Computer Science

Capstone Report - Fall 2022

Using context-free grammar to implement a simple Chinese parser in Haskell

Daisy Huynh,
Raiyan Reza

supervised by
Professor Paul-André Melliès

Preface

Both authors of these papers are undergraduate Computer Science majors at New York University Shanghai. The work is borne out of keen interest in the subtle and marvelous field of formal languages, parsing algorithms, theory of computation, and natural language processing.
Interested readers will find this work a gentle introduction to the art and science of parsing and find the simple implementation of a well-known Parsing algorithm a worthy read.

Acknowledgements

I am immensely grateful to Professor Paul-André Melliès who supervised our work with boundless patience, unwavering enthusiasm, and immense compassion. I am equally grateful to my teammate , Anh Nhat Huynh, for being the superhuman she has been throughout the course of this project. Somehow while training CNN networks, coding a shell for an operating system, and grappling with Dijkstra's algorithm, she found time to work on building an Earley Parser as well.
I also want to express my debt of gratitude to the people who keep me sane behind the scenes. Foremost among them, my Dad, Syed Musa Reza, my Mom, Syeda Taufatur Reza, and my little brother, Rastin Reza.
And, then there are others too. So, in no particular order I want to acknowledge:
Emma Rosensaft, Karolina Czop, Hawra Alakhras, Carroline Makoni, & Nusmila Lohani.
Thank you just for existing.
-Raiyan Reza.

Abstract

Parsing a sentence can be defined as associating a sentence with parse trees. These trees can encode information about the grammar of the sentence. Thus, a parser can be a powerful tool in processing grammatical features of sentences. Our aim, therefore, is to create a primitive parser that can identify the parse trees of very simple sentences in Mandarin using context-free grammar, a powerful class of grammars that are well known for their capacity in approximating a subset of grammar of natural languages.

Keywords

NYU Shanghai Capstone,**Earley Algorithm**, **CYK Algorithm** ,**Parsing**, **LR(k) Algorithm**, **Context-Free Grammar**, **Formal Languages**,**Chomsky Hierarchy**

Contents

1 Introduction

A simple definition of parsing is as follows. Given an input sentence, parsing entails ascribing a structure to the input sentence[1]. Usually, the structure is encoded in the form of a parse tree [1].

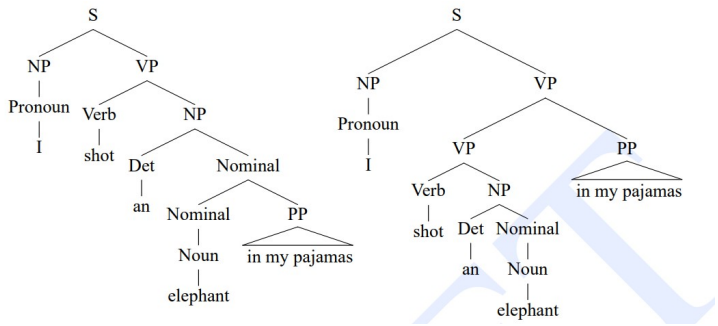


Figure 1: Two Possible Parse Trees For A Simple English Sentence

The diagram above 1 taken from the excellent introductory textbook on Natural Language Processing by D. Jurafasky and J. H. Martin [2], illustrates the two possible parse tree one can attribute to the first line in the sentence Groucho Marx utters in *Animal Crackers*: "I shot an elephant in my pajamas. How he got in my pajamas I don't know." Without the second line to disambiguate the first line, two possible parse trees can be ascribed to the sentence, "I shot an elephant in my pajamas."

These motivating examples illustrates two facts. First, parse trees can capture the syntactic structure needed to encode semantics. The meaning of the sentence changes depending on how one situates "elephant" in the parse tree: either part of the nominal in a noun phrase, as shown by the first parse tree, or a noun in a verb phrase, as shown in the second parse tree. Second, the innate ambiguities of natural languages are captured when one can generate more than one parse tree for a given sentence. As a matter of fact, we can formalize the definition of ambiguity of a sentence by the number of distinct parse trees we can generate for it[3]. We can, therefore, state the degree of ambiguity of a sentence as the number of distinct parse trees we can ascribe to the sentence [3]. An unambiguous sentence likewise would be a sentence which has a degree of ambiguity 1. These two points mean a parsing algorithm must be both correct, generates the right parse tress, and complete, generates all the parse trees.

The formalism underscoring parsing itself originates from Noam Chomsky's work on generative grammar back in 50's. The resultant and relevant literature gives us a definition of formal languages in the following form:

- N, a finite set of nonterminal symbols. A symbol is defined as nonterminal if it can be substituted.
- T, a finite set of terminal symbols. A symbol is defined as terminal if it cannot be substituted.
- R, a finite set of written rules. For example, A$\rightarrow$ aBc, where A,B $\in$ N, and a,c $\in$ T
- S, a special terminal symbol that marks the first symbol.

