Abstract

TOWARDS EFFICIENT DATA NORMALIZATION USING GRAMMAR-BASED PARSING

Somnath Basu Roy Chowdhury, STCI Knowledge & Experience, somnath.basu@microsoft.com
Aresh Mishra, STCI Knowledge & Experience, aresh.mishra@microsoft.com
Ayan Banerjee, Bing, ayanba@microsoft.com

Keywords. Context-free grammar, earley parsing, natural language processing.

This paper presents the formulation of an efficient end-to-end grammar authoring and parsing framework for natural language-based data normalization tasks. We propose a generalized version of an Earley Recognizer combining the LR(0) -DFA formulation from [1] and Leo-reduction from [2]. Our Earley recognizer setup supports annotations and ambiguous tokenization that increases its scalability to many real-world problems. The paper delves into various practical use-cases where grammar-based systems play a vital role in data normalization. We also analyze the performance issues encountered while authoring such grammars and present two solutions to tackle them. Finally, we present the various features of the framework which provides a superior grammar authoring experience for developers.

1. Introduction

Context-free grammars (CFGs) provide a powerful framework for parsing programming languages and parser programs in compilers. A CFG is a set of recursive rules that help define a parsable context-free language. Despite the vast literature behind CFGs, it is not widely used for natural language-based tasks due to its ambiguous nature. Most production systems today use rule-based techniques for data cleansing/normalization. In this paper, we explore data curation based application arenas where CFG-based parsing is a better choice than conventional machine learning techniques as it is robust to input variability and has no data constraints.

Earley parser [3] is the most widely accepted general purpose CFG parser (others include CYK [4] and LALR [5]) parser, which provides a general asymptotic time complexity of $O(n^3)$. It works in $O(n^2)$ for unambiguous grammar and $O(n)$ for LR-regular grammar with [2]. In this paper, we present an end-to-end parsing system (more specifically a software tool), which provides a platform to write CFGs and execute them on a corpus on natural language. At the core of this system is a generalized implementation of the Earley parser which is discussed in detail in Section 4.

The motivation behind this work is application-driven, the original problem we aimed to solve was extracting relevant information from unstructured noisy text phrases and curate them into knowledge/concept graph triples. Table 1 depicts some representative examples of product weight extraction samples to be normalized. Data normalization at a web-scale is difficult using convention string-based pattern matching due to a large number of possible patterns. Grammar-based parsing is the most efficient solution to tackle this problem.

We present an end-to-end grammar framework for grammar authoring and parsing. The four major *contributions* of the paper are the following

1. We propose ambiguous tokenization and text fragment annotation, which can be leveraged while authoring ambiguous context-free grammars.
2. We present a generic Earley recognizer implementation that combines the effectiveness of [1] and [2] while supporting features like ambiguous tokenization.
3. We investigate some practical application arenas where grammar parsing is useful and present two novel approaches to leverage the Earley parsing algorithm to allow pattern search over lengthy text phrases.
4. We explore additional features like entity-linking, which makes parsers scalable to tackle a broad range of real-world normalization tasks.

To the best of our knowledge, we present the first end-to-end system to leverage grammar-based parsing for natural language-based data normalization.

Raw Text	CG/Weight/Value	CG/Weight/Unit
1 kg	1	kg
wt:38.8 lbs	38.8	lbs
=114.64G	114.64	g
15.4lbs /7kg	15.4, 7	lbs, kg

Table 1: An example of a data normalization task.

2. Background

2.1. Context-free grammar

Context-free grammar (CFG) is defined as a set of rules that describe all possible sentence patterns for a given formal language. In a formal language, *terminals* are a set of words defined in the grammar which are actually present in the language, and *non-terminals* are variables in the grammar that can be expanded into non-terminals and terminals. In Chomsky Normal Form, a context-free grammar G is defined by the 4-tuple $G = (V, \Sigma, R, r)$ where

- V is the set of non-terminal variables.
- Σ is the set of terminal symbols.
- R is the set of production rules or finite relation set from V to $(V \cup \Sigma)$.
- r is the root of the grammar, also known as the start symbol.

An example of a practical context-free grammar is shown in Table 2 for the data normalization task presented in Table 1.

(1)	[Weight] = [Value][Unit]
(2)	[Value] = [Decimal]
(3)	[Unit] = kg
(4)	[Unit] = lbs
(5)	[Unit] = G

Table 2: Context-free grammar example.

The CFG is used to parse symbols in a formal language adhering to its production rules and generate a parse tree.

The root of the parse tree is the start symbol r and the leaf nodes denote the terminals. In the next section, we will discuss the basic concepts and workflow of an Earley parser.

