NL Grammar Hierarchies And Formal Languages

Quick Review of Regular Expressions, Finite State Automata, Markov Algorithms Representations of Languages, Grammars

Regular Expressions (summary)

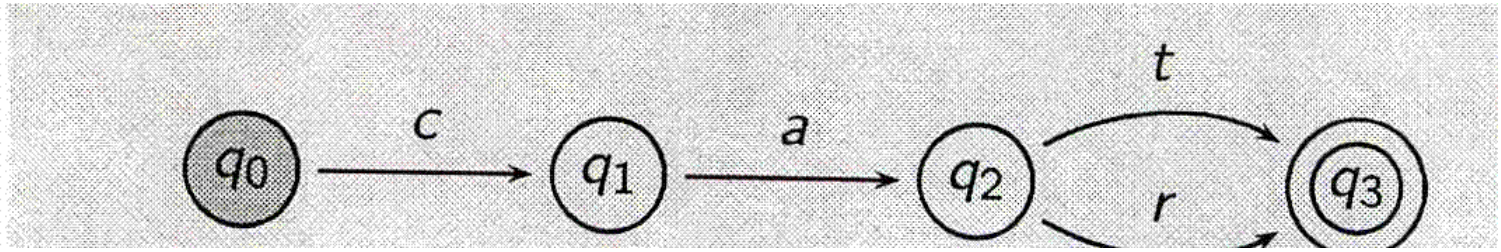
- (*empty set*) ∅ denoting the set ∅.
- (*empty string*) ε denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) *a* in Σ denoting the set containing only the character *a*.
- The following operations are defined:
- (*concatenation*) *RS* denoting the set { αβ | α in *R* and β in *S* }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R* | *S* denoting the set union of *R* and *S*. For example {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}.
- (*Kleene star*) *R** denoting the smallest superset of *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.
- nothing else is a regular expression over ∑

Finite State Automata

- Automata are models of computation: they compute languages.
- A finite-state automaton is a five-tuple $\{Q, q_0, \sum, \delta, F\}$, where $\sum$ is a finite set of alphabet symbols, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final (accepting) states and $\delta : Q \times \sum \times Q$ is a relation from states and alphabet symbols to states.

Finite State Automata

- Example: Finite-state automaton
- Q = {q0, q1, q2, q3}
- ∑ = {c, a, t, r}
- F = {q3}
- δ = {q0, c, q1>, q1, a, q2>, q2, t, q3>, q2, r , q3>}



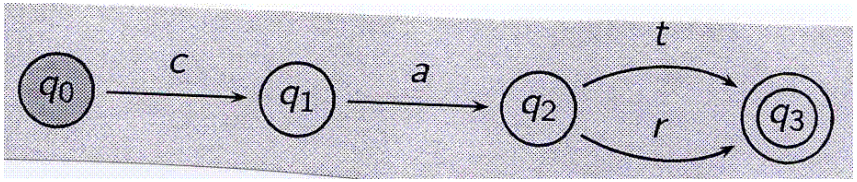
Finite State Automata

- The reflexive transitive extension of the transition relation δ is a new relation, $\hat{\delta}$, defined as follows:
  - for every state $q \in Q$, $(q, \varepsilon, q) \in \hat{\delta}$
  - for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q,w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$ then $(q,w \cdot a, q'') \in \hat{\delta}$.

Finite State Automata

- Example: Paths

  For the finite-state automaton:



  ^δ is the following set of triples:

  q0, Q, q0>, q1, Q, q1>, q2, Q, q2>, q3, Q, q3>,

  q0, c, q1>, q1, a, q2>, q2, t, q3>, q2, r , q3>,

  q0, ca, q2>, q1, at, q3>, q1, ar , q3>,

  q0, cat, q3>, q0, car , q3>

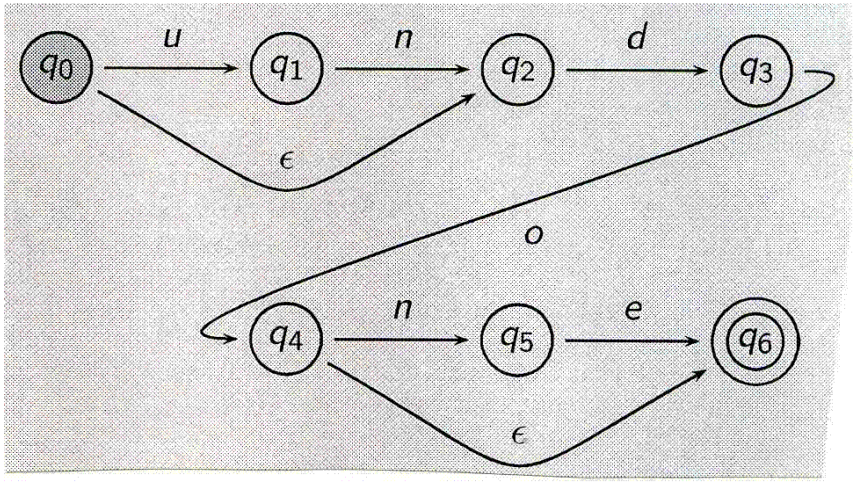
Finite State Automata

An extension: ε-moves.

- The transition relation δ is extended to:

$$\delta \subseteq Q \times (\sum \cup \{\varepsilon\}) \times Q$$

Example: Automata with ε-moves - an automaton accepting the language {do, undo, done, undone}:



Formal language theory – definitions

- If L is a language then the reversal of L, denoted L^R, is the language $\{w \mid w^R \in L\}$.
- If L1 and L2 are languages, then $L1 \cdot L2 = \{w1 \cdot w2 \mid w1 \in L1 \text{ and } w2 \in L2\}$.
- Example: Language operations
  - Let L1 = {i, you, he, she, it, we, they}, L2 = {smile, sleep}.
  - Then $L1^R$ = {i, uoy, eh, ehs, ti, ew, yeht} and L1 · L2 = {ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep}.

Formal language theory – definitions

- If L is a language then the reversal of L, denoted L^R, is the language $\{w \mid w^R \in L\}$.
- If L1 and L2 are languages, then $L1 \cdot L2 = \{w1 \cdot w2 \mid w1 \in L1 \text{ and } w2 \in L2\}$.
- Example: Language operations
  - Let L1 = {i, you, he, she, it, we, they}, L2 = {smile, sleep}.
  - Then $L1^R$ = {i, uoy, eh, ehs, ti, ew, yeht} and L1 · L2 = {ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep}.

Formal language theory – definitions

- If L is a language then the reversal of L, denoted L^R, is the language $\{w \mid w^R \in L\}$.
- If L1 and L2 are languages, then $L1 \cdot L2 = \{w1 \cdot w2 \mid w1 \in L1 \text{ and } w2 \in L2\}$.
- Example: Language operations
  - Let L1 = {i, you, he, she, it, we, they}, L2 = {smile, sleep}.
  - Then $L1^R$ = {i, uoy, eh, ehs, ti, ew, yeht} and L1 · L2 = {ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep}.

Formal language theory – definitions

- If L is a language then $L^0 = \{\varepsilon\}$.
- Then, for $i > 0$, $L^i = L \cdot L^{i-1}$.
- Example: Language exponentiation

  Let L be the set of words {bau, haus, hof, frau}. Then $L^0 = \{\varepsilon\}$, $L^1 = L$ and L^2 = {baubau, bauhaus, bauhof, baufrau, hausbau, haushaus, haushof, hausfrau, hofbau, hofhaus, hofhof, hoffrau, fraubau, frauhaus, frauhof, fraufrau}.