The formalism can be made more precise and rigorous but is sufficiently well-defined for the purpose of this paper and our work. Interestingly, by applying restrictions on the set R it is possible hierarchies formal languages, as is done in the famed Chomsky Hierarchy, into different types commonly referred to in the literature as Type-0, Type-1, Type-2, and Type-3 languages. These different types of formal languages we can derive will satisfy the following relationship: Type-3 $\subseteq$ Type-2 $\subseteq$ Type-1 $\subseteq$ Type-0. Even a more striking result is the fact that each type of formal language has correspondence with a formal model of computation. Type-0 languages are equivalent to a Turing Machine, Type-1 languages to linear bounded automation, Type-2 languages to push-down automation, and Type-3 languages to finite-automation.

This intersection between formal languages and theory of computation provides a rigorous framework to apply rule-based parsing to a subset of natural languages. In particular Type-2 languages, also known as context-free languages, are a good formal model for representing a subset of grammar in real world languages. A simple explanation for the appropriateness of context-free grammar is the fact that its grammar is expressive enough to capture nuances and ambiguities of natural languages but sufficiently restricted to lend itself to parsing algorithms with reasonable time and space complexities[1].

Thus, there has been seminal algorithms for parsing context-free grammar. Among the, some well known algorithms are Cocke-Younger-Kasami (CYK), Earley Parsing algorithm, and LR algorithm [3]. Earley Parsing algorithm is itself well-suited for paring context-free because it is more general and acceptable run time. The CYK algorithm requires prepossessing of inputs into Chomsky Normal Form [3]. Whereas, the LR algorithm cannot handle ambiguities[3].

2 Related Work

2.1 Big Picture & History

Parsing in and itself is a subject of interest in linguistics, computer science, and other relevant fields. Nevertheless, we can situate this subject in the much broader context of historic and contemporary research on recreating and generating human language capacities in machines. One landmark publication on the subject matter is, “Computing Machinery and Human Intelligence". Published in 1950, in this pioneering essay, Turing envisions the possibility of computers mimicking and mastering human speech to the point of deceiving a human interlocutor into the impression that a machine is a fellow flesh and blood human being [4]. Turing asserts that if the human interlocutor cannot discern a human being and machine apart, then there is no valid point making a distinction between a human being and machine anymore. Turing’s bold declarations are far from universally accepted and has been subject of philosophical rumination, academic debate, and popular portrayal in the media ever since its original formulation and presentation. But the salient point is, “Computing Machinery and Human Intelligence" is a lucid description and realization of the centrality of human language to human thinking and reasoning, the possibility of computers to generate and understand human communication, and implications these developments may have.

Turing addresses the hypothetical scenario of a machine having fully mastery of the human language in the first place because of the developments of the time and era that made this scenario worthy of consideration. In the 1940’s and 1950’s earlier works have led to two foundational paradigms: automation and probabilistic and information-theoretic models [2].

Regarding the former, it was a direct consequence of earlier formalism defining computers that were introduced by Turing himself in 1936 [2] and other relevant works. The model of McCulloch-Pitts neuron introduced in the 1940’s modeled neurons as a computing element that could compute propositional logic [2]. In the same era, Claude Shannon, used probabilistic models of discrete Markov processes to automation for languages [2]. This in term inspired Chomsky in the 50’s to model grammar based on finite-state machines. By 1956, Chomsky has advanced enough in his endeavor to introduce context-free grammar or Type-2 languages and in 1959 John Backus independently discovered equivalent results [2].

Context-free grammar and context-free parsing algorithms are, therefore, direct offsprings of the historic foundational paradigms of artificial intelligence, natural language processing (NLP)

and other relevant fields. Even without application to natural language processing, context-free languages and context-free parsing garners attention among computer scientists. An obvious reason is that formal languages, context-free grammar, and parsers can be natural tools for modeling programming languages and devising algorithms to parse them [1]. Parsing context-free grammar is thus a world of research unto itself.

2.2 Some Well-Known Parsing Algorithms

LR(k), Cocke-Younger-Kasami (CYK), and Earley Parsing algorithm are three landmark algorithms for parsing context-free languages. What follows below would be an informal description of these algorithms, followed by tabulation of key points worthy of comparison between these algorithms.

LR(k)

The notation LR(k) stands for *a language translatable from left to right with bound* k[5]. Therefore, LR(k) parsers are parsers that apply to a subset of context-free languages because not all context-free language will meet the LR(k) criterion. To define the term LR(k), it is necessary to introduce some additional formalism. Let α be a string comprising of terminals and non-terminals characters. Then, $\alpha \implies \beta$ denotes the fact that: $\alpha \rightarrow \alpha_0 \rightarrow \alpha_1 \ldots \rightarrow \alpha_n = \beta$ [5]. Recall, that S is a special terminal symbol. The literature also refers to it as the root symbol for reasons that will be made clear. Now, any string α is called sentential form if $S \implies \alpha$. Now, consider the following parse tree cited as a motivating example in [5].