2.2. Earley algorithm

In this section, we analyze the Earley algorithm **[3]** which uses a top-down dynamic programming approach to store parses of the input text phrases in a compact manner. The algorithm leverages a data structure that is an array of *Earley sets*, one set $X(j)$ for each symbol in the text phrase. Every set contains *Earley items*, that represent partial parse of a production rule. An earley item encapsulates the following information:

- A *production rule* in G of the form $S \rightarrow \alpha\beta$.
- *Dot Position*: The current position in the rule till which parsing is complete.
- *Position of origin*: The position at which parsing of the rule began.
- *Current Position*: in the input text phrase.

An earley item is represented as 3-tuple $(S \rightarrow \alpha \bullet \beta, i, k)$ where $\bullet$ denotes the dot position in the production, i is the origin position and k is the current position in the text phrase. This item represents a partial parse of the non-terminal S where α has already been parsed using symbols from position i to k and β is yet to be parsed.

The algorithm proceeds by processing symbols from left to right manipulating the corresponding earley set $X(j)$. $X(0)$ is initialized with the root rule of the grammar and the algorithm repeatedly executes three operators: predictor, scanner and completer.

1. Predictor: For every incomplete earley item of the form $(S \rightarrow \alpha \bullet A, i, k)$ in set $X(j)$ add all earley items of the form $(A \rightarrow \bullet\gamma, k)$ where A appears on the LHS.
2. Scanner: For every earley set $X(j)$ having an earley item $(S \rightarrow \alpha \bullet aA, i, k)$, if the next symbol is a add $(S \rightarrow \alpha a \bullet A, i, k+1)$ to $X(j+1)$.
3. Completer: For every earley item $(A \rightarrow \beta\bullet, m, j)$ in the set $X(j)$, find items of the form $(S \rightarrow \alpha\bullet A\beta, i, m)$ in $X(m)$ and add $(S \rightarrow \alpha A \bullet \beta, i, j)$ to $X(m)$.

The three operators are executed iteratively till no other earley item can be added to a set. The parsing is successful when the root production rule is complete in the final earley set $X(N)$, where N is the length of the text phrase.

3. Methodology and implementation

In this section, we discuss in detail our implementation of the end-to-end parsing framework. At the core of the parsing framework lies a generalized implementation of the Earley recognizer. We also discuss additional features of the framework which makes it scalable for many real-world applications. The overview of the framework is shown in Figure 1.

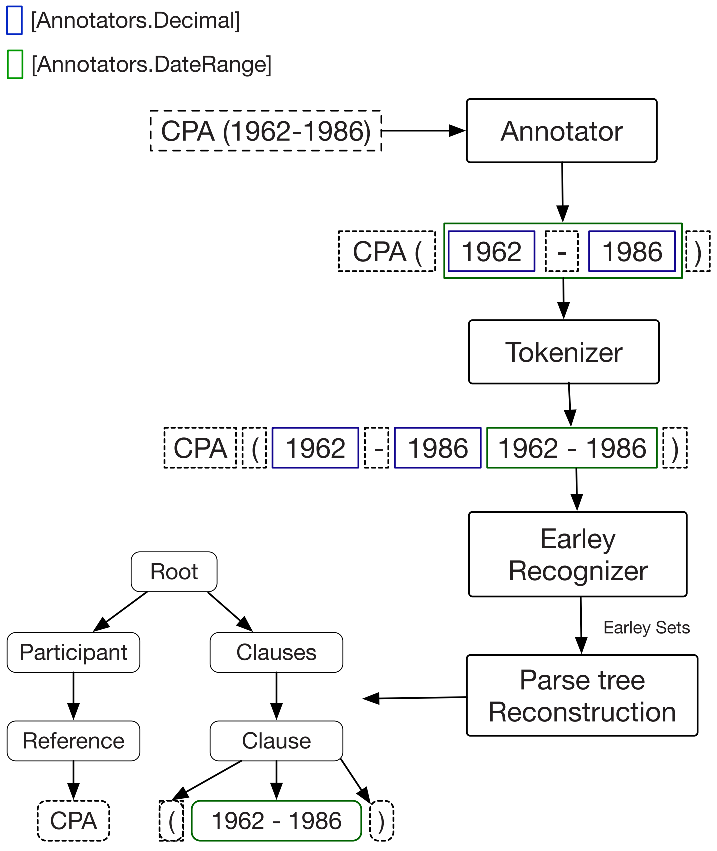

Figure 1: End-to-end grammar work flow.

3.1. Annotations

We introduce the feature of annotating string fragments in an input text phrase to be parsed. Annotations encapsulate parsing of generic string fragments like date, number range, alphanumeric strings, etc. These annotations can then be directly accessed as non-terminal during grammar authoring. An example of annotation operation being applied on a text phrase is shown in Table 3.

Annotation requirements may vary depending on the task at hand, therefore it is available as a *pluggable* feature in the system. Our system even supports multiple/hierarchical annotations. Custom annotators can be leveraged for powerful applications. For example, if we have a grammar which involves names of prominent political persons, the list of recognized political people (names/ wiki URL) can be uploaded as an annotator config, instead of matching any string as a proper noun. We can then leverage [Annotators. PoliticalPerson] non-terminal in the CFG.

