of $O(|P_s| \times |T|)$, where $P_{s_s}$ are the productions in the filtered subgrammar and $|T|$ is the number of terminals in the vocabulary. We call this design principle **counting-production-in** as opposed to the **ruling-productions-out** approach of standard b-filtering.

b-filtering is a particular algorithm that accomplishes what we will refer to as content filtering. The purposes of this paper is not to modify the semantics of content filtering, but instead to provide a faster algorithm to accomplish it. We refer to the method as terminal-tree filtering because it relies on a tree structure to rapidly index into the grammar based on the terminals in the vocabulary.

In lieu of access to the grammars and specific example sentences they use in their work, we will forego direct comparison of the TTF with their benchmarks and instead compare directly to our own implementation of their b-filtering algorithm. This comparison is valid because all of Boullier and Sagot's benchmarks use b-filtering as a preprocessing step and thus their total run time in all experiments is bounded by the b-filtering time. Therefore, since our TTF finds the same subgrammar in a fraction of the run time, it could be used as a superior preprocessing step to the more advanced a and d-filters offered in their work. We will now introduce the terminal-tree filter algorithm.

### A.6 Terminal-tree filtering

#### A.6.1 Building the Tree

Terminal-tree filtering is motivated by the desire to content-filter the grammar rules by ruling-productions-in, as opposed to b-filtering's comparatively wasteful strategy of ruling-productions-out. The key difference, is that we want to be able to reject subsets of the grammar without having to check every single rule within those subsets rules, allowing us to more closely approach the idealized scenario of a run time proportional to the subgrammar size.

In order to do this, we expend an upfront cost amortized across all queries to construct a binary tree, each node of which contains a set of rules which is a subset of the original grammar $G$ (in experiment 3, we also index this set by the length of the RHS). This tree is then used to determine the proper subgrammar for each input.

Each level of the tree corresponds to a terminal in $T$. We assume an indexing $a_i$ over the terminals $a_0, ..., a_{n-1} \in T$. We first present naive TTF which chooses an arbitrary indexing and terminates a node when it has a single rule in it. We then optimize this construction with a more complex partitioning of the rules called **early-termination**, where the rules for each node are partitioned into two sets: those which are terminating at the current level, and those which are continuing into the next layer. A rule terminates at the current level if all symbols in its RHS have already been indexed at this level. If a node has only terminating rules, it is made into a leaf, regardless of how many terminating rules it contains.

The contents of the tree are defined inductively: the root, at level 0, contains all the rules. The children of each node $b_i$ at level $i$ are constructed as follows: if only one rule is contained in $b_i$, make $b_i$ a leaf, otherwise, partition the rules into two sets: put those which contain $a_i$ into the left child and those which do not contain $a_i$ into the right - obviously all nodes at level $n$ are leaves regardless of the number of rules they contain. The **early-termination** criterion, stipulates that the rules in node $b_i$ are partitioned into

terminating rules (those whose deepest aymbol is $a_i$ and non-terminating rules, furthermore $b_i$ is made into a leaf if it contains only a single nonterminating rule or it only contains terminating rules (i.e. if $a_i$ is the deepest symbol contained in its rules).

Again, though it is not essential for the algorithm, in experiment 3, all rules within each node are organized into sets indexed by the length of the RHS.

There are three **structural properties** of note:

1. Each level consists of some $k$ nodes which contain a combination of mutually exclusive rule sets which are exhaustive the set of all rules $P \in G$.
2. There are at most $|P|$ leaf nodes in the tree
3. If $|P| \gg |T|$ then there are $O(|P|)$ total nodes in the tree

#### A.6.2 Filtering

Call $A_s$ the set of indices for all $a_i$ that we are filtering out (that is, the vocab symbols not in our input $s$). The algorithm first calculates $j$, the deepest symbol contained in $s$, then recursively traverses the tree, always traversing only on right children for indices $l$ in $A_s$, until it either reaches a leaf or reaches level $j$. If we have reached a leaf, we check if its rules contain any symbols in $A_s$ and union all those rules which don't into our working set. If we have gone past level $j$, then we are assured that all of the rules in the node pass the filter and can be unioned into the working set. For the early terminating tree, in addition to filtering the rules at leaves like mentioned above, we also take the full set of terminating rules at every node we reach if the symbol for that node $a_i$ is not in $A_s$.

The pseudocode below demonstrates a recursive implementation of the TTF with no early termination, with $\cup$ denoting set union. Where length-filtering is incorporated, this is a set union over the sets whose RHSs are less than or equal to the sentence length, otherwise it's simply a set union of all rules at that node. In the case where early termination is implemented, the algorithm also adds all the rules in the terminating rule set at each node.

Note that for very large grammars, a recursive style implementation of these algorithms will not run in practice. In the git repository for this project, one can find an iterative implementation inspired by continuation-style programming.

The implementation we use can either simultaneously perform length-filtering on the fly or ignore length. This is true for both tree-based filtering and the standard b-filtering we benchmark against. This is because we store the rules in a hash table partitioned by length of the RHS and are able to choose to either only index rules of appropriate length or index rules of any length

The largest rule used in experiment 2 is of length 12, so this technical point is irrelevant for sentences longer than 12 words, so we will not dwell on it too much. Furthermore, TTF query time isn't substantially impacted by whether length-filtering is used.

### A.7 Analysis

#### A.7.1 Terminal-tree Filter is $O(|P| \times |T|)$

We will now provide a construction for a grammar that forces the TTF to consume $O(|P| \times |T|)$ space and achieves a worst-case run time of $O(|P| \times |T|)$ for the TTF. Note that for this grammar, $|G| \approx |T|$. In a separate technical report, we prove that the worst-case run time and memory consumption for the TTF is $O(|P| \times |T|)$, while this section merely proves that these bounds are tight by providing an example grammar which forces such a run time/space consumption[16].

This grammar is constructed on the premise that trees with long, left-branching chains in the tree, create long chains that grow in proportion to the size of the subset of the grammar contained by the leftmost node in the chain, while right-branching chains can create chains of depth proportional to $|T|$ for arbitrarily small subsets of the grammar.

The first claim is true because, if a chain branches left for depth $d$, and the final node contains $k$ rules, each rule must contain at least $d$ RHS symbols (otherwise they wouldn't be on a left branching subtree). Consequently, the size of the subset of the grammar contained in the bottom most is at least $k \times d$. Thus, even though the chain is of size $d$, those nodes were not created vacuously, as they must be offset by some node containing an $O(d)$-sized subgrammar.