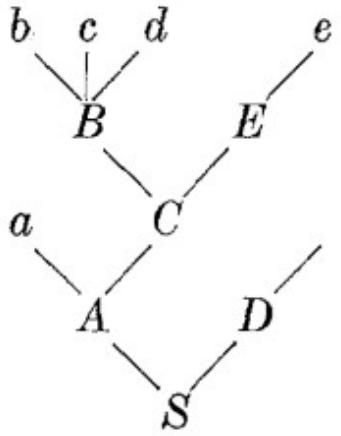


Figure 2: A Parse Tree With Root S

The handle of any such parse tree, of which 2 is just an example, would the leftmost set

of adjacent leaves forming a complete branch [5]. In other words, if we have a sequence of characters $X_1 \dots X_t$, where each X_i is either a terminal or nonterminal, forming the leaves of a tree, a handle would be the smallest k to provide us with the following for some j and Y as visually depicted below from [5]:

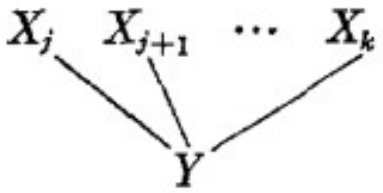


Figure 3: A Handle In A Parse Tree

An LR(k) parser will be a bottom-up parser, starting with the leaves, identifying the handles, and pruning them off to reach the root character. To make this more concrete, the parse tree in 2 corresponds to the following grammar:

$$S \to AD, A \to aC, B \to bcd, C \to BE, D \to \epsilon, E \to e \tag{1}$$

In (1) the standard conventions apply. Upper-case letters represent nonterminals, lowercase letters represent terminals, ϵ stands for an empty string, and Greek letter would be string of having any number of nonterminals and terminals. 2 is the parse tree generated for the input string “bcde" and one of the possible sequence of derivation for it would be:

$$S \to AD \to A \to aC \to aBE \to abcde \tag{2}$$

An LR(k) algorithm pruning the leftmost handle using bottom-up recursion will generate the derivation sequence in (2) in reverse. When given the input, only the leaves will be given. Skipping some of the more precise formalism for sake of brevity, LR(k) algorithm can only work if the handles are unique. More precisely, a grammar is LR(k), that is can be parsed by LR algorithms, if and only if each handle in its parse is uniquely identifiable by the string to its left and the k terminals to its right [5]. Therefore, LR(k) algorithms work for unambiguous context-free grammar. But, when it is applicable it attains a linear asymptotic time-bound for worst-case scenario of $O(n)$ where n is the length of the input

string.

CYK algorithm

Beneath is a pseudo-code description of a CYK recognizer, which with some minor modifications can be converted into a parser[2].

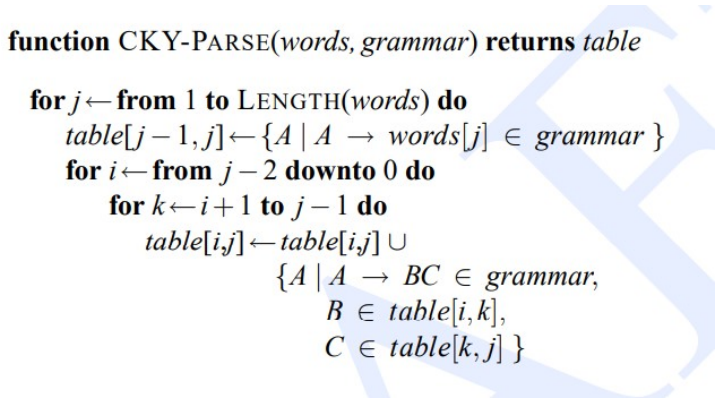


Figure 4: CYK psuedo-code

The paramaeter words stand for a input string and grammar is the set containing the production rules. So, the words argument can take the form, "Book the flight through houston." [2] and each individual word will be "Book", "the", "flight", etc. The algorithm recursively builds up a triangular matrix encoded as a table, where each cell [i,j] where i is the row and j the table represents all the constituents of the sentence corresponding to the non-terminals from the ith word to the jth word. The index is at zero, so the indexing here for the previous sentence would look like "$_0$I$_1$am$_2$not...". The recognizer will recognize an input to be a member of a context-free language if the [0,N] cells contains the special start symbol.