Fragment	Annotation
1962	[Annotators.Decimal]
1986	[Annotators.Decimal]
1962-1986	[Annotators.DateRange]

Table 3: Example of Annotation.
Selected Input Sample: CPA (1962 - 1986)

3.2. Ambiguous tokenization

For the parser to operate on a text phrase, the phrase has to be tokenized into fragments. Tokenization in natural language is non-trivial and ambiguous at times. An example of ambiguous tokenization is shown in Table 4.

The fragments are grouped by their end index and processed sequentially during the Earley algorithm operations. Tokenization is also a pluggable component. Tokens formed are stored in a generic fragment data structure which can be instantiated as a string fragment, alphanumeric fragment, date fragment, and others by leveraging the annotations. Custom tokenizer implementations can help parse text phrases from multiple languages as well.

Fragment/Annotations	Start	Length
CPA	0	1
(	1	1
1962	2	1
[Annotators.Decimal]	2	1
-	3	1
1986	4	1
[Annotators.Decimal]	4	1
1962-1986	2	3
[Annotators.DateRange]	2	3
)	5	1

Table 4: Example of ambiguous tokenization.

Our system supports the following tokenizer implementations:

- *Naive*: tokenizes the string based on common separators.
- *NLP-based*: splits the text phrase using Viterbi decoding [6].
- *Aggressive*: aggressively splits the string based on alphabet transition edges (letter→digit, digit→letter, letter→separator & digit→separator).
- *Combined*: returns the combined tokens from aggressive and NLP-based tokenizer after deduplication.
- *Japanese Word breaker*: tokenizer for Japanese language text phrases.

  Only a single instance of a tokenizer can be used at a time. Tokenizer generates a set of generic fragments ($\mathcal{T} = \{f_1, \ldots, f_n\}$) that are processed by the parser next.

4. Generalized earley algorithm

In this section, we describe in detail our implementation of the parser. The computation of earley sets can be improved if we can precompute earley item transitions to form an LR(0) deterministic finite automata (DFA) as shown in [1]. It appears that groups of earley items always occur together in an earley set, which is defined as a ϵ-DFA state. Aycock and Horspool [1] proposed splitting a ϵ-DFA state into kernel and non-kernel states based on the dot-position of the earley items.

- **Earley States**: In our implementation, we introduce *Earley State* (E_s) which is defined as

$$\mathrm{E_s} = [\mathcal{R}, \mathcal{F}'] \quad (1)$$

where $\mathcal{R}$ is a set of dotted rules which co-occur in earley sets and $\mathcal{F}'$ is a set of generic fragments, s.t. $\mathcal{F}' \subseteq \mathcal{F}$ ($\mathcal{F}$ is the set of all terminals defined in the grammar). Transition between earley states is shown below where $\mathrm{E_s}$ and $\mathrm{E_s}'$ are valid non-empty earley states.

$$\mathrm{E_s}' \times f = \mathrm{E_s}, \quad \forall f \in \mathcal{F}' \quad (2)$$

In the actual implementation of the *earley state* data structure some indicators of kernel states and end state are cached for faster runtime processing.

- **Earley State Item**: We leverage the fact that groups of dotted rules co-occur and postulate that earley items defined in Section 2.2 can have multiple productions instead of one. This helps us reduce the size of earley sets and yields faster processing. We define a new entity *Earley State Item* (E_s) which is linked to an earley state.

$$I(\mathrm{E_s}) = [\mathrm{E_s}, i, f, \mathrm{E_s}', c] \quad (3)$$

where, E_s is the earley state to which this item is linked, i is the *origin* position, f is the *last fragment* encountered which produced this earley state item (shown in Eq. 2), E_s' is the *parent* earley state and c is the *confidence* associated with the state item. f, E_s' and c are optional and only used during the parse tree reconstruction process. Our system provides users the option to assign confidence values to production rules in case of ambiguous grammars. If not assigned, confidence values are initialized uniformly resulting in shallower parse trees being preferred over deeper ones.

- **Earley State Transition**: An earley state transition between two states induced by a fragment is defined as

$$\mathcal{T}_e = [\mathrm{E_s}^{(s)}, \mathrm{E_s}^{(t)}, f] \tag{4}$$

where, $\mathrm{E_s}^{(s)}$ is the source earley state, $\mathrm{E_s}^{(t)}$ is the target earley state and f is the transition fragment. Earley state transitions are pre-computed during grammar compilation and leveraged during ϵ-DFA construction.

- **Earley State Set**: This set encapsulates information about earley state items instead of individual earley items. We define an earley state set as

$$\mathcal{S}(j) = [\mathcal{E}, \mathcal{T}_s] \tag{5}$$

where, $\mathcal{E} = \{\mathrm{E}_s^{(1)}, \ldots, \mathrm{E}_s^{(n)}\}$ is the set of earley state items, j is the fragment index and $\mathcal{T}_s$ is the cached information about transitive steps [2] for the earley state set. $\mathcal{T}_s$ is updated whenever an earley state item is added and this memoization speeds up Leo-reduction process in case of right-recursive grammar.

