Formal Models of Language: Grammars

Paula Buttery

Easter 2019

This handout provides details of the grammars used throughout the course and associated methods for parsing.

0. Refresher of Discrete Maths

In the *Formal Languages and Automata* section of the *Discrete Maths* course we defined a FORMAL LANGUAGE as a set of STRINGS over an ALPHABET.

definition of a formal language

ALPHABETS An alphabet is specified by a ***finite*** set, Σ, whose elements are called symbols. Some examples are shown below:[1]

$\{0,1,2,3,4,5,6,7,8,9\}$ the 10-element set of decimal digits.

$\{a,b,c,...,x,y,z\}$ the 26-element set of lower case characters of written English.

$\{aardvark,...,zebra\}$ the 250,000-element set of *words* in the Oxford English Dictionary.[2]

[1] Note that e.g. the set of natural numbers $\mathbb{N} = \{0,1,2,3,...\}$ cannot be an alphabet because it is infinite.

STRINGS A string of length n over an alphabet Σ is an ordered n-tuple of elements of Σ. And Σ^* denotes the set of all strings over Σ of finite length.

If $\Sigma = \{a,b\}$ then ϵ, ba, bab, aab are examples of strings over Σ.

If $\Sigma = \{a\}$ then $\Sigma^* = \{\epsilon, a, aa, aaa, ...\}$

If $\Sigma = \{cats, dogs, eat\}$ then $\Sigma^* = \{\epsilon, cats, cats\ eat, cats\ eat\ dogs, ...\}$[3]

[2] Note that the term *alphabet* is overloaded: in formal language theory it refers to the set of elements that can combine to form *strings*: these symbols can be numbers, characters, English words or anything else as long as they are all of the same type! When discussing natural languages, *alphabet* refers exclusively to the set of characters used for orthographic representations (i.e. writing down the language's words); the term LEXICON is then used to refer to the set that contains the language's words as elements.

[3] The spaces here are for readable delimitation of the symbols of the alphabet.

LANGUAGES Given an alphabet Σ any subset of Σ^* is a FORMAL LANGUAGE over alphabet Σ.

In *Discrete Maths* you inductively defined subsets of Σ^* (languages) using axioms and rules. Below are some example axioms and rules for generating a language, $\mathcal{L}$, over the alphabet $\Sigma = \{a,b\}$, that contains strings of an a followed by zero or more b's, i.e. $\mathcal{L} = \{a, ab, abb, abbb, ...\}$.

inductively defined languages

AXIOMS Axioms specify elements of Σ^* that exist in $\mathcal{L}$.

$$\frac{}{a}\ (a1)$$

INDUCTION RULES Rules show ***hypotheses*** above the line and ***conclusions*** below the line (also referred to as ***children*** and ***parents*** respectively). The following is a unary rule where u indicates some string in Σ^*:

$$\frac{u}{ub}\ (\text{r1})$$

DERIVATIONS Given a set of axioms and rules for inductively defining a subset, $\mathcal{L}$, of Σ^*, a derivation of a string u in $\mathcal{L}$ is: a finite rooted tree with vertexes that are elements of $\mathcal{L}$ such that: the root of the tree (towards the bottom of the page) is u itself; each vertex of the tree is the conclusion of a rule whose hypotheses are its children; each leaf of the tree is an axiom. Using our axiom and rule, the derivation for the string abb is shown below:

$$\dfrac{\dfrac{\dfrac{}{a}\ (\text{a1})}{ab}\ (\text{r1})}{abb}\ (\text{r1})$$

1. Phrase Structure Grammars

Another way of generating a language is using a PHRASE STRUCTURE GRAMMAR. Also referred to as a GENERATIVE GRAMMAR, this formalism uses a set of REWRITE RULES or PRODUCTION RULES to *generate* strings in the language.[4] A phrase structure grammar over an alphabet Σ is defined by a tuple $G = (\mathcal{N}, \Sigma, S, \mathcal{P})$. We refer to the language generated by grammar G as $\mathcal{L}(G)$.

[4] This is opposed to constraint-based grammars which allow all strings in Σ^* that have not otherwise been constrained.

NON-TERMINALS $\mathcal{N}$: Non-terminal symbols may be ***rewritten*** using the rules of the grammar. We will denote the set of non-terminal symbols as $\mathcal{N}$ where $\mathcal{N} \cap \Sigma = \emptyset$. We generally use capital letters as non-terminals symbols.

definition of generative grammar

TERMINALS Σ: Terminal symbols are elements of Σ and cannot be rewritten. We generally use lower case symbols to refer to terminals.

START SYMBOL S: A distinguished non-terminal symbol $S \in \mathcal{N}$. This non-terminal provides the starting point for derivations.[5]

[5] S is sometimes referred to as the axiom but note that, whereas in the inductively defined sets above the axioms denoted the smallest members of the set that could be combined in some way to form new members, here the axioms denote the existence of particular derivable structures.

PHRASE STRUCTURE RULES $\mathcal{P}$: Phrase structure rules are pairs of the form (w, v) usually written:

$w \rightarrow v$, where $w \in (\Sigma \cup \mathcal{N})^* \mathcal{N} (\Sigma \cup \mathcal{N})^*$ and $v \in (\Sigma \cup \mathcal{N})^*$
That is, we have at least one non-terminal on the left mapping to anything on the right. The symbol(s) on the left-hand side of a rule is often referred to as the ***parent***; and any non-terminals on the right-hand side are called the ***daughters*** or ***children***.

A phrase structure grammar, G, that generates the language $\mathcal{L}(G) = \{a, ab, abb, abbb, ...\}$ is shown in Figure 1.

$$
\begin{aligned}
G = (\mathcal{N}, \Sigma, S, \mathcal{P}) \quad & \text{where} \\
\mathcal{N} &= \{S, A, B\} \\
\Sigma &= \{a, b\} \\
S &= S \\
\mathcal{P} &= \{S \rightarrow a, S \rightarrow aB, B \rightarrow bB, B \rightarrow b\}
\end{aligned}
$$

Figure 1: The grammar that generates $\mathcal{L}(G) = \{a, ab, abb, abbb, ...\}$. Note that it is possible to have more than one rule containing the starting symbol on the left-hand side.

In order to generate a string from a grammar we start with the designated starting symbol; then non-terminal symbols are repeatedly expanded using the rewrite rules until there is nothing further left to expand. The rewrite rules derive the members of a language from their internal structure (or phrase structure—hence the name phrase structure grammars). Figure 2 shows the derivation of the 3 smallest strings in $\mathcal{L}(G)$. We call such diagrams derivation trees (although note that the root of the tree is drawn at the top of the page).

Formally, given $G = (\mathcal{N}, \Sigma, S, \mathcal{P})$ and $w, v \in (\mathcal{N} \cup \Sigma)^*$ a derivation step is possible to transform w into v if:

definition of a derivation

$u_1, u_2 \in (\mathcal{N} \cup \Sigma)^*$ exist such that $w = u_1 \alpha u_2$, and $v = u_1 \beta u_2$

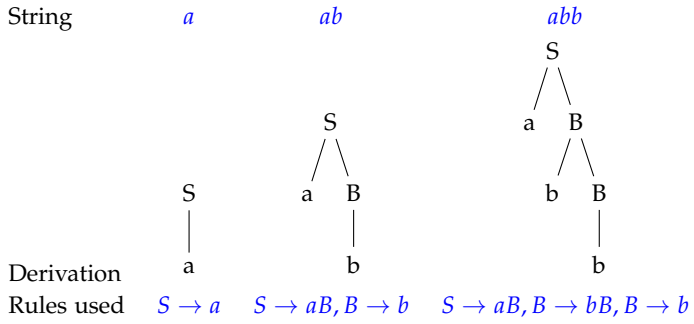


Figure 2: The derivation of the 3 smallest strings in $\mathcal{L}(G)$, derived from the grammar, G, in Figure 1.

and

$$\alpha \to \beta \in \mathcal{P}$$

this is written $w \underset{G}{\Rightarrow} v$

A string in the language $\mathcal{L}(G)$ is a member of Σ^* that can be derived in a finite number of derivation steps form the starting symbol S. We use $\underset{G^*}{\Rightarrow}$ to denote the reflexive, transitive closure of derivation steps, consequently $\mathcal{L}(G) = \{w \in \Sigma^* | S \underset{G^*}{\Rightarrow} w\}$.

equivalences of grammars

Two grammars are WEAKLY EQUIVALENT if they derive the same set of strings. They are STRONGLY EQUIVALENT if they derive the same set of strings with the same tree structures.