The grammar in this case will be a simplified set of English grammar, defined as follows and along side the grammar you can see the output table[2]:

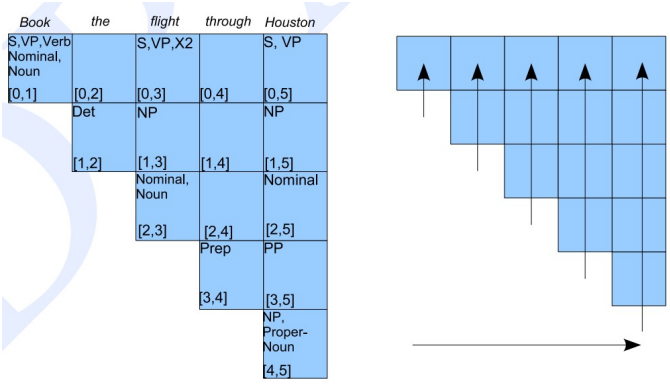
$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$

$S \rightarrow VP$

$NP \rightarrow Pronoun$
$NP \rightarrow Proper\text{-}Noun$
$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$
$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$
$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$

$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$

(a) English Grammar



(b) CYK Recognizer Output

Figure 5: A simplified English Grammar and Parse Table

With some additional steps, this information can be used to generate a parse for the input sentence. The CYK algorithm can generate its output in the worst case time bound of $O(n^3 \times |G|)$, where n is the length of the input string and $|G|$ the cardinality of the grammar set. An additional important point one needs to keep in mind is that CYK algorithm requires preprocessing of input strings so that the input string takes the form of Chomsky Normal Form (CNF). While it is an established fact that any language of Context-Free Grammar can be rewritten in CNF format, this means that CYK algorithm requires an additional step before we can run the algorithm.

Earley Parsing Algorithm

A key element of Earley Parsing Algorithm are Earley Items. A common representation of Earley items in the literature are as follows: $[A \rightarrow \alpha \bullet \beta, j]$ [6]. The $\bullet$ represents a rule's right-hand side and j is a pointer to a set S_j. Each set, S_i,is a set of Earley Items, and this set is known as an Earley Set. Given an input string of length n, the Earley algorithm will generate $n + 1$ Earley Sets. The Earley Set, S_0, will initially have only one Earley Item, $[S^{\text{`}} \rightarrow S\bullet, 0]$. An input string that is successfully parsed must have the Earley item $[S^{\text{`}} \rightarrow \bullet S, 0]$ in the set, S_n.

Suppose that the input string is of the form $X_1 \ldots X_n$, where each X_i is either a terminal

or nonterminal, then an Earley parser will generate the Earley sets S_0, initialized with the value [$S^{\text{`}} \rightarrow S\bullet$,0], to S_n by using any combination of these rules [6]:

- SCANNER: If $[A \rightarrow \ldots \bullet a \ldots, j]$ is in S_i and $a = x_{i+1}$, add $[A \rightarrow \ldots \bullet a \ldots,$ j] to S_{i+1}
- PREDICTOR: If $[A \rightarrow \ldots \bullet B \ldots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$
- COMPLETER: If $[A \rightarrow \ldots \bullet, j]$ is in S_i, add $[B \rightarrow \ldots A \bullet \ldots$,k] to S_i for all items $[B \rightarrow \ldots \bullet A \ldots,$ k] in S_j

To get a better idea of how Earley parsing works, suppose a context-free language is defined by the following grammar [6]:

$$E \rightarrow E + E, E \rightarrow n \tag{3}$$

Furthermore, suppose the Earley parser is fed the string: “n+n". Then the algorithm will generate the following output [6]:

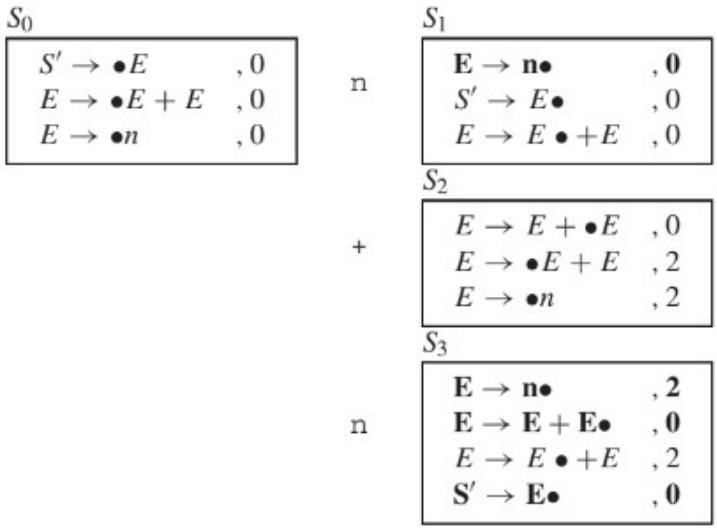


Figure 6: Earley Sets for the input n+n

The information from these Earley sets can be gathered to generate a parse tree. The algorithm does not require any preprocessing, can handle ambiguous grammar, and operates in the worst-case time bound of $O(n^3)$, where n is the length of the input string.

2.3 LR(k) vs CYK vs Earley

LR(k), CYK, & Earley Comparision		
Algorithm	Restriction	Worst-Case Time Complexity
LR(k)	Applicable To Unambiguous Grammar	$O(n)$
CYK	Accepts Input CNF Form	$O(n^3 \times \|G\|)$
Earley	None	$O(n^3)$

Table 1

Table 1 summarizes the key point of comparision between the different parsing algorithms. From these information should be self-evident why we selected Earley-parsing algorithm for our work.