On the other hand, right branching chains are vacuous. This is so because it's possible for some small number of rules $k > 2$ to sit at the end of a right branching chain of length $d$ as long as those rules contain

**Algorithm 19** Terminal-tree filtering

```
1: A_s ← symbols not in the input
2: j ← largest index in A_s
3: set ← ∅
4: root.Recurse(j, set, 0, A_s)
5: procedure RECURSE(j, set, level, A_s)
6:     if this.depth > j then
7:         set ← set ∪ this.rules
8:     else if is_leaf(this) ∧ level ∉ A_s then
9:         set ← filter_list(this.rules, A_s)
10:    else if is_leaf(this) ∧ level ∈ A_s then return set
11:    else if level ∉ A_s then
12:        level + +
13:        set ← set ∪ this.left.Recurse(j, set, level, A_s)
14:        set ← set ∪ this.right.Recurse(j, set, level, A_s)
15:    else if level ∈ A_s then
16:        level + +
17:        set ← node.right.Recurse(j, set, level, A_s)
       return set
18: procedure FILTER_LIST(ruleList, A_s)
19:    set ← ∅
20:    for rule ∈ ruleList do
21:        ruledOut ← false
22:        for terminal ∈ rule.rhs do
23:            if terminal ∈ A_s then
24:                ruledOut ← true
25:                break
26:        if ¬ruledOut then
27:            set ← set ∪ rule
       return set
```

**Algorithm 20** Terminal-tree filtering; early terminating

```
1: A_s ← symbols not in the input
2: j ← largest index in A_s
3: set ← ∅
4: root.Recurse(j, set, 0, A_s
5: procedure RECURSE(j, set, level, A_s)
6:     if this.depth > j then
7:         set ← set ∪ this.terminatingRules
8:         set ← set ∪ this.nonterminatingRules
9:     else if is_leaf(this) ∧ level ∉ A_s then
10:        set ← set ∪ this.terminatingRules
11:        set ← filter_list(this.rules, A_s)
12:    else if is_leaf(this) ∧ level ∈ A_s then return set
13:    else if level ∉ A_s then
14:        set ← set ∪ this.terminatingRules
15:        level + +
16:        set ← set ∪ this.left.Recurse(j, set, level, A_s)
17:        set ← set ∪ this.right.Recurse(j, set, level, A_s)
18:    else if level ∈ A_s then
19:        level + +
20:        set ← node.right.Recurse(j, set, level, A_s)
       return set
```

none of the $d$ symbols corresponding the levels at which the subtree branched right. Of course, some rule must contain those $d$ symbols, otherwise they wouldn't be in the vocabulary at all, therefore for every right-branching chain of length $d$, there must be some left-branching chain containing a subset of the grammar of size $O(d)$.

The question then becomes, is it possible to create an arbitrary number of right-branching chains which are compensated for by only a fixed, finite number of left-branching chains?

We can accomplish this by starting with a grammar $G'$ whose tree is a complete binary tree of depth $|T'|$. We then augment the original $G$ to create one long, left-branching chain of depth $k$ off the left-most leaf, and then right-branching chains of length $k$ off of every other leaf containing more than one rule.

The initial grammar $G'$ would consist of $|T'|$ symbols, one rule for each size-one subset in the powerset of $|T'|$ and two rules for each subset of size greater than one. For all of the $2^{|T'|} - |T'|$ subsets with more than one element, we produce one rule with them appearing in increasing index order in the RHS and a duplicate rule with them appearing in decreasing order. The resulting grammar has $2 \cdot (2^{|T'|} - |T'| - 1) + |T'|$ productions, and is a full binary tree of depth $|T'|$, containing $2^{|T|} - 1$ total leaf nodes, and $2^{|T'|} - |T'| - 1$ leaf nodes with more than one rule. An abbreviated example with $|T'| = 3$ is below:

$$S \rightarrow a \mid b \mid c \mid a\ b \mid b\ a \mid ... \mid c\ b\ a$$

The initial tree size is $|Tree'| = O(2^{|T'|}) = O(|P'|)$, while the initial grammar size is $|G'| = |T'| + \sum_{i=2}^{|T'|} i \cdot \binom{|T'|}{i} = O(|T'| \times 2^{|T'|}) = O(|P'| \times |T'|)$.

We can then augment $G'$ to produce a new $G$ that adds one arbitrarily long left-branching chain to the leftmost portion of the tree, forcing each of the $2^{|T'|} - |T'| - 1$ leaf nodes with more than one rule to add a long right-branching chain. We do this by adding $k$ new unique symbols to the two rules containing all $|T'|$ of the original symbols. This rule ensures that there are two strings in the language for which filtering requires traversing every node in the tree. We don't add any of the new $k$ symbols to any other rule. The resulting grammar has $|T| = k + |T'| = O(k)$, $|P| = |P'| = O(2^{|T'|}$, $|G| = |G'| + k = O(k)$. On the other hand, the resulting tree now has $|Tree| = |Tree'| + k \cdot (2^{|T'|} - |T'| - 1) = O(k \times |P'|) = O(|T| \times |P|)$ - note that $|P| = |P'|$. The new grammar is abbreviated below:

$$S \rightarrow a \mid b \mid c \mid a\ b \mid b\ a \mid ... \mid c\ b\ a\ \alpha_0\ \alpha_1\ \alpha_2$$

We have already shown that this construction contains $O(|P| \times |T|)$ nodes. It is relatively straightforward to see that for all grammars, filtering an input which contains all of the symbols makes the TTF algorithm visit every node, therefore achieving $O(|T| \times |P|)$ run time. This is due to lines 14 and 15 in Algorithm 1. In the next section we show that the b-filter achieves $O(|G|)$ or $O(|P| \times |T|)$ run time on all grammars, even those for which the resulting tree size is closer to $O(|P|)$, like the binary tree described earlier. In the following section, we will argue why the TTF algorithm has better expected performance than the b-filter. Additionally, in experiment 1, we will empirically compare run times on this worst-case grammar and on the best-case binary tree grammar.

Note that we must perform a call to filter_rules every time we reach a leaf, this only adds an $O(c \cdot |P| \times |T|)$ factor, where $c$ is the largest number of rules in a leaf because there are $|P|$ leaves and it takes $|T|$ steps to filter a rule .This is a worst-case in general, though for this particular grammar, there are only two rules which take $|T|$ to filter (the two at the end of the left-branching chain), so we get a constant factor of $O(|T|)$ added due to filtering, as filtering for all the other leaves will take $O(c \cdot |P|)$ negligible time, which is negligible. This doesn't affect the asymptotic worst-case analysis.

For this grammar, adding the **early-termination** criteria can lower the run time to $O(|P| + |T|) = O(|T|) = O(|G|)$, if we are using an ideal indexing, because none of the right-branching subtrees are present, so worst case only the single length $O(|T|)$ left-branch is traversed in addition to the $O(|P|)$ nodes in the full binary section.

To summarize, worst time run times are below:

$$
\begin{aligned}
\text{early TTF} &= |\text{nodes visited}| + |\text{filter rules} \\
&= O(|T|) + O(|T|) \\
&= O(T) \\
&= O(|G|) \\
\text{TTF} &= |\text{nodes visited}| + |\text{filter rules}| \\
&= O(|P| \times |T|) + O(|T|) \\
&= O(|P| \times |T|)
\end{aligned}
$$

median run times should be $O(|P|)$ for both variants, because none of the deep chains should be visited. Though it should be noted, that is only in the case of this particular grammar and this particular indexing, in general, the median run time will be the same as the worst run time.