The Chomsky Hierarchy

Chomsky suggested that phrase structure grammars may be grouped together by the properties of their production rules. He specified 4 types of grammar referred to as the CHOMSKY HIERARCHY of grammars:

TYPE	NAME	FORM OF RULES
3	regular	($A \to Aa$ or $A \to aA$) and $A \to a$ for $A \in \mathcal{N}$ and $a \in \Sigma$
2	context free	$A \to \alpha$ where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$
1	context sensitive	$\alpha A \beta \to \alpha \gamma \beta$ where $A \in \mathcal{N}$ and $\alpha, \beta, \gamma \in (\mathcal{N} \cup \Sigma)^*$ and $\gamma \neq \epsilon$
0	recursively enumerable	$\alpha \to \beta$ where $\alpha, \beta \in (\mathcal{N} \cup \Sigma)^*$ and $\alpha \neq \epsilon$

Note that each rule-form in the hierarchy is a more specific version of the rule-form of the type below: hence every regular language is also a context free language etc.[6] In the following, I will refer to a CLASS of languages (e.g. the class of regular languages) as being all the languages that can be generated by a particular TYPE of grammar. The term POWER is used to describe the ***expressivity*** of each type of grammar in the hierarchy (measured in terms of the number of subsets of Σ^* that the type can generate—-that is, the size of the associated language class). Type 0 (recursively enumerable) grammars are the most ***powerful*** in that they are the most

[6] For those who enjoy the details, note that for CFGs, the RHS of a rule may be the empty string, but in CSGs this is not the case. So, strictly speaking, not every CFG is context-sensitive. We will not consider this point further here...see the lecturer if you would like some extra reading.

language classes

power/expressivity of language types

expressive (they can express the most subsets of Σ^*). Type 3 (regular grammars) are the least powerful (expressing the fewest subsets of Σ^*)

Since a class is a set of languages, we can reason about the closure properties of classes of languages.[7] For instance, we can prove that:[8]

closure properties of languages

[7] Recall from *Discrete Maths* that a set is CLOSED UNDER AN OPERATION if performing that operation on members of the set always produces another member of the same set. For instance, the set of positive integers are closed under addition, but are not closed under subtraction (because the operation of subtraction may result in a negative integer).

[8] proofs of these closures won't be examinable this year

- all Chomsky language classes are closed under the binary operation of union.

  $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) = \mathcal{L}(G_3)$ where G_1, G_2, G_3 are all grammars of the same type (e.g. the union of a context free language with another context free language will yield a context free language).

- all Chomsky languages classes are closed under intersection with a regular language.

  $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \mathcal{L}(G_3)$ where G_1 is a regular grammar and G_2, G_3 are grammars of the same type (e.g. the intersection of a regular language with a context free language will yield another context free language).

Further properties of the regular and context free language classes are outlined in the following sections:

Type 3: Regular Languages

You studied regular languages in *Discrete Maths*. Recall that a language is regular if it is equal to the set of strings accepted by some deterministic finite-state automaton (DFA).

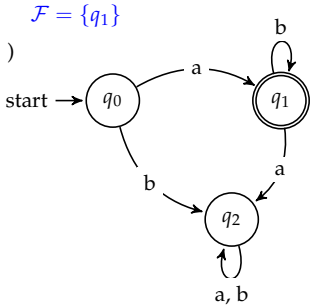
definition of a Deterministic Finite Automata

A DFA is defined as $M = (\mathcal{Q}, \Sigma, \Delta, s, \mathcal{F})$ where:

- $\mathcal{Q} = \{q_0, q_1, q_2 ...\}$ is a finite set of states.
- Σ is the alphabet: a finite set of the transition symbols.
- $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a function $\mathcal{Q} \times \Sigma \to \mathcal{Q}$ which we write as δ. Given $q \in \mathcal{Q}$ and $i \in \Sigma$ then $\delta(q, i)$ returns a new state $q' \in \mathcal{Q}$
- s is a starting state
- $\mathcal{F}$ is the set of all end states

Example: for $\mathcal{L}(M) = \{a, ab, abb, ...\}$ we require that M is defined as:

M=(
$\mathcal{Q} = \{q_0, q_1, q_2\}$,
$\Sigma = \{a, b\}$,
$\Delta = \{(q_0, a, q_1), (q_0, b, q_2), (q_1, a, q_2), (q_1, b, q_1), (q_2, a, q_2), (q_2, b, q_2)\}$,
$s = q_0$,
$\mathcal{F} = \{q_1\}$
)



Given a DFA $M = (\mathcal{Q}, \Sigma, \Delta, s, \mathcal{F})$ the language, $\mathcal{L}(M)$, of strings accepted by M can be generated by the regular grammar $G_{reg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ where:

regular language → regular grammar

- $\mathcal{N} = \mathcal{Q}$ the non-terminals are the states of M
- $\Sigma = \Sigma$ the terminals are the set of transition symbols of M
- $S = s$ the starting symbol is the starting state of M

- $\mathcal{P}$ = $q_i \rightarrow aq_j$ when $\delta(q_i, a) = q_j \in \Delta$
  or $q_i \rightarrow \epsilon$ when $q \in \mathcal{F}$ (i.e. when q is an end state)

We can also prove that regular grammars always generate regular languages by constructing a DFA whose set of accepted strings will be the same as the strings generated by the regular grammar.[9] Regular grammars come in two varieties: LEFT-LINEAR grammars, where the grammar rules are always of the form $A \rightarrow Aa \mid a$; and RIGHT-LINEAR grammars, where $A \rightarrow aA \mid a$ (for $A \in \mathcal{N}$ and $a \in \Sigma$).[10] Every left-linear grammar has a weakly equivalent right-linear grammar.

[9] Ask me or your supervisor if you would like to see a proof

[10] The rules here are written in a concise Backus-Naur form where $A \rightarrow Aa \mid a$ is shorthand for the two rules $A \rightarrow Aa$ and $A \rightarrow a$

pumping lemma for regular languages

The PUMPING LEMMA for regular languages is used to prove that a language is ***not*** regular. The pumping lemma property is:

All $w \in \mathcal{L}$ with $|w| \geq l$ can be expressed as a concatenation of three strings, $w = u_1 v u_2$, where u_1, v and u_2 satisfy:

- $|v| \geq 1$ (i.e. $v \neq \epsilon$)
- $|u_1 v| \leq l$
- for all $n \geq 0$, $u_1 v^n u_2 \in \mathcal{L}$ (i.e. $u_1 u_2 \in \mathcal{L}$, $u_1 v u_2 \in \mathcal{L}$, $u_1 v v u_2 \in \mathcal{L}$, $u_1 v v v u_2 \in \mathcal{L}$, etc.)

To use the Pumping Lemma to prove that a language $\mathcal{L}$ is not regular:[11] for each $l \geq 1$, find some $w \in \mathcal{L}$ of length $\geq l$ so that no matter how w is split into three, $w = u_1 v u_2$, with $|u_1 v| \leq l$ and $|v| \geq 1$, there is some $n \geq 0$ for which $u_1 v^n u_2$ is not in $\mathcal{L}$.

[11] For instance, to prove that $\mathcal{L} = \{a^n b^n | n \geq 0\}$ is not regular. For each $l \geq 1$, consider $w = a^l b^l \in \mathcal{L}$.

If $w = u_1 v u_2$ with $|u_1 v| \leq l$ & $|v| \geq 1$, then for some r and s:

- $u_1 = a^r$
- $v = a^s$, with $r + s \leq l$ and $s \geq 1$
- $u_2 = a^{l-r-s} b^l$

so $u_1 v^0 u_2 = a^r \epsilon a^{l-r-s} b^l = a^{l-s} b^l$

But $a^{l-s} b^l \notin \mathcal{L}$ so by the Pumping Lemma, $\mathcal{L}$ is not a regular language

Type 2: CONTEXT FREE LANGUAGES

CONTEXT FREE GRAMMARS (CFG) give rise to the class of context free languages. As described in the Chomsky Hierarchy table above, a CFG is defined by, $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$, where:

- $\mathcal{N}$ is a set of non-terminals
- Σ is a set of terminals
- S is the starting symbol $S \in \mathcal{N}$
- $\mathcal{P}$ is the set of grammar rules $P = \{A \rightarrow \alpha \mid A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*\}$

Note that despite the flexibility on the right hand side of the derivation rule, CFGs are most commonly written in something called CHOMSKY NORMAL FORM (CNF)[12]. That is, where every production rule has the form, $A \rightarrow BC$, or, $A \rightarrow a$, where, $A, B, C \in \mathcal{N}$, and, $a \in \Sigma$. For every CFG there is a weakly equivalent CNF alternative. For instance, the rule $A \rightarrow BCD$ may be rewritten as the two rules, $A \rightarrow BX$, and, $X \rightarrow CD$. Figure 3 shows the weakly equivalent trees which both have the symbols A, B, C in order at the leaf nodes.