Now that we have introduced the terminology, the complete algorithm for the generic earley recognizer is presented in Algorithm 1.

The input grammar G is compiled first to form the ϵ-DFA states and identify transition edges. The transitions are cached in the form of *earley state transitions* $\mathcal{T}_e$. A lookup dictionary $\mathcal{T}$ is constructed mapping $[\mathrm{E_s}^{(s)}, f]$ to $\mathcal{T}_e$.

We now discuss the details of the Algorithm 1. The input to the recognizer is a set of tokenized fragments and a grammar G. The CFG is compiled to generate $\mathcal{T}$ which is used during parsing. The algorithm proceeds by initializing the first earley set $\mathcal{S}(0)$ and priority queue q with start states involving the root r transitions of the CFG. The priority queue uses the confidence (c) of the *earley state item* as its key.

In our implementation, we do not require an explicit prediction step as we leverage the transition lookup $\mathcal{T}$. The

Algorithm 1 Generalized Earley Recognizer

```
function COMPLETER($I, \mathcal{S}, j$)
    for $f$ in $I$.state.fragments do
        return $I$(TRANSITION($\mathcal{S}(I$.origin$), f$))
    end for
end function

function SCANNER($f, \mathcal{S}, i$)
    $f \leftarrow$ transition fragment
    return $I$(TRANSITION($\mathcal{S}(i - f$.length$), f$))
end function

function TRANSITION($\mathcal{S}(j), f$)
    $\mathcal{T} \leftarrow$ pre-computed lookup earley transitions
    for each $\mathrm{E_s}$ in $\mathcal{S}(j)$.states do
        yield return $\mathcal{T}[\mathrm{E_s}, f]$.target
    end for
end function

function RECOGNIZE($t, G$)
    $t \leftarrow$ set of input tokens
    $G \leftarrow$ context-free grammar
    sort $t$ {by their end index}

    Initialize: $I(\mathrm{E_s}^{(r)}) \leftarrow [\mathrm{E_s}^{(r)}, 0, \emptyset, 1.0]$ {earley
    state item with root transition and $c = 1.0$}
    $\mathcal{S} \leftarrow \emptyset$ {initialize empty earley state sets}
    $q = PriorityQueue()$
    $q.push(I(\mathrm{E_s}^{(r)}))$
    $\mathcal{S}(0).add(I(\mathrm{E_s}^{(r)}))$

    for $j = 0, j \leftarrow j + 1$ while $j < \mathcal{S}.length$ do
        while $q \neq \emptyset$ do
            $I \leftarrow q.pop()$
            $\mathcal{I} \leftarrow$ COMPLETER($I, \mathcal{S}, j$)
            for each $I_c$ in $\mathcal{I}$ do
                $\mathcal{S}(j).add(I_c)$
                $q.push(I_c)$
            end for
        end while

        $\mathcal{F} \leftarrow \{f | f \in t, f.\mathrm{EndIndex} == j + 1\}$
        for $f$ in $\mathcal{F}$ do
            $\mathcal{I} \leftarrow$ SCANNER($f, \mathcal{S}, j$)

            for each $I_s$ in $\mathcal{I}$ do
                $\mathcal{S}(j + 1).add(I_s)$
                $q.push(I_s)$
            end for
        end for
    end for
    return $\mathcal{S}$
end function
```

recognizer sequentially processes fragments from left to right. At each step, elements in the queue are processed using the *completion* operator and any earley state items formed are added back to the earley set and priority queue. Next, we gather all tokens ending at the upcoming index (there can be multiple fragments due to ambiguous tokenization). These fragments are used to generate other earley state items using the *scanning* step which are again added back to the queue. The details of the scanner and completer are discussed below.

- **Scanner**: The input to this module is the array of earley state sets $\mathcal{S}$, transition fragment f and current index i. To support ambiguous tokenization, transition operation needs to be performed on the source earley state set $\mathcal{S}(i - f.length)$. The module returns earley state items I linked with the states $[\mathcal{S}(i - f.length) \times f]$.
- **Completer**: Given an earley state item, this module checks whether the state associated with the item contains any transitions which generate any other earley state item in the origin earley state set. This module returns earley state items linked with states $[\mathcal{S}(I.origin) \times f], \forall f \in I.\text{state.fragments}$.

Proof of correctness

We prove that our generic earley recognizer is correct by showing that the start earley set is the same and the three steps prediction, scanning and completion are consistent with the earley recognizer.

Lemma 4.1 Every earley item corresponds to an Earley state item $I(\mathrm{E_s})$.