### A.7.2 B-filtering is $\Theta(|G|)$

Unfortunately, [8] provides no implementation or analysis for b-filtering. The most straightforward approach to b-filtering would involve checking every rule to see if it contains any symbols not in the input. This can be done by constructing the set $A_s$ of symbols not in the input and checking for each $\alpha \in A_s$ and for each $p \in P$ whether $\alpha \in p$. This yields a run time proportional to $\Theta(|P| \times |A_s|)$. This run time is a fixed tight-bound for all input strings and input grammars. Thus, although in the worst-case grammar provided above, the b-filter has a theoretical advantage over the TTF, the $O(|P| \times |T|)$ bound for the TTF is loose, and run time can be far below that for many of the strings in the grammar. In later sections, we will show that even on the worst-case-construction grammar, that the TTF achieves a theoretical speed up for all but the two longest strings in that language.

### A.7.3 Typical TTF Runtime is Faster than the Worst-case $O(|P| \times |T|)$

We will argue first that worst-case run times for the TTF are worse than those for the b-filter. We will, however, also argue that empirical performance is far superior for three reasons:

1. For some worst-case grammars, *typical* sentences in the language show better filter times under TTF than b-filtering.
2. Good indexing can prevent suboptimal tree shapes like the one given in the previous section
3. A tree will have $O(|P|)$ nodes whenever $|P| \gg |T|$. Therefore, *typical* grammars will look more similar to full binary trees than to long chains.

We show that **property 1** is true for this grammar, though it is impossible to say what is typical for ALL grammars given that we cannot assign a distribution to the space of all grammars one might see in practice. We discuss how indexing as described in **property 2** can reduce worst-case or average-case run time. Finally, we show that **property 3** always holds and we will argue that it is more typical for $|P| \gg |T|$ than $|T| \gg |P|$ in practical grammars.

A separate technical report**??** for this shows $O(|P| \times |T|)$ is a worst-case upper bound for the run time of the TTF. In an earlier section we showed that this upper bound is actually a tight bound due to the existence of the worst-case grammar we constructed. Despite the worst case bound being inferior to that for b-filtering, we now prove **property 1** for this grammar by showing that $O(|P| \times |T|)$ run-time occurs for precisely two out of the $|P|$ strings in the language. For the other $|P|$ we visit on average some constant fraction of the $O(|P|)$ nodes in the full binary tree portion of the tree. This is so because of our early stopping criterion. Most strings in the language contain only symbols from the initial vocabulary $T$ and therefore never need to search more than $T$ layers deep. Thus on an input that contains all the symbols in $T$, when we reach layer $T$, we take all of the rules in all of the right children. This step also takes $O(|P|)$ operations because each leaf contains 1 rule.

Thus our median run time for TTF is $O(|P|) \ll O(|G|)$, where $O(|G|)$ is the run time for b-filters. This is what is meant when we say that *typical* run times are faster for our algorithm. Of course, we can make the list of augmented vocabulary symbols arbitrarily large such that the mean run time for TTF is always greater than the mean run time for b-filtering.

**Property 2** is true whenever you can find an index such that long right-branching chains appear at the top of the tree, as this forces there to be a fixed number of such chains. For instance, by indexing all of the $k$ augmented vocabulary symbols before the original elements in $T$ we get one very long left-branching train that terminates in one leaf, and then a long right-branching chain terminating in a full binary tree. Contrast this to our suboptimal indexing from before in which the number of chains was proportional to $|P|$. This would reduce tree size to $O(|T'|)$, giving identical worst-case run time to the b-filter. Of course, in this case, the typical string in the language needs to traverse the entire $k$-length right-branching chain, increasing the median run-time from $O(|P|)$ to $O(|T|)$. Thus we see that **property 2** allows us to strike balance according to whether we value low median run times or low worst run time. This problem is also made more complicated by the fact that for the documents we are interested in parsing, there is a non uniform distribution over the strings in the language that we'd need to take into account when deciding what behavior we prefer.

**Property 3** is true because, for a grammar with a small vocabulary, right-branching chains have essentially constant length. Put differently, each node has at worst a small, constant number of parents with only one child. This means that the tree is approximately a binary tree. The binary tree has $O(|P|)$ leaves and therefore has $O(|P|)$ total nodes.

### A.8 Empirical results

Experiments one and two were run in Java 10.0.2 on a System76 Serval with 64 GB Dual-channel DDR4 at 2400 MHz (4× 16 GB) RAM, and 4.2 GHz i7-6700K (4.2 GHz, 8MB Cache, 4 Cores, 8 Threads) processor. Experiment 3 was run on a Dell Precision M2800 with 16 GB RAM and an Intel Core i7 4th Gen 4710MQ (2.50 GHz) processor.

#### A.8.1 Experiment 1: Empirical Comparison on a Worst-case Grammar

The first experiment tests a worst-case grammar construction consisting of a core full-binary tree grammar augmented by a large number of vocabulary symbols inserted only into a single production. The vocabulary is intentionally indexed to produce a binary tree with long vacuous right-branching chains at each leaf.

We created grammars according to the formula in the worst-grammar construction in the previous section. We initially chose a small vocabulary from which to create a full-binary tree by generating the powerset of all terminals and producing two rules for each powerset, one with the terminals in increasing lexicographic order and one in decreasing. We then augment the grammar with a large number of symbols which get appended to only the two largest of the original strings. 11 grammars were created with parameters vocabularies of size 11 or 12 and with augmented vocabularies of 20, 40, 60, 80, or 100 thousand. The grammars ranged in size between 62,517 and 249,140. Further details are included in the technical report[16].

Displayed in Figure A.1 is the filter time for the worst-case grammar described earlier: stars denote run time on the worst-case string in the language (the string containing all symbols), circles denote run time on the median string in the language. Red denotes the naive TTF, green denotes b-filter, and blue is the TTF with early termination.

Empirical results confirm that both variants of the TTF outperform the b-filter on median elements of the sentence and that the early terminating TTF shows better results than the TTF, with a superior constant factor controlling the asymptotic performance. Furthermore, it appears that the early terminating TTF actually has an edge over the b-filter even on the worst case input. Additionally, the b-filter performs similarly under both the worst-case and the median-case. This experiment only had two different sizes of $|P|$ so no meaningful regression was performed against $|P|$ for our run times.

The run times across median/worst length and across all three filters all correlated most strongly with $|T|$ followed by $|G|$, followed by $|P| \times |T|$. The full table of $R^2$ values is in the technical report[16].

Curiously, the run time curves appear to be best fit by $|G|$ and not by $|P| \times |T|$ (in the case of the worst-length inputs) or $|P|$ (which we should expect for the medians, though we don't have enough different $|P|$'s to run a regression). We would expect $|G|$ to be a better fit than $|P| \times |T|$ for the worst-length times for the early terminating TTF and the b-filter but not for the regular TTF. The latter result can perhaps be explained by the product $|P| \times |T|$ introducing more variance, hurting the empirical correlation. Surprisingly, we find a strong correlation between the median-length run times and $|T|$. It is possible that this results from some

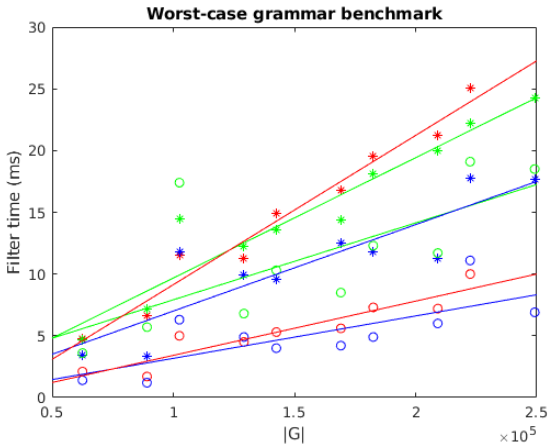



Figure A.1: Worst-case Grammar Run Time

artifact related to memory consumption affecting run time, where memory consumption is worse for large $|T|$.