[12] Chomsky normal form is a requirement of some parsing algorithms.

Chomsky normal form

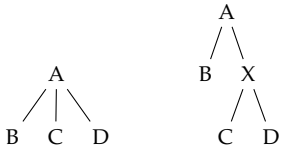
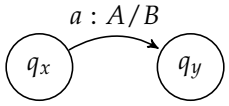


Figure 3: Partial derivations from 2 weakly equivalent context free grammars: the second grammar in Chomsky Normal Form

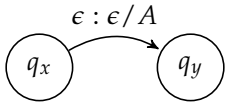
Whereas the set of strings accepted by a DFA was a ***regular language***, the set of strings accepted by a PUSH DOWN AUTOMATON (PDA) is a ***context free language***. A PDA is similar to a DFA except that is also has an associated STACK (last-in-first-out memory). The transition function for a PDA is based on the current state and transition symbol (as for a DFA) but also on the symbol currently at the top of the stack. On transitioning from one state to the next we may also ***push*** onto the stack or ***pop*** the stack. Diagrammatically we use the notation $a : A/B$ on the transitions (where $a \in \Sigma$ and $A, B \in \Gamma$) to indicate that we are encountering the transition symbol a and popping A from the top of the stack to replace it with B. A transition of the form $\epsilon : \epsilon/A$ corresponds to simply pushing A onto the stack; a transition of the form $\epsilon : A/\epsilon$ corresponds to popping A from the stack. The initial stack symbol is often written as z_0.

- in state q_x on encountering transition symbol a transition to state q_y popping A from the top of the stack and pushing B onto the stack



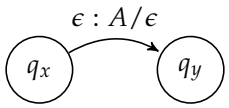
BEFORE	AFTER
A	B
z_0	z_0

- in state q_x transition to state q_y pushing A onto the stack



BEFORE	AFTER
z_0	A
	z_0

- in state q_x transition to state q_y popping A from the stack



BEFORE	AFTER
A	z_0
z_0	

A PDA is defined as $M = (\mathcal{Q}, \Sigma, \Gamma, \Delta, s, \bot, \mathcal{F})$ where:

- $\mathcal{Q} = \{q_0, q_1, q_2 ...\}$ is a finite set of states.
- Σ is the input alphabet: a finite set of the transition symbols.
- Γ is the stack alphabet: a finite symbols that may be pushed and popped from the stack.
- $\Delta \subseteq (\mathcal{Q} \times (\Sigma \cup \epsilon) \times \Gamma) \times (\mathcal{Q} \times \Gamma^*)$ is a relation $(\mathcal{Q} \times (\Sigma \cup \epsilon) \times \Gamma) \rightarrow (\mathcal{Q} \times \Gamma^*)$ which we write as δ. Given $q \in \mathcal{Q}$, $i \in \Sigma$ and $A \in \Gamma$ then $\delta(q, i, A)$ returns (q', α), that is, a new state $q' \in \mathcal{Q}$ and replaces A at the top of the stack with $\alpha \in \Gamma^*$
- s is the starting state

Example:

$\mathcal{L}(G_{cfg}) = \{ab, aabb, aaabbb, ...\}$

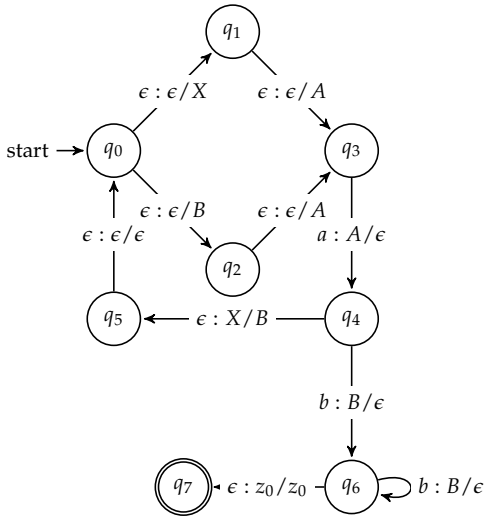
that is $\mathcal{L}(G_{cfg}) = \{a^n b^n\}$

$G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ where

$$\begin{aligned} \mathcal{N} &= \{S, A, B, X\} \\ \Sigma &= \{a, b\} \\ S &= S \\ \mathcal{P} &= \{S \rightarrow AX, \\ &\quad S \rightarrow AB, \\ &\quad X \rightarrow SB, \\ &\quad A \rightarrow a, \\ &\quad B \rightarrow b\} \end{aligned}$$

$M_{cfg} = (\mathcal{Q}, \Sigma, \Gamma, \Delta, s, \bot, \mathcal{F})$ where

$$\begin{aligned} \mathcal{Q} &= \{q_0, q_1, q_2\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{S, X, A, B\} \\ \Delta &= \{(q_0, \epsilon, \epsilon) \rightarrow (q_1, X), ...\} \text{ as diagram} \\ s &= q_0 \\ \bot &= z_0 \\ \mathcal{F} &= \{q_1\} \end{aligned}$$



definition of a Push Down Automaton

- $\perp$ is the initial stack symbol
- $\mathcal{F}$ is the set of all end states

The pumping lemma for context free languages (CFLs) is used to show that a language is not context free. The pumping lemma property for CFLs is:

pumping lemma for context free languages

All $w \in \mathcal{L}$ with $|w| \geq k$ can be expressed as a concatenation of five strings, $w = u_1 y u_2 z u_3$, where u_1, y, u_2, z and u_3 satisfy:

- $|yz| \geq 1$ (i.e. we cannot have $y = \epsilon$ and $z = \epsilon$)
- $|yu_2z| \leq k$
- for all $n \geq 0$, $u_1 y^n u_2 z^n u_3 \in \mathcal{L}$ (i.e. $u_1 u_2 u_3 \in \mathcal{L}$, $u_1 y u_2 z u_3 \in \mathcal{L}$, $u_1 y y u_2 z z u_3 \in \mathcal{L}$ etc.)

That is, every sufficiently long string in the language can be subdivided into five segments such that the middle three segments are less than some k, the second and fourth are not both null, and no matter how many times the second and fourth elements are simultaneously pumped, the resulting string is in the language .

To use the pumping lemma to prove that a language $\mathcal{L}$ is not context free for each $k \geq 1$, find some $w \in \mathcal{L}$ of length $\geq k$ so that no matter how w is split into five, $w = u_1 y u_2 z u_3$, with $|yu_2z| \leq k$ and $|v| \geq 1$, there is some $n \geq 0$ for which $u_1 y^n u_2 z^n u_3$ is not in $\mathcal{L}$.

Type 1: CONTEXT SENSITIVE LANGUAGES *and Type 0:* RECURSIVELY ENUMERABLE LANGUAGES

To complete the story, it is worth knowing that context sensitive languages are those that are recognised by a LINEAR BOUNDED AUTOMATON (LBA). This type of machine is similar to the PDA described above except that, instead of a stack, there is a ***tape*** of finite length on which symbols from $\Sigma*$ may be written (i.e. the automaton is not restricted to a last-in-first-out memory but may read/write on the tape by moving left, right or staying stationary).

Linear Bounded Automata and Turing Machines

The recursively enumerable languages are those that are recognised by a TURING MACHINE,[13] which is like an LBA without restriction on the tape length.

[13] You will cover Turing machines in your Complexity Theory lectures so I will not reproduce the definition here—it is not examinable in this course

Parsing Phrase Structure Grammars

When we say we PARSE a string of a language we are referring to one of two possible tasks:

1 The first meaning of *parse* refers to the task of deciding whether some string belongs to a language (ie. for a given input string, discovering whether there is a sequence of steps that start in the starting state and end in an accepting state of an associated automaton). This is referred to as the RECOGNITION PROBLEM. We can define the COMPLEXITY of a language class by finding the length of the ***longest accepting computation*** (or the longest ***running time***) of M_{class} on an input of length n.[14]

running time and class complexity

[14] For instance, the recognition problem for a string of length n in a regular language is $O(n)$. Informally, consider a DFA M_{reg}: beginning in the start state, for each input symbol i encountered $\delta(q, i)$ is computed and a next state q' returned. After all the input symbols have been encountered we can return a 1 if the final state is an accepting state (that is if $q' \in \mathcal{F}$). The maximum number of steps is at most linear with the length of the input string as one computation of δ is required per symbol encountered. This is a worst case analysis: for instance, the language of strings that start with a could be recognised in constant time.