2.4 Parsing Mandarin Using Earley Parser

X. Xia el.[7] greatly informed our work. Their work establishes that innate features of Mandarin grammar posses difficulty for naively application Earley Parsing algorithm. The syntactic flexibility of Mandarin and the absence of morphological markers generates too many parses per sentence, some of the parses which are nonsensical.

Thus, they extend the formalism of context-free parsing to cut down the parse forest naively using context-free grammar and parsing will generate. However, these modifications are beyond the scope of our work and merely demarcate the boundaries of technical feasibility for the project we have in mind.

2.5 Functional Programming and Haskell

Functional programming directly descends from λ-calculus, which is a formal model of computation that is equivalent to a Turing machine [8]. Examples of functional programming language are Lisp, Haskell, and Erlang. Unlike, imperative languages, functional programming languages do not mutate or modify the state of a variable. An immediate advantage of this is that side-effects are avoided entirely in a purely functional programming language like Haskell. To be more technical, functional programming language gives the benefit of *referential transparency* [8]. A function given the same values will always evaluate the exact same results.

Moreover, the building block of a functional programming language are expressions, which are concrete values, variables, and other functions, where functions can informally be defined as expressions that apply to inputs and reduce them. This gives a functional programming high degree of abstraction and composability, which among other things, leads to neat compact and minimalist codes [8].

Other piece of literature suggests functional programming may be more efficient[9]. But to us the most attractive feature of Haskell is it's closeness to formalism of computer science. Therefore, a parser written in Haskell will better exemplify the ideas underscoring a parser.

2.6 Our Work

In the context of the existing literature, our work aims to replicate earlier results using simpler tools and models. In essence, our parser will be a toy parser that rightly codes Earley algorithm, achieve its time and space complexity, and succeeds in parsing a limited subset of Mandarin.

3 Solution

Our first step was to provide a reasonable argument on the soundness and completeness of an Earley parser.

3.1 Soundness of an Earley parser

Given a context-free grammar, we use the notation w for finite sequences of terminal symbols, and the notation u, v for finite sequences of terminal and non-terminal symbols.

Moreover, given a non-terminal A, the language L_A is defined as the set of finite words of terminals will be the leaves of parse tree with the root A. Likewise, the language L_a of a terminal is defined as the singleton $\{a\}$. Additionally, $u.v$, means concatenation of u with v.

We can now introduce two important definitions.

Definition 1. *Given two sets L_1 and L_2 of finite sequences of terminal symbols,*

$$L_1 \cdot L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$$

for the set of finite words of terminal symbols obtained by concatenating a finite word $w_1 \in L_1$ with a finite word $w_2 \in L_2$.

Definition 2. *The language L_u associated to a finite sequence u of non-terminal and terminal symbols is defined by induction on the size of the finite sequence u.*

- $L_\varepsilon = \{\varepsilon\}$,
- $L_{a \cdot u} = L_a \cdot L_u$,
- $L_{A \cdot u} = L_A \cdot L_u$,

Note that for every pair (u, v) of finite sequences of terminal and non-terminal symbols,

$$L_{u \cdot v} = L_u \cdot L_v.$$

From these definitions a number of properties follow. One, is the following lemma:

Proposition 1. *For every rule $A \to w$ of the grammar,*

$$L_w \subseteq L_A$$

Proof of **Proposition 1** follows from the fact that w by definition is $a_0 \cdot a_1 \cdot \ldots a_n$. So from the definitions given, it holds that $L_w = L_{a_0 \cdot a_1 \cdot \ldots a_n}$. Furthermore, once again by definition, $L_{a_0 \cdot a_1 \cdot \ldots a_n} = L_{a_0} \cdot L_{a_1} \ldots L_{a_n}$. Recalling, that language of a terminal is singleton, thus we are concatenating singletons: $\{a_0\} \cdot \{a_1\} \cdot \ldots \{a_n\}$, which gives us the string: $a_0 a_1 \ldots a_n$. By definition this will be one of the branches of the language of L_A, thus $L_w \subseteq L_A$.

We have introduced enough definition to state the following theorem, which is equivalent to the soundness of an Earley parser.

Theorem 1 (Soundness)**.** *Suppose that the Early algorithm applied on the finite word of terminal symbols*

$$w \quad = \quad a_0 \cdots a_n$$

induces the item $[S' \to S\bullet, 0]$ *in the set* S_n*. In that case, the word* w *is an element of the language* L_S *generated by the context-free grammar.*

In order to prove the property, we show by induction on the number of steps performed by the Earley parser:

Proposition 2. *Suppose given a word*

$$w \quad = \quad a_0 \cdots a_n$$

and two elements $i, j \in \mathbb{N}$ *such that* $i < j \leq k$*. Suppose, moreover, that the item*

$$[A \to u \bullet v, i]$$

appears in the set S_j *during the parsing of the word* w*. In that case, the word* $a_i \cdots a_j$ *is an element of the language* L_u*.*

The proof behind this assertion follows from **Proposition 1** and recursively applying **Definition 2**. **Proposition** 1 let's us claim $L_u \subseteq L_A$. So the language L_u exists here. Furthermore, from **Definition 2** consider the fact that the premise permits us to assert, $L_{a_{j-1} \cdot u'} = L_{a_{j-1}} \cdot L_{u'}$, where u' is a_j. Now, by recursively using this definition, we simple get to the point where we can state $L_{a_i \cdot u'} = L_{a_i} \cdot L_{u'}$, where u' is $a_{i+1} \ldots a_j$. So clearly, $a_i \ldots a_j$ belong to L_u.