Proof: An earley item is defined as $[d_r, i]$ where dotted rule $d_r \in \mathcal{R}$ and i is the origin position. Every d_r is part of at least one earley state $\mathrm{E_s}$ in the ϵ-DFA. As earley state items are directly linked to a state $\mathrm{E_s}$ and origin position i, an earley item is always linked to an earley state item.

Lemma 4.3 Earley State Set $\mathcal{S}(0)$ initialization is consistent in generic earley recognizer

Proof: In earley algorithm, dotted rules associated with grammar root r's transitions are added. Root transitions are encapsulated in the start state ${\mathrm{E_s}}^{(r)}$ in the ϵ-DFA. Earley state item linked to $\mathcal{S}(j)$ is added in our algorithm. ${\mathrm{E_s}}^{(r)}$ has all the dotted rules in the start state and therefore $\mathcal{S}(0)$ initialization is consistent.

Observation 1: The *prediction* step is consistent with the ϵ- DFA algorithm as the same dotted rules in the original algorithm are present in $\mathcal{S}(j)$ by Lemma 4.1.

Lemma 4.3 Scanning is consistent while handling ambiguous tokens

Proof: Scanning is consistent till the recognizer encounters an ambiguous token as the start earley state set is the same by Lemma 4.2. Ambiguous tokens f_a add additional earley state items to $\mathcal{S}(j)$ without acting on any other state items in the same set. Addition of extra state items only result in more parse trees being reconstructed starting from the end state $\mathcal{S}(N)$, thereby scanning remains consistent.

Observation 2: The *completion* step is unchanged from the earley algorithm and thereby is consistent.

Proposition 1: *Generic earley recognizer (GER) is correct*. We prove this proposition by using Theorem 1.

Theorem 4.4 $L_{\mathrm{GER}}(\mathrm{G}) = L(\mathrm{G})$

Proof: $L(\mathrm{G})$ is the language defined by the grammar G. GER starts with the same earley state set $\mathcal{S}(0)$ by Lemma 4.2, and the processing stages remain consistent by Lemma 4.3, Observation 1 & 2. GER thereby produces $\mathcal{S}(N)$ with the same set of dotted rules by Lemma 4.1. For an input $l \in L(\mathrm{G})$, to be recognized the following condition needs to be satisfied

$$[r, 0] \in S(N) \tag{6}$$

The earley item with the root rule r and origin 0 must be present in Earley set S. By Lemma 4.1 we can show that the corresponding earley state item is present in final earley state set.

$$I({\mathrm{E_s}}^{(r)}, 0) \in \mathcal{S}(N) \tag{7}$$

Therefore it follows that

$$L_{GER}(\mathrm{G}) = L(\mathrm{G}) \tag{8}$$

4.1. Leo reduction

For right recursive grammars, often when a dotted rule gets completed it may complete other rules to form a deterministic chain of the form $(A \to \gamma\bullet) \to (B \to \alpha A\bullet) \to (C \to \beta B\bullet)$. While processing earley sets sequentially we may encounter the same chains frequently leading to increased earley state size and processing time. Leo [2] suggested detecting such transitive steps and adding $(C \to \beta B\bullet)$ directly to the current earley set. We have encapsulated this information along with earley state sets to *memoize* transitive steps $(\mathcal{T}_s)$. Whenever a state item is added to the set, $\mathcal{T}_s$ a mapping from child to parent $[I \to I.\text{parent}]$ is added in the following manner $\mathcal{T}_s[I].add(I.\text{parent})$.

The implementation of the reduction process is described in Algorithm 2. The LeoReduction() function is called for earley state items obtained from the completer module. Then, we perform a DFS based search starting the from the parent earley state set if only there is an unique parent present. This module returns the **topmost parent** of the deterministic chain [2].

Algorithm 2 Leo Reduction Implementation

```
function LeoReduction($I, \mathcal{S}$)
    for rule in $I$.state.$\mathcal{R}$ do
        if rule.Complete then
            $parent \leftarrow I$
            while $\mathcal{S}(I.\text{origin}).\mathcal{T}_s$ has $I$ and
                $\mathcal{S}(I.\text{origin}).\mathcal{T}_s[I].\text{Count} == 1$ do
                $parent \leftarrow \mathcal{S}(I.\text{origin}).\mathcal{T}_s.\text{First}$
            end while
        end if
        if $parent \neq I$ then
            yield return $parent$
        end if
    end for
end function
```

In the experiments we performed we found out that in data normalization tasks using natural language, right-recursion *seldom* arises in CFGs. In most cases, such right-recursion can be refactored into a left-recursive grammar. Although Leo-reduction is useful in right-recursive grammars with long chains, it adds an *overhead* cost which affects the performance of other grammars significantly. Therefore, we have chosen to keep Leo-reduction as an optional feature and instead choose to alert users when the grammar is right-recursive.