   The complexity of each of the language classes in the Chomsky hierarchy is show below.

TYPE	LANGUAGE CLASS	COMPLEXITY
3	regular	$O(n)$
2	context free	$O(n^c)$
1	context sensitive	$O(c^n)$
0	recursively enumerable	*undecidable*

2 The second meaning of *parse* refers to the task of deriving the "correct" underlying structure of a string that is already known to be in a language. This second meaning is what we refer to when we parse sentences of natural language for computer applications. Parsing is usually the first stage in a pipeline of processes and the "correct" parse here would mean the one that represents the most useful[15] phrase structure of the sentence (also referred to as ***constituent structure***) given the words in the sentence.

parse structure

[15] As determined by the application.

For reasons of compromise between efficiency and expressivity it is desirable to model natural language using CFGs —although arguably CFGs are not expressive enough to capture the phrase structure of all natural language (see the ***Formal vs. Natural Language*** handout).

The ***LR(k) shift-reduce parsers*** you have studied in *Compiler Construction* are most useful for recognising the strings of ***deterministic*** languages (that is, languages where no string has more than one analysis) which have been described by an unambiguous grammar. You have covered LR(k) shift-reduce parsers in detail in *Compiler Construction* so I will not reproduce the formal definitions, but, as an *aide-memoire*:

- The LR parsing algorithm has two actions: SHIFT and REDUCE
- Initially the input string is held in the buffer and the stack is empty.

- Symbols are ***shifted*** from the buffer to the stack
- When the top items of the stack match the RHS of a rule in the grammar then they are ***reduced***, that is, they are replaced with the LHS of that rule.

Natural languages (and the phrase structure grammars used to model them) are inherently ambiguous and is not well suited to LR parsers which operate deterministically recognising a single derivation without backtracking. The following describes the Earley parser: this is a parser that uses a top-down approach to explore the whole search space, recovering multiple derivations where they exist. It uses a dynamic programming algorithm that records partial derivations in a CHART (a table) in order to ***parse*** strings. By convention, each row in an Earley chart is referred to as an EDGE. The algorithm can be used to both ***recognise*** whether a string is in a language, and also ***derive the structure(s)*** of the string (by means of recording the steps taken during recognition).

Earley parser

Given an input grammar $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ and input string u the algorithm works top-down left-to-right exploring the space of all possible derivation trees. The progress of the algorithm is encoded in something called a DOTTED RULE or PROGRESS RULE: a rule of the form $A \rightarrow {}_{\bullet}\alpha\beta \mid \alpha_{\bullet}\beta \mid \alpha\beta_{\bullet}$ where $A \rightarrow \alpha\beta \in \mathcal{P}$.[16] The position of the dot in the dotted rule indicates progress. That is, it indicates which symbols have already been *used up* deriving a portion of the input string, and which symbols are left to be explored. Rules of the form $A \rightarrow {}_{\bullet}\alpha\beta$ have all symbols still to be explored; rules of the form $A \rightarrow \alpha\beta_{\bullet}$ have been completely *used up* deriving a portion of the string.[17]

[16] You will be familiar with this idea from the shift-reduce parsers introduced in the *Compiler Construction* course

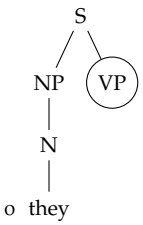
An edge in the chart (dynamic programming table) records a dotted rule, and its SPAN. The span refers to the portion of the input string which is consistent with the partial tree. The edge $A \rightarrow \alpha_{\bullet}\beta\ [i, j]$ is recorded in the chart if it is consistent with the input symbols up to i and spans the input up to j. More formally, for input string $u = a_1 ... a_n$ and grammar $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$, an edge $A \rightarrow \alpha_{\bullet}\beta\ [i, j]$ is added if:

- $S \underset{G^*}{\Longrightarrow} a_1 ... a_i A\gamma$ where γ are symbols in u yet to be parsed
- and $\alpha \underset{G^*}{\Longrightarrow} a_{i+1} ... a_j$

If we wish to discover the structure of a parse, an edge must also contain a record of the immediately previous partial tree(s) that made the current partial tree possible. This is accomplished by giving each edge a unique id and recording their contribution to the current edge in the ***history***.

The chart is initialised with the edge $S \rightarrow {}_{\bullet}\alpha\beta\ [0, 0]$; and the input string $u = a_1 ... a_n$ is recognised when we add the edge $S \rightarrow \alpha\beta_{\bullet}\ [0, n]$.[18]

[17] For an illustration of a dotted rule in use, consider the partial tree below which has been derived when attempting to parse the sentence *they can fish*:



In this example, the algorithm is yet to explore the VP node of the derivation tree but the NP node has been fully explored. This would be recorded in the chart as $S \rightarrow NP_{\bullet}VP$. The partial tree that has been derived is consistent with the input string up to the end of the word *they*—it ***spans*** from position 0 to 1—this span is also recorded in the chart. The chart edge associated with this example might look like the following:

ID	RULE	[start, end]	HIST
⋮			
e_i	$S \rightarrow NP_{\bullet}VP$	$[0, 1]$	h_k

[18] I am assuming only one starting symbol. If there is more than one you can add the rule $S_1 \rightarrow {}_{\bullet}S$ for a new starting symbol S_1.

The algorithm is detailed formally below with a helper example from a toy grammar. Figure 4 shows the grammar rules for our toy grammar and the input sentence we are trying to parse with its numbered locations.

- INITIALISE THE CHART: The chart is initialised with $S \rightarrow {}_{\bullet}\alpha\beta\ [0,0]$.

  In rule induction notation this can be considered to be an axiom:

$$\frac{}{S \rightarrow {}_{\bullet}\alpha\beta\ [0,0]}\ \text{(induction step)}$$

- MAIN BODY OF ALGORITHM: For each word in the sentence we proceed through the 3 steps :

  Prediction This step adds new edges to the chart and can be thought of as expanding tree nodes in the top-down derivation. A non-terminal (tree node) is unexplored if it occurs in any previous edges with a dot on its LHS. So for all edges in the chart, find any non-terminals with a dot on their LHS and expand them according to the rule set. They will appear with a span of [n,n] where n is the end location of the edge that is being expanded. Note that, an edge should only be added if it does not already appear in the chart (this will guarantee termination).

  In rule induction notation we have the following rule:

$$\frac{A \rightarrow \alpha_{\bullet} B\beta\ [i,j]}{B \rightarrow {}_{\bullet}\gamma\ [j,j]}\ \text{(predict step) where } B \rightarrow \gamma \in \mathcal{P}$$

  Scan This step allows us to check if we have a node that is consistent with the input sentence. If the input sentence is $u = a_1 ... a_n$ we can add a new edge if $A \rightarrow {}_{\bullet}a\ [i, j-1]$ and $a = a_j$.

  In rule induction notation we have the following rule:

$$\frac{A \rightarrow {}_{\bullet}a\ [i,j-1]}{A \rightarrow a_{\bullet}\ [i,j]}\ \text{(scan step) when } a = a_j$$

  For natural language sentence parsing tasks, Σ can be the finite set of words in the language (a very large set). As such, when carrying out the predicting step from a rule like $NP \rightarrow {}_{\bullet}\ N$ we would end up adding a new edge for every *noun* in the language. To save us from creating all these edges we can privilege a set of the non-terminals and perform a forward look-up of the next aj to see whether it will be consistent. In our example this set would be $\mathcal{N}_{PofS} = \{N, V, P\}$, that is, all the non-terminal symbols that represent the ***parts of speech*** of the language (things like *nouns*, *verbs*, *adjectives*...). So, during the scanning step, we find edges containing non-terminals in $\mathcal{N}_{PofS}$ with a dot on their LHS and check if the upcoming word is consistent with the part-of-speech. Iff it is consistent then we add an edge to the chart.