Moving forward, we can establish that each of the three rules of an Earley parser— SCANNER, PREDICTOR, and COMPLETER— corresponds to a step of the induction.

SCANNER

Define that in the Earley algorithm, given an input of $x_1, x_2, \ldots x_n$, we will have the original set S_0 and set S_i for every input x_i.

Assume that we have an Earley item in set S_i written as $[A \to \ldots a \bullet \ldots, j]$ where j is the pointer to where the production begins (the origin position) of the set S_j. This assumption is corresponding to the Proposition 2 of the Soundness Theorem which shows that the Scanner algorithm holds true for the loop invariant of the Soundness Theorem we are trying to prove.

We know that there is an intial set S_0 written as $[S' \rightarrow \bullet S, j]$ where the pointer is pointing to as the origin position.

Then, there has to exist an Earley item in set S_{i-1} written as
$[A \rightarrow \ldots \bullet a \ldots, j]$ such that $x_{i-1} = a$ so to create a sequence of set from the origin position that pointer j is pointing to.

PREDICTOR

Assume we have a sequence of terminals and non-terminals A written as $[A- > Xab]$ which can also be rewritten as $[L_A- > L_X]$ where L_A and L_X are the language of a non-terminal such that $L_X = \{ab|ab \in L_X\}$ (L_X contains the terminals ab).

Let sequence A be in the state $[L_A- > \bullet L_X]$. Since we denote that $L_X = \{ab|ab \in L_X\}$, we will thus evaluate the terminals that L_X contains which is ab.

The statement above also preserves the invariant of the Soundness Theorem which is that given the item $[L_A- > \bullet L_X]$, and the word x = a,b the word a,b is an element of the language L_x

COMPLETER

Assume we also have a sequence of terminals and non-terminals A written as $[A- > abXcd]$ which can also be rewritten as $[L_A- > abL_Xcd]$. L_A and L_X are the languages of a non-terminal.

Then, define the language of non-terminal as a set of finite terminals represented as follows: $L_A, L_B = \{ab|a \in L_A \& b \in L_B\}$

Using the definition above to apply to A, we have: $[L_A- > L_{Xa}L_{Xb}]$ such that L_{Xa} contains ab and L_{Xb} contains cd.

Assume we are at the state of $[L_A \rightarrow \bullet L_{Xa}L_{Xb},]$ which is the given state of Proposition 2 of the Soundness Theorem thus preserving the invariant of the induction proof. We then evaluate the first non-terminal L_{Xa} which contains ab as mentioned above. When the evaluation reaches the state where $[L_{Xa}- > ab\bullet]$. Following the sequence A consists of terminals and non-terminals as defined above, the Earley parser will now evaluate the next language of non-terminal L_{Xb} consists of terminals cd as it is in the state $[L_A- > L_{Xa}\bullet L_{Xb}]$.

3.2 Completeness of an Earley Parser

Regarding completeness, our argument is less rigorous and will be a sketch to what we believe is an argument that is correct in essence.

Assume that the set F is the collection of all the parse trees generated by running an Earley Parser. Furthermore, for sake of contradiction assume the Earley Parser is incomplete. Therefore, there would exist a parse tree T that is correct but not included in F. From our earlier work, we know that if T is correct than it each of its leaves, if concatenated will belong to L_S. In other words, the tree the $\forall T^{'} \in F, T^{'} \cap T$ will produce an intersection having at least one element, S.

Now, Earley Parser is a top-down parser. It works from the root, S. The PREDICTOR guesses possible parses in a set, S_i and the COMPLETER adds those that apply to the Earley Sets, S_j such that $i < j$, so based on previous works, a word w such that $w = a_i \dots a_j$ is an element of the parse tree with root A, where A is a nonterminal. Due to application of PREDICTOR and COMPLETER, only Earley Sets, S_j, having the Earley element, $[S^{'} -> S\bullet,0]$ will be valid parses. Thus, a parse tree, T, will be missed if PREDICTOR and COMPLETER are incorrect which would be a contradiction, since they are proven correct! Thus, the Earley Parser is complete.

3.3 Implementation

After our attempts to establish the soundness and completeness of an Earley Parser we used simple Python to prototype our primitive Earley parser [10]. The source code is hosted in a shared Github repository. Please keep in mind that we regard this as the first version of the Earley Parser and may over time change from what is presented in this paper.