### A.8.2 Experiment 2: Empirical Comparison on a Best-case Grammar

This experiment is similar to the previous except it uses grammars which result in full binary trees, using the method described earlier. Full binary tree grammars were produced with vocabularies $|T| \in [15, 22]$ resulting in sizes between 491,505 and 92 million. A full spec of the grammars is included in the technical supplement[16].

In this experiment, tree size is proportional to $|P|$. $|G|$ is also roughly proportional to $|P|$ because $|T|$ is negligible. This gives us the result that the b-filter and TTF experience asymptotically equivalent run time proportional to $|P|$.

Displayed in Figure A.2 is the filter time for inputs on a best-case grammar which produces a full binary tree: as in Figure 1, stars denote run time on the worst string in the language, circles denote run time on the median string in the language. Red denotes the naive TTF, green denotes b-filter, and blue is TTF with early termination.

The empirical results in Figure A.2 and Figure A.3, which shows a detail of the median-length TTF run times from Figure A.2, confirm that all three algorithms, irrespective of input length, have run time proportional to $O(|P|)$. For median length inputs, the TTF algorithms are **400x** faster than the b-filter. We also find that on the worst-length input the b-filter is superior to the naive TTF but is actually outperformed by the early terminating TTF.

### A.8.3 Experiment 3: Empirical Comparison on a Grammar of English

The following experiment was conducted on a proprietary implementation of the TTF in use at Charles River Analytics using a grammar which was a fragment of the English language developed to for efforts related to information extraction in the cybersecurity domain. The implementation described here does not include early terminating TTFs and uses length-filtering to rule out rules which have longer RHSs than the input. The correctness of the length-filtering predicate is only guaranteed when the grammars contain no epsilon productions or if the RHSs contain only terminals.

#### English language grammar

The test CFG for this experiment was produced by converting a Systemic Functional Grammar [22] of English based on Edouard Hovy's SFG from the Penman project [24].

The resulting grammar has $|P| \approx 13,000,000$ and $|G| \approx 115$ million. Compare this to the two grammars under study by [8] which had $\approx 500,000$ rules each and total size 1 million and 12 million, respectively.

There are only 40 terminals in the vocabulary, making the resulting terminal-tree in our filter bear a structure much more similar to that for the grammar in experiment 2, than the one from experiment 1, in

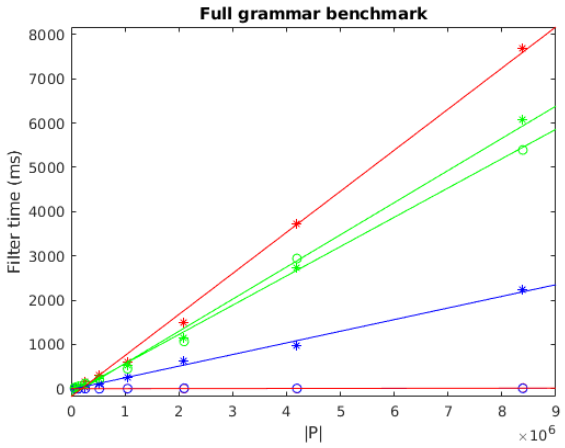



Figure A.2: Best-case Grammar Run Time

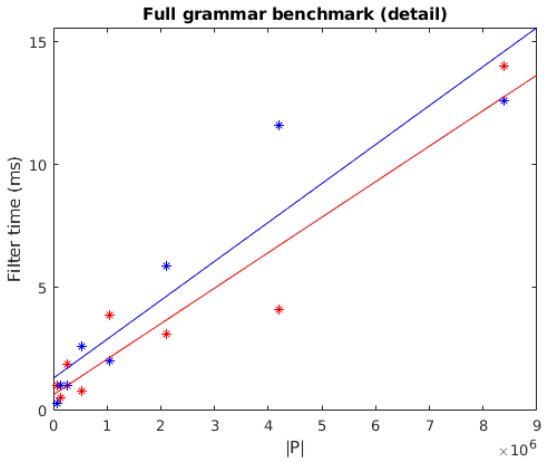



Figure A.3: Detail of Median-length Filter Times

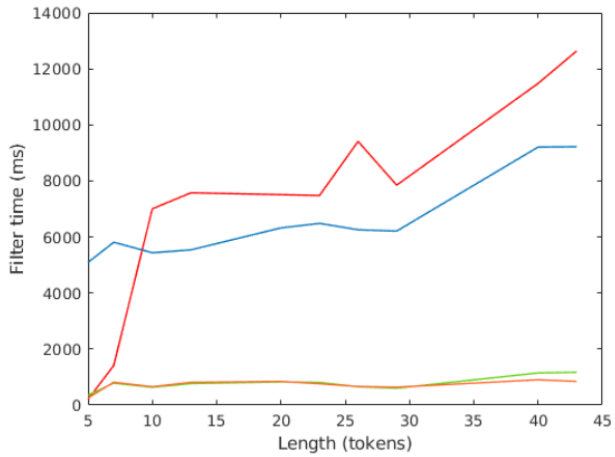



Figure A.4: Results on English Language Grammar

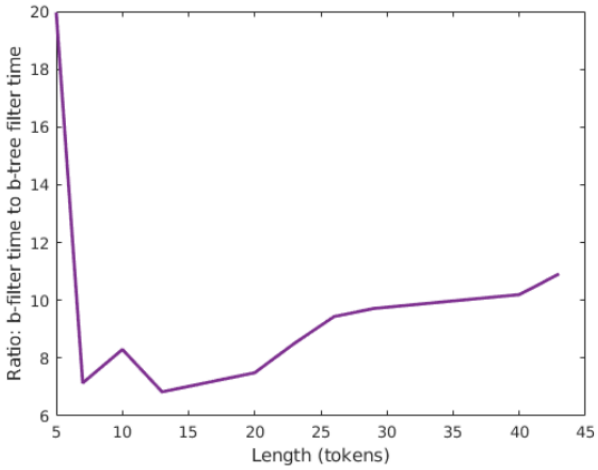



Figure A.5: Ratio of B-filter Run Time to TTF Run Time

particular it should be closer to a full binary tree and should have few long right-branching chains.

The performance is analyzed on a handpicked set of 10 sentences which can be found in the technical report[16].

**Results**

Figure A.4 contains timed benchmarks comparing filtering time for the TTF coupled with length filtering (orange), the TTF without length filtering (green), b-filtering without length filtering (blue), and b-filtering coupled with length filtering (red).Note the approximate 8-fold speedup of TTFs in the large sentence limit.