We wish to parse the sentence *they can fish* using $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ where:

$$\begin{aligned}
\mathcal{N} &= \{S, NP, VP, PP, N, V, P\} \\
\Sigma &= \{can, fish, in, rivers, they...\} \\
S &= S \\
\mathcal{P} &= \{S \rightarrow NP\ VP \\
&\quad NP \rightarrow N\ PP \mid N \\
&\quad PP \rightarrow P\ NP \\
&\quad VP \rightarrow VP\ PP \mid V\ VP \mid V\ NP \mid V \\
&\quad N \rightarrow can \mid fish \mid rivers \mid ... \\
&\quad P \rightarrow in \mid ... \\
&\quad V \rightarrow can \mid fish \mid ...\ \}
\end{aligned}$$

0 they 1 can 2 fish 3

Figure 4: A toy grammar and sentence to parse

ID	RULE	[start, end]	HIST
e_0	$S \rightarrow \bullet\ NP\ VP$	[0,0]	

Figure 5: chart for toy example after the *initialisation*

ID	RULE	[start, end]	HIST
e_0	$S \rightarrow \bullet\ NP\ VP$	[0,0]	
e_1	$NP \rightarrow \bullet\ N$	[0,0]	
e_2	$NP \rightarrow \bullet\ N\ PP$	[0,0]	

Figure 6: chart for toy example after the *predict* step

ID	RULE	[start, end]	HIST
e_0	$S \rightarrow \bullet\ NP\ VP$	[0,0]	
e_1	$NP \rightarrow \bullet\ N$	[0,0]	
e_2	$NP \rightarrow \bullet\ N\ PP$	[0,0]	
e_3	$N \rightarrow they\ \bullet$	[0,1]	

Figure 7: chart for toy example after the *scan* step

Complete This step propagates fully explored tree nodes in the chart. A node has been fully explored when it appears in a dotted rule with a dot at the very RHS. The non-terminal on the LHS of this rule (before the arrow) is the node that has been fully explored. So, on finding an edge that contains a rightmost dotted rule, propagate the dots in all edges that were *waiting* for this node to complete: do so by adding a new edge (rather than overwriting the old edge) so that the tree structure can be retrieved later—the IDs of the edges involved are added in a list in the history column of the chart. The history can be used on parse completion to understand how the tree structure was derived. Note that if any new rightmost dotted rules are created by this step they should themselves also be propagated.

In rule induction notation we have the following rule:

$$\frac{A \to \alpha \bullet B\beta \ [i,k] \qquad B \to \gamma \bullet \ [k,j]}{A \to \alpha B \bullet \beta \ [i,j]} \text{ (complete step)}$$

The running time of the Earley parser is $O(n^3)$. There are various implementation optimisations that can be performed but all reduce the constant factor rather than the exponent of the polynomial.

The following shows the full chart obtained when parsing the sentence, *they can fish* with the G_{cfg} in Figure 9. The final column is for your reference only and indicates which word we are processing when the edges are added.[19]

ID	RULE	[start, end]	HIST
e_0	$S \to \bullet\ NP\ VP$	$[0,0]$	
e_1	$NP \to \bullet\ N$	$[0,0]$	
e_2	$NP \to \bullet\ N\ PP$	$[0,0]$	
e_3	$N \to they\ \bullet$	$[0,1]$	
e_4	$NP \to N\ \bullet$	$[0,1]$	e_3
e_5	$NP \to N\ \bullet\ PP$	$[0,1]$	e_3
e_6	$S \to NP\ \bullet\ VP$	$[0,1]$	e_4

Figure 8: chart for toy example after the *complete* step. Note that only completed edges have a history (edges derived from prediction and scanning steps do not). Some implementations explicitly note the difference between the two types of edge by referring to the completed edges as the ***table/chart*** and the other edges as the ***agenda***.

[19] Note that edge e_{34} exhibits an instance of ***ambiguity packing***, i.e when analyses of the same type that are covering the same portion of the input string are *packed* into a single entity; and e_{27} and e_{28}, for instance, demonstrate sub-tree sharing.

ID	RULE	[start, end]	HIST	**word n**
e_0	$S \rightarrow \bullet\ NP\ VP$	[0, 0]		**word 0**
e_1	$NP \rightarrow \bullet\ N$	[0, 0]		**word 1**
e_2	$NP \rightarrow \bullet\ N\ PP$	[0, 0]		
e_3	$N \rightarrow they\ \bullet$	[0, 1]		
e_4	$NP \rightarrow N\ \bullet$	[0, 1]	(e_3)	
e_5	$NP \rightarrow N\ \bullet\ PP$	[0, 1]	(e_3)	
e_6	$S \rightarrow NP\ \bullet\ VP$	[0, 1]	(e_4)	
e_7	$PP \rightarrow \bullet\ P\ NP$	[1, 1]		**word 2**
e_8	$VP \rightarrow \bullet\ V$	[1, 1]		
e_9	$VP \rightarrow \bullet\ V\ NP$	[1, 1]		
e_{10}	$VP \rightarrow \bullet\ V\ VP$	[1, 1]		
e_{11}	$VP \rightarrow \bullet\ VP\ PP$	[1, 1]		
e_{12}	$V \rightarrow can\ \bullet$	[1, 2]		
e_{13}	$VP \rightarrow V\ \bullet$	[1, 2]	(e_{12})	
e_{14}	$VP \rightarrow V\ \bullet\ NP$	[1, 2]	(e_{12})	
e_{15}	$VP \rightarrow V\ \bullet\ VP$	[1, 2]	(e_{12})	
e_{16}	$S \rightarrow NP\ VP\ \bullet$	[0, 2]	(e_4,e_{13})	
e_{17}	$VP \rightarrow VP\ \bullet\ PP$	[1, 2]	(e_{13})	
e_{18}	$NP \rightarrow \bullet\ N$	[2, 2]		**word 3**
e_{19}	$NP \rightarrow \bullet\ N\ PP$	[2, 2]		
e_{20}	$VP \rightarrow \bullet\ V$	[2, 2]		
e_{21}	$VP \rightarrow \bullet\ V\ NP$	[2, 2]		
e_{22}	$VP \rightarrow \bullet\ V\ VP$	[2, 2]		
e_{23}	$VP \rightarrow \bullet\ VP\ PP$	[2, 2]		
e_{24}	$PP \rightarrow \bullet\ P\ NP$	[2, 2]		
e_{25}	$N \rightarrow fish\ \bullet$	[2, 3]		
e_{26}	$V \rightarrow fish\ \bullet$	[2, 3]		
e_{27}	$NP \rightarrow N\ \bullet$	[2, 3]	(e_{25})	
e_{28}	$NP \rightarrow N\ \bullet\ PP$	[2, 3]	(e_{25})	
e_{29}	$VP \rightarrow V\ \bullet$	[2, 3]	(e_{26})	
e_{30}	$VP \rightarrow V\ \bullet\ NP$	[2, 3]	(e_{26})	
e_{31}	$VP \rightarrow V\ \bullet\ VP$	[2, 3]	(e_{26})	
e_{32}	$VP \rightarrow V\ NP\ \bullet$	[1, 3]	(e_{12},e_{27})	
e_{33}	$VP \rightarrow V\ VP\ \bullet$	[1, 3]	(e_{12},e_{29})	
e_{34}	$S \rightarrow NP\ VP\ \bullet$	[0, 3]	(e_4,e_{32}) or (e_4,e_{33})	

Parsing the sentence *they can fish* using $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ where:

$$
\begin{aligned}
\mathcal{N} &= \{S, NP, VP, PP, N, V, P\} \\
\Sigma &= \{can, fish, in, rivers, they...\} \\
S &= S \\
\mathcal{P} &= \{S \rightarrow NP\ VP \\
&\quad NP \rightarrow N\ PP \mid N \\
&\quad PP \rightarrow P\ NP \\
&\quad VP \rightarrow VP\ PP \mid V\ VP \mid V\ NP \mid V \\
&\quad N \rightarrow can \mid fish \mid rivers \mid ... \\
&\quad P \rightarrow in \mid ... \\
&\quad V \rightarrow can \mid fish \mid ...\ \}
\end{aligned}
$$

also

$$\mathcal{N}_{PofS} = \{N, V, P\}$$

0 they 1 can 2 fish 3

Figure 9: Above is a repeat of Figure 4 for reference—a toy grammar with sentence to parse

2. Dependency Grammars

dependency grammars

A DEPENDENCY TREE is a directed graph representation of a string where the only nodes are the symbols in the string and each edge represents a syntactic relationship between the symbols. A DEPENDENCY GRAMMAR is a grammar that derives dependency trees. Formally $G_{dep} = (\Sigma, \mathcal{D}, s, \bot, \mathcal{P})$ where:

- Σ is the finite set of alphabet symbols
- $\mathcal{D}$ is the set of symbols to indicate whether the dependent symbol (the one on the RHS of the rule) will be located on the left or right of the current item within the string $\mathcal{D} = \{\mathcal{L}, \mathcal{R}\}$
- s is the root symbol for the dependency tree (we will use $s \in \Sigma$ but sometimes a special extra symbol is used)
- $\bot$ is a symbol to indicate a halt in the generation process
- $\mathcal{P}$ is a set of rules for generating dependencies: $\mathcal{P} = \{(\alpha \to \beta, d) \mid \alpha \in (\Sigma \cup s),\ \beta \in (\Sigma \cup \bot),\ d \in \mathcal{D}\}$