4 Results and Discussion

4.1 Initial Plan and Current Implementation

Our original plan was to implement our Parser in Haskell for reasons cited in Section 2.5. However, this proved to be beyond our current technical scope. Thus, it was decided to quickly prototype our parser in Python; take advantage of its OOP features; and design a

parser that is a proof of concept.

Some of the more state-of-art treatment in the previous literature [7] [6] were neglected for simplification and feasibility.

Thus, the issue of ϵ-rules, improving upon established time and space efficiencies, supplementing rule-based parsing with probabilistic heuristics were not a point of consideration in the present implementation of our Earley Parser.

But the Earley Parser at present is correct and complete, a subject on which we invested the bulk of our effort.

4.2 Select Inputs and Outputs

(a) **Input=aaa**

Suppose we have a context-free grammar defined as such: $S- > SS|a$

Then, Earley Algorithm must generate a output that corresponds to the following:

Excepted Output	
Set Number	Earley Items
0	[S‘ → •S,0], [S→ •SS,0], [S→ •a,0]
1	[S‘ →S•,0], [S→S•S,0], [S→a•,0], [S→ •SS,1],[S→ •a,1]
2	[S‘ →SS•,0], [S→S•S,1], [S→a•,1], [S→ •SS,2],[S→ •a,2]
3	[S‘ →SS•,1], [S→S•S,2], [S→a•,2], [S‘→S•,0]

Table 2

```
PARSE FOR INPUT:aaa
S 0 : [S' ->.S , 0], [S ->.SS , 0], [S ->.Ta , 0],

S 1 : [S' ->S. , 0], [S ->S.S , 0], [S ->Ta. , 0], [S ->.SS , 1], [S ->.Ta , 1],

S 2 : [S ->SS. , 0], [S ->S.S , 1], [S ->Ta. , 1], [S ->.SS , 2], [S ->.Ta , 2],

S 3 : [S ->SS. , 1], [S ->S.S , 2], [S ->Ta. , 2], [S' ->S. , 0],
```

Figure 7: Output for The Grammar S->SS, S->a and Input aaa

As the reader can see from 7, our prototype parser generates the right output. Note, that the “T" in the output we get is just a character to mark “Terminal" characters and, therefore, does not represent any character in the nonterminal or terminal set.

(b) **Input=nn**

Suppose we have a context-free grammar defined as such: $S` -> E, E -> EE|n$

At this point our extensive description of Earley parsers and the previous example should give the reader a good basis to infer the theoretical output. As such, we will leave it to the reader to see that indeed the output shown in the figure below is correct.

```
PARSE FOR INPUT:aaa
S 0 : [S' ->.S , 0], [S ->.SS , 0], [S ->.Ta , 0],

S 1 : [S' ->S. , 0], [S ->S.S , 0], [S ->Ta. , 0], [S ->.SS , 1], [S ->.Ta , 1],

S 2 : [S ->SS. , 0], [S ->S.S , 1], [S ->Ta. , 1], [S ->.SS , 2], [S ->.Ta , 2],

S 3 : [S ->SS. , 1], [S ->S.S , 2], [S ->Ta. , 2], [S' ->S. , 0],
```

Figure 8: Output for The Grammar S`->nn, E->EE|n and Input=nn

(c) **Input= 我叫D (wojiaoD— I am D)**

Suppose we have a context-free grammar defined as such:

$S` -> S, S -> SPVN, P ->$ 我(wo) $, V ->$ 叫(jiao) $, N -> D$

Initializing the right grammar, our system successfully parses this simple sentence in Mandarin into the output that is shown below.

```
PARSE FOR INPUT: 我叫D
S 0 : [S' ->.S , 0], [S ->.PVN , 0], [P ->.T我 , 0],

S 1 : [S' ->.S , 0], [S ->P.VN , 0], [P ->T我. , 0], [S ->P.VN , 0], [V ->.T叫 , 1],

S 2 : [S' ->.S , 0], [S ->PV.N , 0], [V ->T叫. , 1], [S ->PV.N , 0], [N ->.TD , 2],

S 3 : [S' ->.S , 0], [S ->PVN. , 0], [N ->TD. , 2],
```

Figure 9: Earley Sets for A Simple Sentence In Mandarin

Thus, as the outputs above show, our parser is capable of parsing context-free grammar and generating the correct and complete Earley Sets. The information in the Earley codes encode the necessary information to recreate a parse tree with further processing.

4.3 Time Complexity

The literature clearly established that the worst-case time complexity of an Earley Parser is in the order of $O(n^3)$, where n is the length of the input string. Despite, the cubic

time-bound, Earley parser can actually attain linear runtime for a considerable subset of problems.

For example. for arithmetic language [6] having only the addition operation, our empirical study suggests that the parser will be bound by $O(n)$, where n is the size of the input.