Figure A.4 demonstrates that b-filtering is competitive only for short sentences and only when combined with length-filtering. Figure A.5 displays the ratio of the b-filter's filter time to the TTF's filter time for sentences of various lengths in the English Language Grammar. As you can see, in all length domains studied here, the TTF is at least 7 times faster than b-filtering thus showing a significant improvement over the previous state-of-the-art in [8]. We note that while the b-filtering algorithm exhibits latencies of as long as 8 seconds, our latency remains under a second. This shows that under the prior state-of-the-art, practical parsing of large corpuses would be out of reach for this grammar.

### A.9 Conclusion and future work

This work has demonstrated a new state-of-the-art filtering technique for massive context free grammar. We have provided a benchmark comparison of a new algorithm, Terminal-tree filtering, which is equivalent to the b-filter used in [8]. The experimental results show that across a range of grammars and workloads designed to mimic the worst-case scenario for our algorithm (experiment 1), the best-case secnario (experiment 2), or a real-life grammar developed at Charles River Analytics for domain-specific NLP tasks (experiment 3), the TTF demonstrates anywhere from slight improvements over the state-of-the-art b-filter presented in [8] on our worst-case grammar to dramatic empirical advantages on grammars exhibiting more plausible properties.

Additionally, we proved a theoretical results explaining the performance of our algorithm. Its performance is optimized on grammars where $|P| \gg |T|$ and in which the queries to the parser tend to be strings randomly drawn from the language, rather than just the longest strings. Both such assumptions are likely to be valid in many use cases. We have also shown that despite our theoretical disadvantages on grammars $|P| \ll |T|$, it is still possible to achieve moderate speed-ups depending on whether the load is primarily strings drawn randomly from the language or solely the longest strings in the language.

Our algorithm tractably filters grammars as large as $|G| \approx 100$ million, several orders of magnitude larger than the largest grammars processed in the most recent comprehensive studies on this subject [8]. Our system dramatically improves the practicality of non-statistical parsing with massive CFGs.

This work expands the upper limits for grammar sizes that can be tractably used in non-statistical parsing applications. Given this increase in the upper limits of available grammar sizes a fruitful area for future work is research on general and robust non-statistical methods for ranking and ordering parses, as well as returning parses in ranked order.

**Acknowledgments**

This material is based upon work supported by DARPA under Contract No. FA8750-17-C-0011. 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 DARPA.

# Appendix B

# Memory-efficient streamed Earley parsing

### B.1 Abstract

This work revisits the noise-skipping Earley parsing presented in chapter three and adapts the core logic to be capable of streaming - that is, processing inputs of unbounded length, one token at a time and returning sentential forms as they appear.

One use case to motivate this approach is deployment in self-monitoring systems, where log entries may be written sporadically over the lifetime of the system and certain sequences of entries can be parsed into events for which some separate diagnostic system must be notified.

Under a continuous stream model it is important to restrict the number of maintained states to the minimum space required to parse all valid sequences.

In this chapter I will first give an overview of the core pieces of metadata from chapter three that need to be altered, discuss the procedural changes necessary to the core logic, and finally, discuss the algorithm necessary to keep track of which states are "active" and which states are ready to be purged from the parser's internal data structures.

In the end, the total memory consumption at any given time index $t$ is proportional to the cube of the longest span active at any given time. That is, if $(D, [@, \delta, B, \mu], [i, j])$ is the active state with smallest start index $i$ then memory consumption is proportional to $O((t-i)^3)$. This is asymptotically equivalent to the memory consumption for an Earley parser with fixed span of size $t - i$, and thus presents a lower bound for memory consumption.

More importantly, the memory consumption, in practice, should not increase dramatically over time, as i, in most cases will likely be proportional to t in most uses.

### B.2 Standard Earley parser memory consumption

A standard Earley implementation knows ahead of time how large its input is (for example, you feed it a sentence at a time, rather than a word at a time) and therefore can initialize its data structures to some fixed size upon start-up.

An important data structure alluded to in chapter three is hash table mapping symbols to all states who have that symbol following their dot, which I will refer to here as $after\_dot$ map. This map is used in the completion phase to find states whose dot can be moved.

For example if $(D, [\delta, @], [j', k])$ has just been completed, we can directly access all the states which are anticipating a $D$ and have end index less than $j'$ , e.g. $(A, [\alpha, @, D], [i, j])$.

This data structure is not strictly necessary for a standard Earley parser. One can simply look at every state in chart $j'$ and find all states whose symbol after the dot is $D$. When noise skipping is introduced, it becomes more worthwhile to introduce the $after\_dot$ map, in order to avoid doing an exhaustive search for charts $j' - w$ through $j'$

We will see that in the streamed case the $after\_dot$ map is a necessity because we only store the charts for $[t - buffer\_size, t]$, where $buffer\_size$ is a parameter, but there could very well exist active states whose end indices are less than $t - buffer\_size$ and whose symbol after the dot is D. Thus in order to avoid maintaining stale charts, we will ensure correctness and compactness by making sure that $after\_dot$ map has all states awaiting completion but none which are no longer capable of being completed.

For example, imagine that our buffer size is 100 and we have $(S, [@, A], [0, 0])$ and $(A, [a_1, a_2, ..., a_{101}, @], [0, 101])$. At this point, it is still valid to complete $(S, [@, A], [0, 0])$ but its chart (chart 0) will have just been deleted during the last 'take', and the current charts in memory will be charts 1 through 101.

Finally, the $completions\_map$ was introduced entirely due to utility functions and thus has no analog in standard Earley parsing, though it is absolutely crucial both that it still serve its primary function in the streamed parser setting and that it be modified to be compact.

#### B.2.1 The Chart

The primary alteration to the chart is that instead of there being an array of charts of length $|input| + 1$, we create a circular buffer (or any bounded data structure) of size $buffer\_size$.

Since we will need to simultaneously use relative indices to access the circular buffer and absolute indices to name the states, we must also maintain a count of the circular buffer's offset to convert absolute indices $(t, t-1, t-2, ..., 1)$ to relative indices.

During each loop (each time step) we also maintain an extra chart called the dormant chart that will be copied over to the "end" (relative) index at the next valid time step - how this is done will be covered in

section 2.2, there is important nuance to this process that arises because we may enter a OOV span for an indeterminate amount of time such that the dormant scan chart produced at $t$ is not copied over until $t + k$.

Finally, whereas a standard Earley implementation may associate one $after\_dot$ map with each chart (because note that the indices of the candidate states for completion matter), the streamed analogue must migrate that map to have global scope and be indexed first by the end (absolute) index of the state then by the symbol after the dot.

It is important to note that in theory, the buffer size could be 2 and the algorithm would still work. It is unclear whether there is any practical or theoretical performance gain from choosing any value larger than 2.

### B.2.2 The $after\_dot$ map

As noted, the $after\_dot$ map has become decoupled from any particular chart and instead must now contain all active states. As such it is also a doubly-keyed hash map, first by index, then by symbol after the dot. As such we must maintain keys for all all indices all index/LHS pairs for which their is at least one active state must also be maintained.