Diagrammatic representations of a dependency tree for the string *bacdfe* generated using $G_{dep} = (\Sigma, \mathcal{D}, s, \bot, \mathcal{P})$ where:

$$
\begin{aligned}
\Sigma &= \{a...f\} \\
D &= \{\mathcal{L}, \mathcal{R}\} \\
s &= \text{a} \\
\mathcal{P} &= \{(a \to b, \mathcal{L} \mid c, \mathcal{R} \mid d, \mathcal{R}) \\
&\quad (d \to e, \mathcal{R}) \\
&\quad (e \to f, \mathcal{L}) \\
&\quad (b \to \bot, \mathcal{L} \mid \bot, \mathcal{R}) \\
&\quad (c \to \bot, \mathcal{L} \mid \bot, \mathcal{R}) \\
&\quad (f \to \bot, \mathcal{L} \mid \bot, \mathcal{R})\}
\end{aligned}
$$

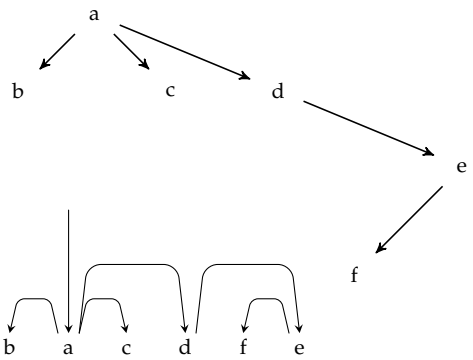


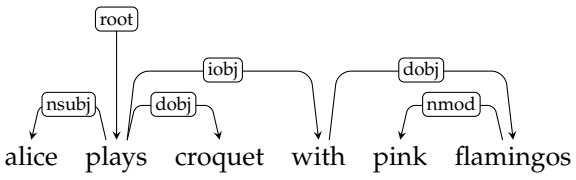
Figure 10: Above, two different style dependency tree representations for the string *bacdfe*. Notice that the same rules would have been used to generate the string *badfec* making dependency grammars useful when there is some degree of flexibility in the symbol order of grammatical strings in $\mathcal{L}(G_{dep})$

In dependency grammars we refer to the term on the LHS of a rule as the HEAD and the RHS as the DEPENDENT (as opposed to *parents* and *children* in phrase structure grammars). An example dependency tree is shown in Figure 10. A valid derivation tree is one that is ***rooted*** in s and ***weakly connected*** (that is, when considered as an undirected graph, the symbols in the string are connected). Dependency trees can be described as being PROJECTIVE or NON-PROJECTIVE—loosely speaking, a projective tree can be drawn without edges crossing whereas a non-projective tree cannot. The difference has implications for parsing complexity.

With a small modification to the rules, a label can be added to each generated dependency $\mathcal{P} = \{(\alpha \to \beta : r, d) \mid \alpha \in (\Sigma \cup s),\ \beta \in (\Sigma \cup \bot),\ d \in \mathcal{D},\ r \in \mathcal{B}\}$ where $\mathcal{B}$ is the set of dependency labels. When used for natural language parsing, dependency grammars will often label each dependency with the *grammatical function* (or the ***grammatical relation***) between the words.

grammatical relations

For illustration, consider the sentence *Alice plays croquet with pink flamingos*; a phrase structure constituency tree for this sentence is shown in Firgure 11. A labelled dependency tree (in both formats) for the same sentence is show below:[20]



[20] The relation labels used here follow Carroll et al., 1999. There are other labelling conventions, most notably the Universal Dependencies of Nivre et al., 2016. Here *nsubj* is a subject relation; *dobj*, *iobj* are direct and indirect object relations; *nmod* is a noun modifier. You will learn more about linguistic dependencies in *Natural Language Processing*.

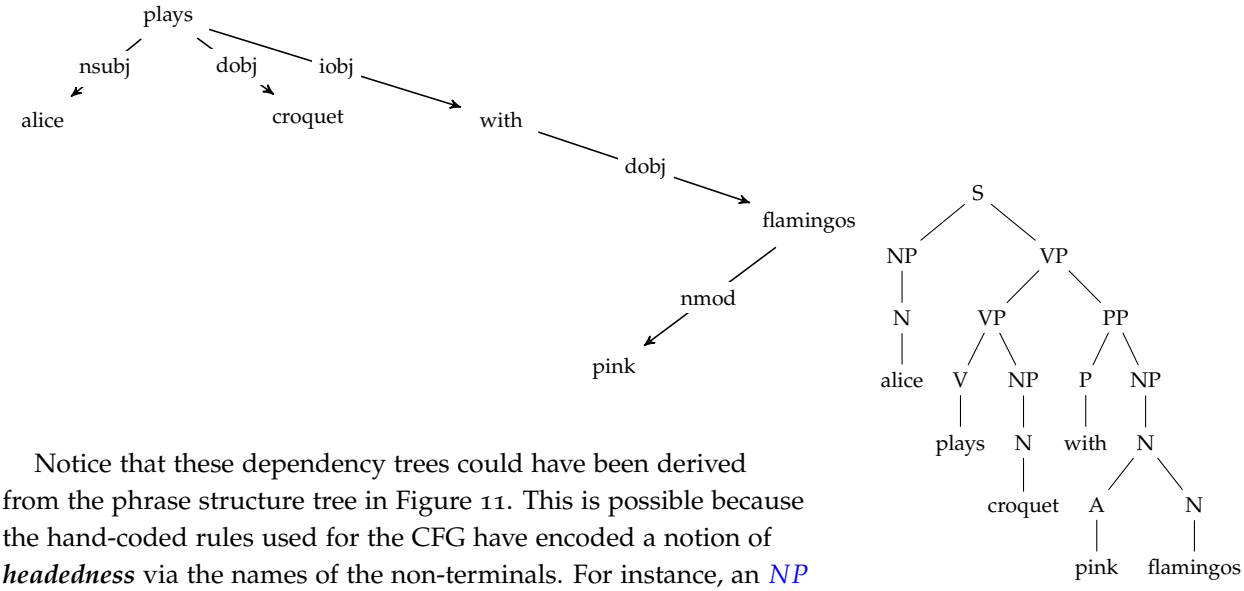
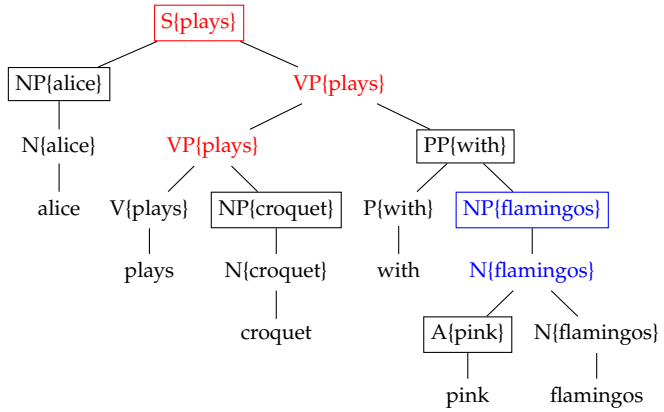


Figure 11: Above, the sentence *Alice plays croquet with pink flamingos* derived from some CFG.

Notice that these dependency trees could have been derived from the phrase structure tree in Figure 11. This is possible because the hand-coded rules used for the CFG have encoded a notion of ***headedness*** via the names of the non-terminals. For instance, an *NP* is a *noun phrase*, its *head* is a *noun*. By convention the head of an *S* node is always the verb from the *VP*. Below is the phrase structure tree from Figure 11 with the head symbols passed up the tree.



dependency grammar vs. CFG

By highlighting nodes that show the highest position of any given head (indicated by the boxes) we reveal a tree strikingly similar to our dependency tree. Since dependency grammars do not encode hierarchical constituency, collapsing paths but retaining connectedness will allow us to obtain the required dependency tree (for instance the nodes highlighted in *red* or *blue* can be collapsed with connectedness between heads remaining the same). From this example you should be able to see that we can make a systematic mapping from a phrase-structure tree to a dependency tree (by assuming a notion of headedness which passes up from one of the children to the parent). Following this insight, it is possible to prove that projective dependency grammars are weakly equivalent to context-free grammars.[21]

[21] The recognition problem for dependency grammars exhibiting unconstrained non-projective structures turns out to be $\mathcal{NP}$-complete (although this matters less when using data driven methods as outlined below).