4.2. Detecting right-recursion

Consider a CFG G that contains a right-recursive production rule of the form $A \rightarrow \alpha A$. The earley algorithm will proceed by processing E_s having $(A \rightarrow \bullet\alpha A)$, to E_s' having $(A \rightarrow \alpha \bullet A)$ and back to E_s having $(A \rightarrow \bullet\alpha A)$, creating a loop. Therefore, detecting right-recursion effectively boils down to detecting loops in ϵ-DFA post grammar compilation. Loops are detected by DFS traversal across an earley state transition graph for cycle detection. Self-loops are ignored as they don't necessarily mean right-recursion and can also be generated by a left-recursive rule of the form $A \rightarrow A\alpha$ or an identity production $A \rightarrow A$.

4.3. Parse tree reconstruction

Parse tree reconstruction process is shown in Algorithm 3. We search in the final earley state set for an earley state item which is linked with an end state. If there are multiple candidates, we select the one with highest confidence. We then traverse to the parent earley state items in a depth first manner. GetParseTree() in Algorithm 3 returns a set of earley state items involved in completion of the earley *recognition*. The parse tree can be easily derived from this as earley state items encapsulate information about dotted rules and the last matched fragment.

Algorithm 3 Parse tree reconstruction

```
function GetParseTree($\mathcal{S}$)
    $\mathcal{E}' \leftarrow \emptyset$
    for $I$ in $\mathcal{S}(N).\mathcal{E}$ do {$N$: # of tokens}
        if $I$.state.IsEndState() then
            $\mathcal{E}'.add(E_s)$
        end if
    end for
    sort $\mathcal{E}'$ {descending order of their confidence}
    $item \leftarrow \mathcal{E}'(0)$ {best parse}
    $parse \leftarrow \emptyset$
    while $item.f$ do
        {add parent earley state item}
        $parse.add(I(item.E_s'))$
    end while
    return $parse$
end function
```

5. Pattern search

We observed that many context-free grammars authored for real-world data normalization tasks seemed to suffer from performance issues. The parse trees formed in those samples appeared to be quite deep resulting in *large* earley state sets and thereby *increased* processing time. Upon closer inspection, we found out a common pattern among these grammars. One example of such a grammar is shown in Table 5.

Where, [*] matches any fragment and [Weight] is the root of the grammar defined in Table 2. Such grammars result in

(1)	[Noise] = [*]
(2)	[Noise] = [Noise][*]
(3)	[Root] = [Noise][Weight]
(4)	[Root] = [Weight][Noise]
(5)	[Root] = [Noise][Weight][Noise]

Table 5: Context-free grammar for pattern search.

deeper parser trees as the dotted rule [Noise] keeps expanding. For input having a larger number of tokens (~100) the execution time increases steeply as time complexity is $O(N^3)$. From the grammar presented it is evident that the user is trying to search for the pattern [Weight] from a noisy text phrase. We propose two approaches to tackle this problem which provides a much better real-time performance.

5.1. Containment parse

Containment parse uses the base pattern [Weight] defined in Table 2 and tries to search it in the input text. The intuition behind this approach is that if we run earley parsing on substrings of the *smaller* length instead of the entire text phrase, the complexity is much less (for N=100, $O(N^3) \sim 10^6$, while if we split it into 10 parts, $N^2 O(N^3) \sim 10^5$). However, naive substring matching is slow and as the length of the pattern match can vary the search space is large.

Algorithm 4 Containment Earley Initialization

```
function ContainmentInitialize($\mathcal{S}$)
    for $i$ in $\mathcal{S}$.length do
        $\mathcal{S}(i).add(I(\mathrm{E_s}^{(r)}))$
    end for
end function
```

Instead, we plan to leverage the earley state set to figure out text segments of interest. Suppose after processing earley state sets $\mathcal{S}(0) \to \mathcal{S}(j)$, we find that $\mathcal{S}(j) = \emptyset$. This signifies that parsing the pattern from $[f(0), \ldots, f(j)]$ was unsuccessful and we can reset $\mathcal{S}(j) \leftarrow \mathcal{S}(0)$. The problem with this approach is that it will miss grammar matches if the actual match is from $[f(i), \ldots, f(k)], i < j \ \& \ k > j$. To overcome this we initialize all earley state sets with root earley state items as shown in Algorithm 4.

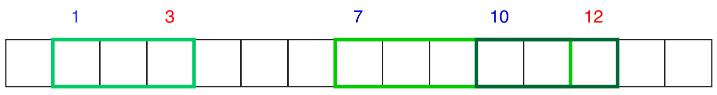

Figure 2: Visualization of containment parsing. The figure shows scenarios where there are multiple pattern matches including overlapping patterns. The labeled indices show the *start* and *end* positions of the pattern match.

Containment parse works well only for a sub-section of context-free grammars of the form presented in Table 5. Complications arise when multiple/overlapping occurrences of the root pattern are present in the input pattern shown in Figure 2. Our system handles this scenario by returning all possible parses from the text and it is up to the user to choose the best one. One disadvantage of using containment parse is that it cannot handle fully overlapping pattern matches for ambiguous grammars, as by design GetParseTree() in Algorithm 3 returns the shortest match.