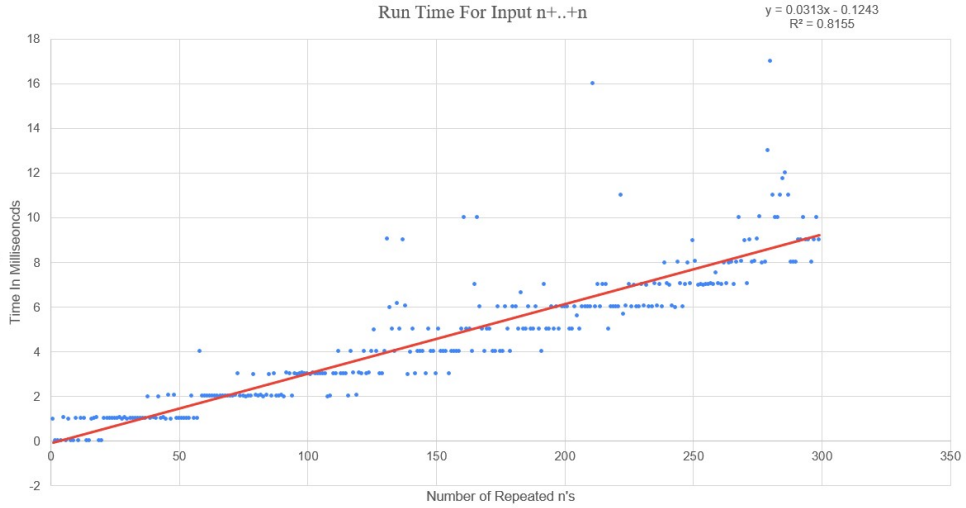


Figure 10: RunTime of Earley Parser For The Simple Input n+ . . . + n

10 depicts the input size going from 1 to 299 characters. The time taken for the code to run was measured using the Python library time, which is sufficiently accurate for this simple test. Importing the data on Excel and using the inbuilt tools therein we generated the scatter plot shown above. The linear trend line has a R^2 value of 0.8156, thus, the assuming that the time taken for our code to execute and the input size for this specific set of inputs is linear is a reasonable assumption.

This bodes well for our implementation. The fact that our code attains best-case run time is a tentative suggestion we used approrpriate data structures and good alogrithms to build our Earley Parser.

4.4 Discussion

Our project has ample room for growth. For a start, our Python code can be the departure point implementing our parser in Haskell, which was our original goal. Moreover, we can address some of the problems raised in the literature [6] [7] and incorporate more state-of-the-art treatment of Earley parsing, provide more robust time complexity analysis, both at a theoretical and empirical level, and in general strive for greater level of refinement.

5 Conclusion

Section 2 clearly establishes the historic context and significance of parsing and, in addition, presents a literature review of seminal papers and landmark results. Situating our work in this bigger and broader context, at present we have successfully recreated the foundational results of using context-free grammar to parse Mandarin.

We hope in the future to expand on these works to go above and beyond well established results. Section 4.4 already serves as a guideposts to areas where we can further refine our prototypes and find ourselves closer to the state-of-the-art rather than the historical.

6 Personal Contribution

While this project was a team work between Daisy Huynh and I, each of had our own individual contributions as well. Vast majority of written report was done by me; I added to the proof of Daisy had written, in particular the proof of the **Proposition 1** and **Proposition 2** and provided a sketch of why the Earley Parser is complete. In addition, I have written codes [10] but ultimately those were not used for the interest of time.

References

[1] K. Sikkel, *Parsing Schemata:A Framework for Specification and Analysis of Parsing Algorithms*, 1st ed., ser. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin, Heidelberg. [Online]. Available: https://doi.org/10.1007/978-3-642-60541-3

[2] D. Jurafsky and J. H. Martin, *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2009. [Online]. Available: http://www.amazon.com/Speech-Language-Processing-2nd-Edition/dp/0131873210/ref=pd_bxgy_b_img_y

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

[4] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950. [Online]. Available: http://www.jstor.org/stable/2251299

[5] D. E. Knuth, "On the translation of languages from left to right," *Information and control*, vol. 8, no. 6, pp. 607–639, 1965.

[6] J. Aycock and R. N. Horspool, "Practical Earley Parsing," *The Computer Journal*, vol. 45, no. 6, pp. 620–630, 01 2002. [Online]. Available: https://doi.org/10.1093/comjnl/45.6.620

[7] X. Xia and D. Wu, "Parsing chinese with an almost-context-free grammar," in *Conference on Empirical Methods in Natural Language Processing*, 1996.

[8] C. Allen, J. Moronuki, and S. Syrek, *Haskell Programming from First Principles*. Lorepub LLC, 2016. [Online]. Available: https://books.google.com/books?id=5FaXDAEACAAJ

[9] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[10] D. Huynh and R. Reza, "Capstone Project: Using Context-Free Grammar To Implement Mandarin," 12 2022. [Online]. Available: https://github.com/srr408/earleyParserProtoype.git