Parsing with Dependancy Grammars

shift-reduce dependency parsing

The most commonly used methods for dependency parsing for natural language involve a modification of the LR shift-reduce parser for phrase structure grammars. The ***shift*** operator continues to move items of the input string from the buffer to the stack but the ***reduce*** operator is replaced with the operations LEFT-ARC and RIGHT-ARC which *reduce* the top two stack symbols leaving the *head* on the stack (that is, the *rightmost* or *leftmost* respectively). The dependency rule that has been used to make the reduction is recorded for recovery of the dependency tree after recognition of the input string.

For illustration, consider $\mathcal{L}(G_{dep}) \subseteq \Sigma^*$; during parsing the stack may hold γab where $\gamma \in \Sigma*$ and $a, b \in \Sigma$, and b is at the top of the stack:

- LEFT-ARC reduces the stack to γb and records use of rule $b \rightarrow a$
- RIGHT-ARC reduces the stack to γa and records the use of rule $a \rightarrow b$

For natural language there would be considerable effort in manually defining $\mathcal{P}$—this would involve determining the dependencies between all possible words in the language; and creating a deterministic language is impossible (since natural language is inherently ambiguous). However, natural language parsing is achieved deterministically by selecting parsing actions using a machine learning classifier. The features for the classifier include the items on the stack and in the buffer at a given point as well as properties of those items (including word-embeddings for the items). Training is performed on ***dependency banks*** (that is, sentences that have been manually annotated with their correct dependencies). It is said that the parsing is grammarless—since no grammar is designed ahead of training.

To avoid the problem of early incorrect resolution of an ambiguous parse, multiple competing parses can be recorded and a beam search used to keep track of the best alternative parses.[22]

Example of shift-reduce parse for the string $bacdfe$ generated using $G_{dep} = (\Sigma, \mathcal{D}, s, \perp, \mathcal{P})$ (note, we can ignore $\alpha \rightarrow \perp$ rules here because we are parsing rather than generating):

$$
\begin{array}{lcl}
\Sigma & = & \{a...f\} \\
D & = & \{\mathcal{L}, \mathcal{R}\} \\
s & = & \text{a} \\
\mathcal{P} & = & \{(a \rightarrow b, \mathcal{L} \mid c, \mathcal{R} \mid d, \mathcal{R}) \\
 & & (d \rightarrow e, \mathcal{R}) \\
 & & (e \rightarrow f, \mathcal{L})\}
\end{array}
$$

STACK	BUFFER	ACTION	RECORD
	bacdfe	SHIFT	
b	acdfe	SHIFT	
ba	cdfe	LEFT-ARC	$a \rightarrow b$
a	cdfe	SHIFT	
ac	dfe	RIGHT-ARC	$a \rightarrow c$
a	dfe	SHIFT	
ad	fe	SHIFT	
adf	e	SHIFT	
adfe		LEFT-ARC	$e \rightarrow f$
ade		RIGHT-ARC	$d \rightarrow e$
ad		RIGHT-ARC	$a \rightarrow d$
a		TERMINATE	$root \rightarrow a$

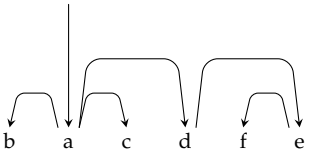


Figure 12: Above, shift-reduce parse for the string $bacdfe$. Note that, for a deterministic parse, lookahead is needed.

[22] You may have read about Google's *Parsey McParseface*: this is an English language dependency parser that uses word-embeddings and a neural network to score parse actions. A beam search is used to compare competing parses.

3. *Tree Adjoining Grammars*

Whereas in phrase structure grammar symbols were rewritten with other symbols, in a TREE ADJOINING GRAMMAR trees are rewritten as other trees. Like a phrase structure grammar, a tree adjoining grammar is defined by a set of terminals and non-terminals, as well as a starting symbol; but rather than phrase structure rules the grammar consists of sets of two types of elementary tree: INITIAL TREES and AUXILIARY TREES that may be combined using the operations of SUBSTITUTION and ADJUNCTION. As with a phrase structure tree a complete derivation is one that has the starting symbol at its root and terminal symbols only at the leaves. We can define a tree adjoining grammar $G_{tag} = (\mathcal{N}, \Sigma, S, \mathcal{I}, \mathcal{A})$ where:

definition of a TAG

$\mathcal{N}$ is the set of non-terminals

Σ is the set of terminals

S is a distinguished non-terminal $S \in \mathcal{N}$ that will be the root of complete derivations

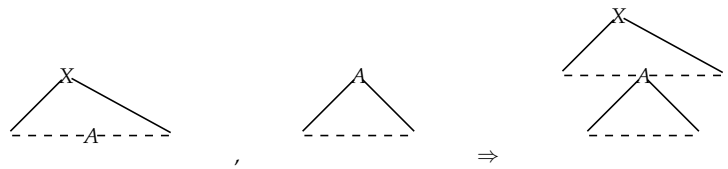
$\mathcal{I}$ is a set of ***initial trees*** (also known as α trees). Internal nodes of an α tree are drawn from $\mathcal{N}$ and the leaf nodes from $\Sigma \cup \mathcal{N} \cup \epsilon$.

$\mathcal{A}$ is a set of ***auxiliary trees*** (also know as β trees). Internal nodes of a β-tree are drawn from $\mathcal{N}$ and the leaf nodes from $\Sigma \cup \mathcal{N} \cup \epsilon$. One leaf of a β-tree is distinguished as the ***foot*** and will be the same non-terminal as at its root (the foot is often indicated with an asterisk).

A derivation is the result of recursive composition of elementary trees via one of two operations:

- SUBSTITUTION: a substitution may occur when a non-terminal leaf (that is, some $A \in \mathcal{N}$) of the current derivation tree is replaced by an α-tree that has A at its root. A schematic of a *substitution* where $X, A \in \mathcal{N}$:

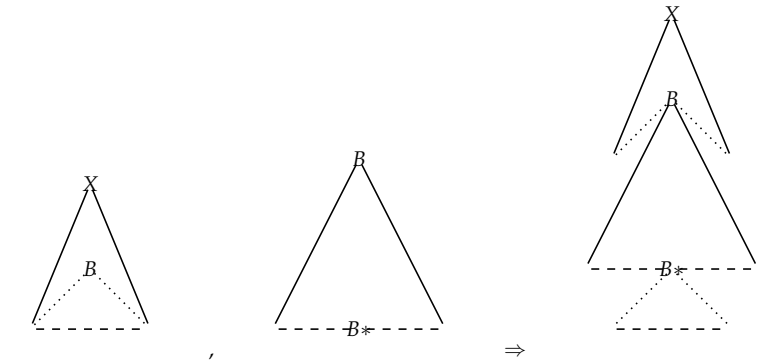
TAGs substitution



current derivation α-tree resulting tree

- ADJUNCTION: an adjunction may occur when an internal non-terminal node of the current derivation (some $B \in \mathcal{N}$) tree is replaced by a β tree that has a B at its root and *foot*.

TAGs adjunction



$\mathcal{L}(G_{tag})$ is the set of strings yielded from all of the possible derived trees for $G_{tag} = (\Sigma, \mathcal{N}, S, \mathcal{I}, \mathcal{A})$. A string from $\mathcal{L}(G_{tag})$ may be recognised in polynomial time—$O(n^6)$ where n is the length of the string. The class of tree adjoining languages turns out to be a superset of the class of context free grammars but a subset of the class context sensitive languages. As such, grammars weakly equivalent to tree adjoining grammars are known as MILDLY CONTEXT SENSITIVE languages. Tree adjoining grammars have been argued to have sufficient expressivity to capture all known natural languages (see the *Formal vs. Natural Language* handout).

mildly context sensitive

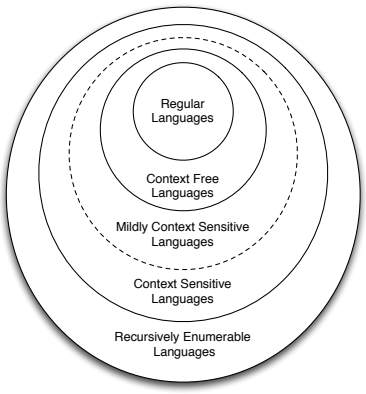
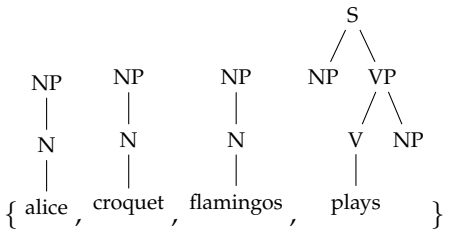