The 'contract' enforced in order to ensure correctness and compactness is that any state which could, at some point in the future be completed (that is - have its dot moved, either via a completion or a scan) must be in the map. I call these states 'active' states.

If this were not the case then some solutions would be lost and 'correctness' would be violated.

In order to ensure compactness, only those states which are "active" should be in the map at the end of each time step in the algorithm.

The difficulty of ensuring correctness and compactness is that determining which states are active can only be done by implicitly maintaining a dependency graph of all the states.

Any state which is actively scanning and hasn't exceeded the skip width is a leaf of the dependency graph - but there can be any arbitrary number of edges radiating out at any depth out from the leaves.

The problem is made more difficult by the fact that which states are leaves can change at each time step (an issue I will address in section 2.2 and section 2.4) making it more practical to only implicitly maintain the graph by tracking dependency counts (similar to ref counts in garbage collection).

### B.2.3 The $completions\_map$

The completions map is vital for calculating the best attribute values for SPPF nodes during the forest building process discussed in chapter four. It does, however, grow proportional to the number of states seen so far, which means that it must be flushed when completions are no longer needed. It is not entirely clear however what policy should be adopted to know when a piece of completion metadata is no longer needed.

It may be the case that it is provably correct to remove a completion metadata entry as soon as one of the entries becomes inactive. If this were the case, it would be difficult to calculate this dynamically as we'd need to make the completions metadata map be configured to be queried by single states at a time.

A much simpler solution emerges however - recall that in chapter three it was noted that mappings from completion metadata state-tuple keys $(nt, q) \mapsto X_p$ to resulting states p are unique. Thus a piece of completion metadata can be uniquely scoped to the new state $p$ produced during the completion.

This means that the global completion metadata map can be replaced by maps scoped solely to the states whose attributes they contain e.g. if a map contains the mappings $(nt, q) \mapsto X_p$, $(nt'q)ßX_p)$, $(nt'', q)ßX_p$, etc. Thus when a state $p$ becomes dead and its memory is freed, the corresponding completions memory is freed as well.

This modification makes it possible to maintain completion information right up until the moment it will never be used again and to implicitly clear it without having to directly find the entries to delete - as one would have to do if the map were global.

## B.3 Core procedural changes

The procedure for streamed noise-skipping parsing differs predominantly for three reasons

1. OOV spans can't be precomputed and in fact, once we enter one we don't know when it is going to be over

2. Typically when we complete a scan for a token at index $i$ we place a completed state for that scan in chart entry $i + 1$, the scan state's end index. This is generally the case for all completions which ensue from that scan - their end index is $i + 1$ and they must be placed on the chart following the current working one. However, in the streamed parsing case, there is not necessarily a next chart (as we will see, there is a chart full of dormant states to be carried over into the next time step) and thus completion must happen as a separate loop from that governing scanning and prediction, as it occurs on a different chart. Thus completions get migrated to a separate phase of the "take" to be after all predictions and scans are completed

3. In the non-streamed parser, having all of our future charts laid out ahead of time simplified scanning and prediction. We could simply place a state corresponding to the prediction/scan on each chart entry $j, j + 1, ..., j + w$, with a distinct state created for each future chart. Furthermore, during completion we could look back into previous charts to see if there were any states whose end index is within $j' - w$. Since we are now working with a bounded buffer of past charts and store no future charts, it is necessary to take all states which are awaiting a scan or a prediction at a later time step to be copied from time step to time step and killed when they have skipped too far. Furthermore, it now becomes necessary book-keeping to maintain a $new\_end\_index$ value for each state which has been copied, indicating its current absolute index. For example at time step $k$ a state $(S, [a, @, a], [j, j + 1])$ would have $new\_end\_index = k$ indicating it has been carried over $k - j - 1$ times. This problem, combined with modification number two, above, calls for us to introduce a dummy chart of dormant states to be carried over into the next time step

#### B.3.1 The "take" loop

The algorithm now proceeds one time step at a time, where each time step corresponds to taking the next token in the stream. Thus each step, which I refer to as a "take" consists of:

1. Incrementing the absolute index of the $last\_token$ read and the index of the $end$ of the buffer, potentially overwriting the buffer,

2. Clearing the dormant states chart to be a fresh chart

3. Modifying the OOV maps, introduced in chapter three, if the current token being taken is an OOV token

4. Copy over dormant states from the last take into the current working chart if still active and purging them if they have skipped too many tokens

5. A loop which performs all scans and predictions for the states in the current working chart (which have been copied over from the dormant states chart in the last part of the algorithm)

6. A loop which performs completions resulting from the scan/prediction loop if there was a successful scan

7. Finally, a clean-up portion which prunes states which are no longer active

I will not elaborate on steps 1-3 as they are implementation specific and entail relatively straightforward book-keeping.

In section 2.2, I will explain how steps 5-6 create dormant states to be copied over in the next 'take'.

In section 2.3, I will explain how step 6 correctly performs completions using the new and improved $after\_dot$ map and the $new\_end\_index$ field calculated during the scanning/prediction step.

Finally, in section 2.4 I will explain how step 4 happens - a discussion I have deferred until the reader has a clearer concept of how $new\_end\_index$ and the dormant states chart works.

Briefly, removing the dead states in part 7 works by iterating over all states marked as dead in the previous 'take', identifying states which were dependent on them and decrementing their $ref\_count$. If any state's $ref\_count$ goes to zero, it is marked as dead and then its dependents are processed, etc.

### B.3.2 Producing dormant states

While reading tokens at time step $t$ during step 5 of the "take" loop given above, states are read in order (given by the topological sort criteria from chapter 2) from the current working chart and states are add to the dormant states chart if one of two conditions hold:

1. If the current token is an OOV token, then for each state $s$ read during step 5, $s$ is mutated so that its $new\_end\_index$ is incremented by one.

2. If for a given state $s$, $new\_end\_index - end\_index \leq w$ (that is, it has been copied over fewer than $w$ times), $s$ is cloned and its $new\_end\_index$ is incremented

Due to the second condition, a normal scan and prediction take place depending on what kind of state $s$ is. In case 1, the loop continues and we begin the next 'take', and if $new\_end\_index - end\_index > w$ then we mark as $s$ as "dead" before continuing.

Scanning and prediction are unmodified from the non-streamed parser, save for few changes. When scanning, if the state doesn't match the current token it is marked as 'dead'. In prediction, for each new state predicted based on $s$, a new state $ns$ is created (as per usual) and $s$ is marked as dependent on $ns$, incrementing the $ref\_count$ of $s$ by one.

The above two cases cover the main ways states die directly. States also implicitly die during completion and during the state clean-up of step 7.

### B.3.3 Completions

Completions are performed during step 6 and are performed only if a successful scan occurred in step 5 - in which case the $end$ of the buffer has been incremented by one (to accomodate the new scanned state), while $last\_token$ is still equal to $t$ (this is why I said earlier that the minimum buffer size is 2 for this algorithm).

At each step, the states in $Charts[end]$ are popped off and either completed if their dot is last, or added to the dormant states chart.