5.2. Earley state item validator

Apart from deep parse trees, another reason why execution time is high at times is that unsuccessful attempts at parsing text phrases runs too long. For example, if you have a grammar to parse person names as shown in Table 6. Now if you encounter an alphanumeric fragment, the parsing should ideally exit as proper nouns cannot be *alphanumeric*.

(1)	[Name] = [*]
(2)	Name] = [Name][*]
(3)	[PersonName] = [Mr.][Name]
(4)	[PersonName] = [Mrs.][Name]
(5)	[PersonName] = [Ms.][Name]

Table 6: Person name grammar.

We propose the idea of having an optional validator function associated with production rules. The validator function is executed when an earley state item with the dotted rule is completed. The Completer() returns the earley item only if the validation is *successful*.

The main idea is to stop processing earley state items where we already have some cues that it is *not* going to yield the end earley state, where the root production is complete. This helps reduce execution time for long noisy inputs. *Validators* are a powerful tool as they are not limited to any sub-category of input CFG and can be leveraged to enhance the performance of any grammar.

As we use generic fragments, validators can be utilized to enable a broad range of applications. One of the popular use-cases we have encountered is during parsing HTML source code. Using our custom *Webpage Fragment*, validations like header tag check, list tag presence, blacklisted HTML tags, etc. have been authored for various applications. Validator incorporation is also a customizable feature and a default pass-through validator is added to every production rule.

6. Features

In this section, we will discuss some of the salient features of the end-to-end parsing system which makes it a powerful tool for developers. As we designed our system for data normalization for web-scale applications, we built features to satisfy fundamental software engineering requirements like *versioning*, *reusability*, *visualization* and *debugging*.

6.1. Versioning and reusability

Grammars are stored in XML format and are versioned. Our tool allows *hierarchical* grammar authoring which means you can directly use *non-terminals* from a dependent grammar without having to define them. This enables users to *refactor* grammars similar to code, and reusability helps users write more complex grammar.

6.2. Type-aware curation

Entity linking is another powerful feature that has been leveraged in our tool for authoring type-aware grammar. An example of the same is shown below in Table 10.

Fragment/Annotations	Start	Length
Love and Theft	0	1
[infobox_album]	0	1
-	1	1
Bob Dylan	2	1
[infobox_person]	2	1

Table 7: Example of entity-linking based type curation. $1 and $2 are the wikipedia links of the given entities respectively.

Selected Sample: <a href="$1">Love and theft</a>
- <a href="$2">Bob Dylan</a>

A set of URLs can be configured for entity linking for being identified as URL fragments and Wikipedia infobox information has been used to detect the type of the entity detected. The types are used as annotations to write grammars to detect *Grammy nominations* from Wikipedia pages eg: [Root] = [infobox_album] - [infobox_artist].

6.3. Visualization and debugging

Our tool is equipped with enhanced visualization and debugging capabilities. Few features in this domain that are supported by our tool are:

- Parse tree visualization for input samples.
- Partial parsing of input text segments.
- Right-recursion checking in grammar and ϵ-DFA visualization.
- Capture certain non-terminals of the grammar and visualize the output of the same for sampled input phrases shown in Figure 4.
- Ability to cloud save grammars for future use.

7. Experiments

Our system is developed using C# and the user interface is provided as a windows app. Our implementation provides a platform to author as well as visualize the parsing results in the same framework.

The grammar authoring tool has been used by a significant number of developers parsing millions of unstructured/semi-structured textual information and it has been tested using 50+ real-world grammars. We will show the performance results of the containment grammar and compare it with the vanilla implementation.

We conducted our experiments using the Date-time CFG (details mentioned in Appendix A Table 8). The grammar was tested on input texts of varying fragment sizes, the input dataset is shown in Appendix A Table 9. We report the recognizer execution time averaged over 100 trials for each input entry. We have achieved an amortized execution time improvement of ~60% over the dataset samples.

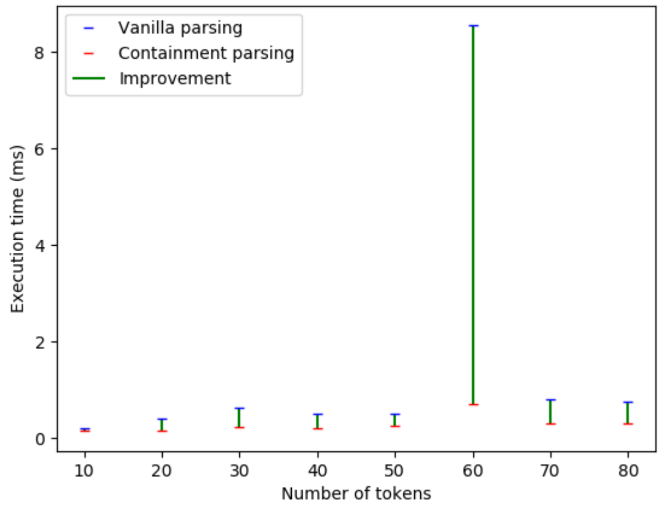

Figure 3: Performance improvements for containment grammar over vanilla implementation.