Figure 13: A Venn diagram showing the class of minimally context sensitive languages within the Chomsky hierarchy

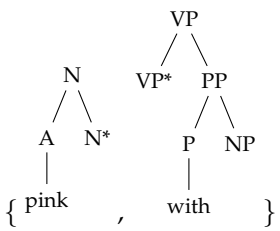
The following is a toy example of a tree adjoining grammar for English; we will use the grammar to derive the sentence *Alice plays croquet with pink flamingos*. $G_{tag} = (\mathcal{N}, \Sigma, S, \mathcal{I}, \mathcal{A})$ where:

$$\mathcal{N} = \{S, NP, VP, PP, N, P, V, A\}$$
$$\Sigma = \{alice, plays, croquet, with, pink, flamingos\}$$
$$S = \text{S}$$

$\mathcal{I}$ =



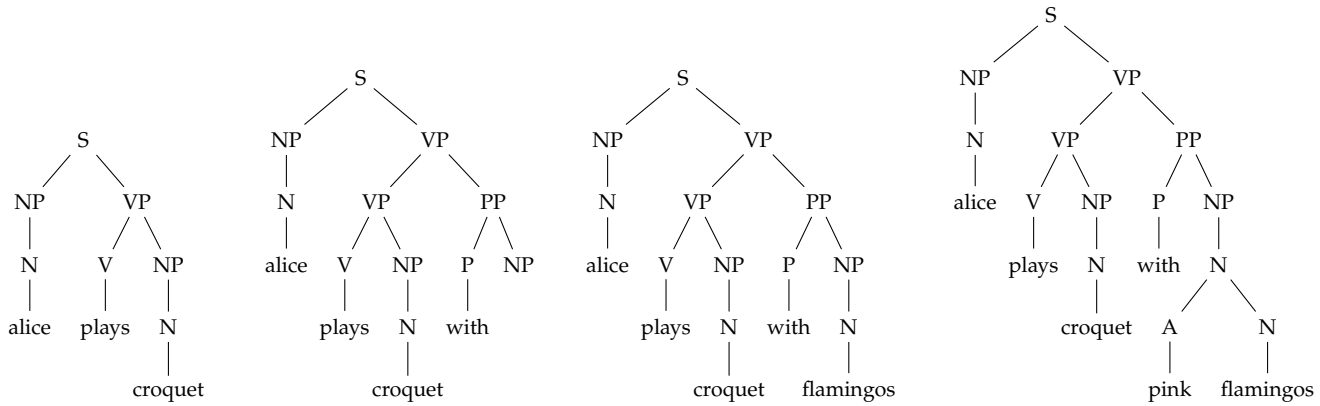
$\mathcal{A}$ =



Notice that information about the way in which words function in a sentence has been captured in the elementary trees. For instance, it is easy to see that *plays* is a verb that requires two arguments or that *pink* and *with* will modify existing non-terminals with extra information. When used to process natural language, gram-

mars that capture a word's *type* information in this way are referred to as ***lexicalized grammars***. A derivation for *Alice plays croquet with pink flamingos* may be obtained as follows:

lexicalized grammars



1. two substitutions 2. adjunction 3. substitution 4. adjunction

4. Categorial Grammars

In a ***classic categorial grammar*** all symbols in the alphabet are associated with a finite number of *types*.[23] Types are formed from primitive types using two operators, $\backslash$ and $/$. If P_r is the set of ***primitive types*** then the set of all types, T_p, satisfies:

- $P_r \subset T_p$
- if $A \in T_p$ and $B \in T_p$ then $A \backslash B \in T_p$
- if $A \in T_p$ and $B \in T_p$ then $A / B \in T_p$

[23] when used for modelling natural language, these *types* reflect linguistic grammatical function as did the tree-lets in the tree adjoining grammar. Hence a categorial grammar is also a *lexicalized grammar*

Due to this definition, it is possible to arrange types in a hierarchy: a type A is a *subtype* of B if A occurs in B (that is, A is a subtype of B iff $A = B$; or ($B = B_1 \backslash B_2$ or $B = B_1 / B_2$) and A is a subtype of B_1 or B_2).

A relation, $\mathcal{R}$, maps symbols in the alphabet Σ to members of T_p. A grammar that associates at most one type to each symbol in Σ is called a ***rigid grammar*** whereas a grammar that assigns at most k types to any symbol is a ***k-valued grammar***. We can define a classic categorial grammar $G_{cg} = (\Sigma, P_r, S, \mathcal{R})$ where:

definition of a categorial grammar

Σ is the alphabet/set of terminals

P_r is the set of primitive types

S is a distinguished member of the primitive types $S \in P_r$ that will be the root of complete derivations

$\mathcal{R}$ is a relation $\Sigma \times T_p$ where T_p is the set of all types as generated from P_r as described above

A string has a valid parse if the types assigned to its symbols can be combined to produce a derivation tree with root S. Types may be combined using the two rules of function application:

function application

- FORWARD APPLICATION is indicated by the symbol $>$:

$$\frac{A/B \qquad B}{A} >$$

- BACKWARD APPLICATION is indicated by the symbol $<$:

$$\frac{B \qquad A\backslash B}{A} <$$

Note that classic categorial grammars have a simple correspondence with context free grammars:

- Given a classic categorial grammar, G_{cg}, with lexicon Σ, then the ***range*** of G_{cg} is the *range* of the relation R (that is, all of the types in T_p that were assigned to a symbol in Σ).
- So $T_p(G_{cg}) = \{A | A \in range(\mathcal{R})$ or A is a subtype of $range(\mathcal{R})\}$.
- To create a context free grammar $G_{cfg} = (\mathcal{N}, \Sigma, S, \mathcal{P})$ with strong equivalence to $G_{cg} = (\Sigma, P_r, S, \mathcal{R})$ we can define G_{cfg} as:

$$\begin{array}{rcl} \mathcal{N} & = & P_r \cup range(\mathcal{R}) \\ \Sigma & = & \Sigma \\ S & = & S \\ P & = & \{B \rightarrow A\ B\backslash A | B\backslash A \in T_p\} \\ & & \cup \{B \rightarrow B/A\ A | B/A \in T_p\} \\ & & \cup \{A \rightarrow a | \mathcal{R} : a \rightarrow A\} \end{array}$$

For illustration, below is a derivation tree for the string xyz using the grammar $G_{cg} = (\Sigma, P_r, S, \mathcal{R})$ where:

$$\begin{array}{rcl} Pr & = & \{S, A, B\} \\ \Sigma & = & \{x, y, z\} \\ S & = & S \\ \mathcal{R} & = & \{(x, A), (y, S\backslash A/B), (z, B)\} \end{array}$$

$$\dfrac{\dfrac{x}{A}\,\mathcal{R} \qquad \dfrac{\dfrac{y}{S\backslash A/B}\,\mathcal{R} \qquad \dfrac{z}{B}\,\mathcal{R}}{S\backslash A}\,>}{S}\,<$$

If we are to use a categorial grammar to model natural languages we may need something more powerful than equivalence to a CFG. There is a variant of categorial grammar called COMBINATORY CATEGORIAL GRAMMAR that exhibits equivalence to the mildly context sensitive languages. A combinatory categorial grammar allows FUNCTION COMPOSITION and TYPE RAISING as well as the function application described above. These extra rules are defined below where $A, B, C, T \in T_p$:

For a natural language example, below is a derivation tree for the string *alice chases rabbits* using the grammar $G_{cg} = (\Sigma, P_r, S, \mathcal{R})$ where:

$$\begin{array}{rcl} Pr & = & \{S, NP\} \\ \Sigma & = & \{alice, chases, rabbits\} \\ S & = & S \\ \mathcal{R} & = & \{(alice, NP), (chases, S\backslash NP/NP), \\ & & (rabbits, NP)\} \end{array}$$

$$\dfrac{\dfrac{alice}{NP}\,\mathcal{R} \qquad \dfrac{\dfrac{chases}{S\backslash NP/NP}\,\mathcal{R} \qquad \dfrac{rabbits}{NP}\,\mathcal{R}}{S\backslash NP}\,>}{S}\,<$$

function composition

- FUNCTION COMPOSITION is indicated by the symbols $B_>$ and $B_<$:

$$\frac{A/B \qquad B/C}{A/C} B_>$$

$$\frac{B\backslash C \qquad A\backslash B}{A\backslash C} B_<$$

type raising

- TYPE-RAISING is indicated by the symbols $T_>$ and $T_<$:

$$\frac{A}{T/(T\backslash A)} T_>$$

$$\frac{A}{T\backslash(T/A)} T_<$$