The completion method is otherwise modified only by introduction of some book-keeping to determine which states are now dead or dereferenced (which I will describe in section 3). Additionally, the completion method is modified to access the global $after\_dot$ map instead of what was originally a by-chart map in the non-streaming algorithm.

### B.3.4 Copying dormant states from the last "take"

Copying dormant states into the current chart occurs in step 4 of the "take". First, any state which has their dot set the beginning of their rule - that is, states produced by a prediction in the previous "take" - have their start and end indices (which are equal) reset to be $last\_token$. Then, any state $s$ for which $new\_end\_index \neq last\_token$, must have been carried over 1 or more OOV spans and therefore must have its $new\_end\_index$ reset to $last\_token$. Doing so also requires rehashing it in any hashable data structures it occurred in, such as the $afer\_dot$ map and the hash map storing the dependencies used for reference counting.

## B.4 The active state algorithm

I have already given two examples of how states die: they try to scan a token and they don't match or they are being carried over from the dormant scans chart and have skipped more tokens than the skip width allows.

I have also explained that states marked as dead also dereference their dependents during step 7.

I will now describe how the dependencies are set up, and note the locations in the completion method where references are modified.

Dependencies are added in three places:

1. prediction: when a state $s = (A, [\alpha, @, B, \beta], [i, j])$ predicts all possible rules containing B as an LHS, for each new resulting state $ns = (B, [@, \gamma], [j, j])$, $s$ is made dependent on $ns$ and $ns$'s ref count is incremented by one

2. clone: when a state is cloned in case 2 of section 2.2, all states dependent on the original state are all made dependent on the clone and its ref count is set appropriately

3. completion: when a state $s$'s dot is moved as a result of completion, all dependents $d$ of the original state add, as a dependent, the new state $s'$ resulting from moving the dot forward - the ref counts for $d$ are not incremented here because the new state $s'$ is replacing the old one $s$.

Dependencies are decremented in several places:

1. When a state is marked dead, during step 7 of clean-up, ref count gets decremented for every dependent of the dead state.

2. If performing completion via a state $s = (D, [\delta, @], [j', k])$ - for every state $st = (A, [\alpha, @, D], [i, j]$ completed by $s$ whose dot is in the penultimate index, the ref count of $st$ is decremented. This must happen because in item 1, in the list above, $s$ was added as a dependent of $st$ during prediction and $s$ is now dead.

3. If completion of $st$ is occurring via $s$ and $s$ is a terminal symbol,then we decrement the ref count of all dependents $d$ of $st$. We do this because item 3 in the list above shouldn't apply, so while we are removing the dependency between $d$ and $st$ no new dependency is added in its place

4. If we are advancing the the dot in $st$ by 1 due to completion, then for every dependent $d$ of $st$, if $ns$ (the state resulting from moving $st$'s dot forwards) already exists in that dependent's dependencies, we decrement the ref count because item 3, from the prior list, applies but we are not replacing the old state $st$ with a new one (because the new state $ns$ already has been included in $d$'s ref count).

For any state $st$ have its dot moved by completion of $s$ in case 4, we mark $st$ as dead. We don't allow it to decrement the ref counts of its dependents during step 7, because the decrements have already been accounted for in the cases above. We can ensure this by simply removing its references to its dependents so that the clean-up process doesn't get the chance to decrement theiref counts.

This makes sense, because when we add dependencies during completion, we have created a successor state $ns$ which all dependents $d$ of $st$ are now dependent on, in place of $st$. The ref count is not changed if $ns$ is a new state.This is so because $ns$ as a successor to $st$, $ns$ inherits all of $st$'s dependency relations. You can think of $st$ as having not really 'died' in this sense, but rather having been replaced with $ns$. Thus we need to make sure that $st$'s dependents aren't reaped when cleaning up dead states, because they aren't really $st$'s dependents since $ns$ inherited them. If, however, $ns$ isn't new we decrement $d$'s ref count because we have killed $st$ without adding a new state to $d$'s dependency list to compensate for it.

In case 2, we do allow $st$ to keep its references to its dependents, making it possible to once again decrement their ref counts when killed. This is so because in case 2, there is no successor $ns$ to replace $st$ (because the dot can't move anymore). The entire chain of dependency extending from $d$ to $st$ is now complete, and with no states to inherit its dependencies, it is solely responsible for decrementing the ref counts of its dependents upon its death.

# Bibliography


[1] François Barthélemy et al. "Guided Parsing of Range Concatenation Languages". In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. ACL '01. Toulouse, France: Association for Computational Linguistics, 2001, pp. 42–49. DOI: 10.3115/1073012.1073019. URL: https://doi.org/10.3115/1073012.1073019.

[2] G Edward Barton. "On the complexity of ID/LP parsing". In: *Computational Linguistics* 11.4 (1985), pp. 205–218.

[3] Gabriël J. L. Beckers et al. "Birdsong neurolinguistics: songbird context-free grammar claim is premature." In: *Neuroreport* 23 3 (2012), pp. 139–45.

[4] Luisa Bentivogli et al. "Neural versus Phrase-Based Machine Translation Quality: a Case Study". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 257–267. DOI: 10.18653/v1/D16-1025. URL: https://www.aclweb.org/anthology/D16-1025.

[5] Nathan Bodenstab et al. "Beam-Width Prediction for Efficient Context-Free Parsing". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 440–449. URL: https://www.aclweb.org/anthology/P11-1045.

[6] Pierre Boullier. "Guided Earley Parsing". In: *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France, Apr. 2003, pp. 43–54. URL: https://www.aclweb.org/anthology/W03-3005.

[7] Pierre Boullier. "Supertagging: A Non-Statistical Parsing-Based Approach". In: *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France, Apr. 2003, pp. 55–65. URL: https://www.aclweb.org/anthology/W03-3006.

[8] Pierre Boullier and Benoît Sagot. "Are Very Large Context-Free Grammars Tractable?" In: *Trends in Parsing Technology: Dependency Parsing, Domain Adaptation, and Deep Parsing*. Ed. by Harry Bunt, Paola Merlo, and Joakim Nivre. Dordrecht: Springer Netherlands, 2010, pp. 201–222. ISBN: 978-90-481-9352-3. DOI: 10.1007/978-90-481-9352-3_12. URL: https://doi.org/10.1007/978-90-481-9352-3_12.

[9] Eugene Charniak. "Statistical Techniques for Natural Language Parsing". In: *AI Magazine* 18 (1997), pp. 33–44.

[10] Danqi Chen and Christopher Manning. "A Fast and Accurate Dependency Parser using Neural Networks". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. DOI: 10.3115/v1/D14-1082. URL: https://www.aclweb.org/anthology/D14-1082.

[11] Qian Chen et al. "Enhanced LSTM for Natural Language Inference". In: *ACL*. 2016.

[12] N Chomsky and MP Schützenberger. *The Algebraic Theory of Context-Free Languages*. Vol. 35. Computer Programming and Formal Systems. Amsterdam: Elsevier, 1963, pp. 118–161. ISBN: 9780444534002.

[13] N. Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6. URL: http://www.sciencedirect.com/science/article/pii/S0019995859903626.