We notice that for one of the input samples (token size = 60), the improvement is 10x. The reason behind this is the presence of multiple pattern matches and a match present at the very end of the input string. Normal parsing produces a very deep parse tree leading to long backtracking during earley set processing. For many production grammars containment parsing has shown up to 5x-10x speed enhancements. In general the speed enhancements are significant in case of recursive grammars with long trails of [Noise] fragment matches.

8. Conclusion

In this paper we present an end-to-end grammar authoring tool for data normalization involving natural language. We introduce the concept of annotations & ambiguous tokenization and show how they aid natural language parsing. We present our implementation of the generic earley recognizer which supports ambiguous tokenization. We also analyze a few sub-categories of grammar and present approaches to improve performance in such scenarios. Finally, we explore the features of the software tool that is used for grammar authoring. Grammar-based parsing techniques haven't found much popularity in natural language-based applications and literature involving parsers have remain limited to theoretical results only. We believe our work has explored various real-world application domains aimed at natural language normalization using parsers. We hope that our work will encourage further exploration of parsing-based data normalization.

9. Appendix

Table 8 describes the date-time context-free grammar used in the experimental setup. For vanilla parsing [Root] is used as the root of the grammar, [DateTime] has been used as the root for containment parsing.

(1)	[Noise] = [*]
(2)	[Noise] = [Noise][*]
(3)	[Root] = [Noise][DateTime]
(4)	[Root] = [DateTime][Noise]
(5)	[Root] = [Noise][DateTime][Noise]

Table 8: Context-free grammar for pattern search.

Table 9 shows the input dataset on which the experiments were performed. Only 5 of the 8 inputs yield successful parses (inputs with token size: 10, 40, 50, 60, 70). The pattern matches in the text are shown as [*match*].

Input text	# of tokens
"Little Joe, the Wrangler", written by Thorp himself. In [*1910*]	10
"Sarantaporos" (Greek: is a mechanized infantry brigade of the Hellenic Army"	20
"superbase" with flights of F/A-18F Super Hornets, F-35 Lightning II, KC-30A, C-17 Globemaster. No. 35 Squadron (Operating C-27J Spartan) is planned to move to the base from RAAF Base Richmond	30
"Potential Breakup Song" is a song co-written and performed by American duo Aly & AJ, and is the first single from their second studio album, Insomniatic. It was a very successful single in the summer and early fall of [*2007*]	40
"The Historical Status of China's Tibet" is the title of a Chinese Government propaganda publication that attempts to substantiate China's claim that Tibet is an inalienable part of China. This book was originally published in Chinese and distributed widely within China. It was awarded the Excellent Book Award in [*1996*]	50
"([*2005*] film) Hazeis a [*2005*] Japanese thriller / horror film written and dire Haze ([*2005*] film). Haze is a [*2005*] Japanese thriller / horror film written and directed by Shinya Tsukamoto who also stars in the movie. After appearances at several international festivals in 2005, the film debuted theatrically in Japan on [*March 4, 2006*]."	60
"Class I railroad, operating between Waukegan, Illinois and Gary, Indiana. The railroad served as a link between Class I railroads traveling to and from Chicago, although it operated almost entirely within the city's suburbs, and only entered Chicago where it served the U.S. Steel South Works on the shores of Lake Michigan. Nicknames for the railroad included "The J" and "The Chicago Outer Belt Line". At the end of [*1970*]	70
"16th-century fort situated in heart of the Vellore city, in the state of Tamil Nadu, India built by Vijayanagara kings. The fort was at one time the headquarters of the Aravidu Dynasty of the Vijayanagara Empire. The fort is known for its grand ramparts, wide moat and robust masonry. The fort's ownership passed from Vijayanagara kings, to the Bijapur sultans, to the Marathas, to the Carnatic Nawabs and finally to the British, who held the fort until India gained independence"	80

Table 9: Input text information used for parsing. The grammar pattern match for each of the input text is shown as [*match*]. The second, third and last inputs are not recognized by the parser given the date-time grammar.

References

[1] J. Aycock and R. N. Horspool, “Practical earley parsing,” *The Computer Journal*, vol. 45, no. 6, pp. 620–630, 2002.

[2] J. M. Leo, “A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead,” *Theoretical Computer Science*, vol. 82, no. 1, pp. 165–176, 1991.

[3] J. Earley, “An efficient context-free parsing algorithm,” *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[4] J. Cocke, *Programming Languages and their Compilers: Preliminary Notes.* New York University, 1969.

[5] S. C. Johnson et al., *YACC: Yet Another Compilercompiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[6] G. D. Forney, “The viterbi algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.