[14] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (Sept. 1956), pp. 113–124. ISSN: 2168-2712. DOI: 10.1109/TIT.1956.1056813.

[15] Stephen Clark and James R. Curran. "Log-Linear Models for Wide-Coverage CCG Parsing". In: *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*. EMNLP '03. USA: Association for Computational Linguistics, 2003, pp. 97–104. DOI: 10.3115/1119355.1119368. URL: https://doi.org/10.3115/1119355.1119368.

[16] Jeremy Dohmann. *Technical Report*. https://github.com/a-fast-filtering-algorithm/A-fast-filtering-algorithm-for-massive-context-free-grammars. 2020.

[17] Jeremy Dohmann and Kyle Deeds. "A Fast Filtering Algorithm for Massive Context-free Grammars". In: *Proceedings of the 2020 ACM Southeast Conference, ACM SE '20, Tampa, FL, USA, April 2-4, 2020*. Ed. by J. Morris Chang, Dan Lo, and Eric Gamess. ACM, 2020, pp. 62–70. DOI: 10.1145/3374135.3385266. URL: https://doi.org/10.1145/3374135.3385266.

[18] J Earley. "An efficient context-free parsing algorithm [Dissertation]". In: *Computer Science Department, Carnegie-Mellon University* (1968).

[19] Jay Earley. "An Efficient Context-Free Parsing Algorithm". In: *Commun. ACM* 13.2 (Feb. 1970), pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035. URL: https://doi.org/10.1145/362007.362035.

[20] Carlos Gómez-Rodrıguez, Iago Alonso-Alonso, and David Vilares. "How Important is Syntactic Parsing Accuracy? An Empirical Evaluation on Sentiment Analysis". In: *CoRR* abs/1706.02141 (2017). arXiv: 1706.02141. URL: http://arxiv.org/abs/1706.02141.

[21] Joshua Goodman. "Semiring Parsing". In: *Computational Linguistics* 25.4 (1999), pp. 573–606. URL: https://www.aclweb.org/anthology/J99-4004.

[22] Michael Halliday and Christian M.I.M Matthiessen. *An Introduction to Functional Grammar*. Hodder Education, 2004.

[23] John E. Hopcroft, Jeffrey D. Ullman, and Rajeev Motwani. *Introduction to automata theory, languages and computation*. Pearson education, 2014.

[24] Eduard Hovy. "The Penman Natural Language Project". In: *Proceedings of the Workshop on Speech and Natural Language*. HLT '90. Hidden Valley, Pennsylvania: Association for Computational Linguistics, 1990, p. 430. DOI: 10.3115/116580.1138614. URL: https://doi.org/10.3115/116580.1138614.

[25] Mark Johnson. "PCFG Models of Linguistic Tree Representations". In: *Comput. Linguist.* 24.4 (Dec. 1998), pp. 613–632. ISSN: 0891-2017.

[26] Martin Kay. "Guides and oracles for linear-time parsing". In: *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT)*. 2000, pp. 6–9.

[27] Donald E. Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(65)90426-2. URL: http://www.sciencedirect.com/science/article/pii/S0019995865904262.

[28] Zhifei Li and Jason Eisner. "First- and Second-Order Expectation Semirings with Applications to Minimum-Risk Training on Translation Forests". In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, Aug. 2009, pp. 40–51. URL: https://www.aclweb.org/anthology/D09-1005.

[29] George F. Luger and William A Stubblefield. *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java for Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 6th. USA: Addison-Wesley Publishing Company, 2008. ISBN: 0136070477.

[30] George F. Luger and William A. Stubblefield. *Artificial Intelligence (2nd Ed.): Structures and Strategies for Complex Problem-Solving*. USA: Benjamin-Cummings Publishing Co., Inc., 1993. ISBN: 0805347801.

[31] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. "Generating Typed Dependency Parses from Phrase Structure Parses". In: *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. Genoa, Italy: European Language Resources Association (ELRA), May 2006. URL: http://www.lrec-conf.org/proceedings/lrec2006/pdf/440_pdf.pdf.

[32] Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. "Probabilistic CFG with Latent Annotations". In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 75–82. DOI: 10.3115/1219840.1219850. URL: https://www.aclweb.org/anthology/P05-1010.

[33] P. M. Maurer. "Generating test data with enhanced context-free grammars". In: *IEEE Software* 7.4 (July 1990), pp. 50–55. ISSN: 1937-4194. DOI: 10.1109/52.56422.

[34] Amit Mishra and Sanjay Kumar Jain. "A survey on question answering systems with classification". In: *Journal of King Saud University - Computer and Information Sciences* 28.3 (2016), pp. 345–361. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2014.10.007. URL: http://www.sciencedirect.com/science/article/pii/S1319157815000890.

[35] Geoffrey K. Pullum and Gerald Gazdar. "Natural languages and context-free languages". In: *Linguistics and Philosophy* 4.4 (Dec. 1982), pp. 471–504. ISSN: 1573-0549. DOI: 10.1007/BF00360802. URL: https://doi.org/10.1007/BF00360802.

[36] Yasubumi Sakakibara et al. "Stochastic Context-Free Grammars for tRNA Modeling". In: *Nucleic Acids Research* 22 (Dec. 1994). DOI: 10.1093/nar/22.23.5112.

[37] Bram van der Sanden et al. "Parse Forest Disambiguation". PhD thesis.

[38] Elizabeth Scott. "SPPF-Style Parsing From Earley Recognisers". In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008). Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007), pp. 53–67. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.03.044. URL: http://www.sciencedirect.com/science/article/pii/S1571066108001497.

[39] Elizabeth Scott, Adrian Johnstone, and GR Economopoulos. *BRN-table based GLR parsers*. Tech. rep. Technical Report TR-03-06, Royal Holloway, University of London, Computer . . ., 2003.

[40] Stuart M. Shieber. "Evidence Against the Context-Freeness of Natural Language". In: *The Formal Complexity of Natural Language*. Ed. by Walter J. Savitch et al. Dordrecht: Springer Netherlands, 1987, pp. 320–334. ISBN: 978-94-009-3401-6. DOI: 10.1007/978-94-009-3401-6_12. URL: https://doi.org/10.1007/978-94-009-3401-6_12.

[41] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. "Principles and Implementation of Deductive Parsing". In: *CoRR* abs/cmp-lg/9404008 (1994). arXiv: cmp-lg/9404008. URL: http://arxiv.org/abs/cmp-lg/9404008.

[42] Tomek Strzalkowski et al. "Building Effective Queries In Natural Language Information Retrieval". In: *Fifth Conference on Applied Natural Language Processing*. Washington, DC, USA: Association for Computational Linguistics, Mar. 1997, pp. 299–306. DOI: 10.3115/974557.974601. URL: https://www.aclweb.org/anthology/A97-1044.

[43] Masaru Tomita. "An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications". AAI8517539. PhD thesis. USA, 1985.

[44] Daniel H. Younger. "Recognition and parsing of context-free languages in time n3". In: *Information and Control* 10.2 (1967), pp. 189–208. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(67)80007-X. URL: http://www.sciencedirect.com/science/article/pii/S001999586780007X.