Formal language theory – definitions

- The Kleene closure of L and is denoted L* and is defined as $\bigcup_{i=0}^{\infty} L^i$.
- $L^+ = \bigcup_{i=0}^{\infty} i=1\ L^i$
- Example: Kleene closure

  Let L = {dog, cat}. Observe that $L^0 = \{\varepsilon\}$, $L^1 = \{dog, cat\}$, $L^2 = \{catcat, catdog, dogcat, dogdog\}$, etc. Thus L* contains, among its infinite set of strings, the strings ε, cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catdogcat, dogcatcat, dogdogcat, etc.
- The notation for * should now become clear: it is simply a special case of L*, where L = ∑

Markov Algorithms

A Markov Algorithm is a finite sequence $P_1, P_2,...,P_n$ of Markov productions to be applied to strings in a given alphabet according to the following rules. Let S be a given string. The sequence is searched to find the first production P_i whose antecedent occurs in S. If no such production exists, the operation of the algorithm halts without change in S. If there is a production in the algorithm whose antecedent occurs in S, the first such production is applied to S. If this is a conclusive production, the operation of the algorithm halts without further change in S. If this is a simple production, a new search is conducted using the string S' into which S has been transformed. If the operation of the algorithm ultimately ceases with a string S*, we say that S* is the result of applying the algorithm to S.

Markov Algorithms

Example:

Take the alphabet to be {a, b, c, d}. The algorithm is given below.

- Algorithm M1

M11: [conclusive]	a *d*	→	•*d* c
M12: [simple]	*b* a	→	W
M13: [simple]	a	→	*b* c
M14: [simple]	*b* c	→	*b b* a
M15: [simple]	W	→	a

Taking S = "*dcb*" we apply the algorithm

by M15 *dcb* becomes a*dcb*

by M11 a*dcb* becomes *d*c*cb* and halts.

Markov Algorithms

Example:

Let β be a marker not in the alphabet. If S is a string in the alphabet, the result of applying algorithm M3 to S is the string SA.

Algorithm M3

M31: [interchange]	$\beta\,\delta \rightarrow \delta\,\beta$	δ, A ∈ member of alphabet
M32: [conclusive]	$\beta \rightarrow \bullet A$	
M33:	$W \rightarrow \beta$	

Since S initially does not contain β, the third production is then used to move β past the symbols in S. If S contains n occurrences of symbols, then after n steps we obtain the string Sβ. At this point the first production no longer applies, and the second production produces SA. Since this production is conclusive, the string SA is then the result.

Markov Algorithms

In the preceding example, we have introduced a new notation. Namely, in the first production we have used the variable δ which ranges over the symbols in the alphabet. Thus the first line is not really a production, but rather a production schema, denoting all the productions which can be obtained by substituting symbols of the alphabet for δ.

Because of the manner in which the Markov algorithms are used, the order in which the productions are written is vital. If the first two lines of algorithm M3 were interchanged, the result would be to transform *S* into *AS*, rather than into *SA*, and the productions represented by $\beta\delta \rightarrow \delta\beta$ would never be used.

Markov Algorithms

Example:

Another procedure which is quite common is that of reversing a string of characters. We do this by moving the first character to the end as before, then moving the next character down to the position just preceding the first character, and so on. Markers: μ, β

Algorithm M10

M101:	μβ	→ •*W*	δ, *f* ∈ members of the alphabet
M102:	μδβ	→ βδ	
M103:	μδ*f*	→ *f*μδ	
M104:	μδ	→ βδ	
M105:	*W*	→ μ	

Markov Algorithms

Illustrating this algorithm on the string "ABCD" we have

by M105 => μ A B C D
by M103 => B μ A C D
by M103 => B C μ A D
by M103 => B C D μ A
by M104 => B C D β A
by M105 => μ B C D β A
by M103 => C μ B D β A
by M103 => C D μ B β A
by M102 => C D β B A
by M105 => μ C D β B A
by M103 => D μ C β B A
by M102 => D β C B A

by M105 => μ D β C B A
by M102 => β D C B A
by M105 => μ β D C B A
by M101 => D C B A

Representations of Languages

How can we decide on a finite representations for languages? One way to represent a language is to give an algorithm which determines if a sentence is in the language or not. A more general way is to give a procedure which halts with the answer "yes" for sentences in the language and either does not terminate or else halts with the answer "no" for sentences not in the language. Such a procedure or algorithm is said to recognize the language. As it turns out, there are languages we can recognize by a procedure, but not by any algorithm.

The method described can represent languages from a recognition point of view. We can also represent languages from a generative point of view. That is, we can give a procedure which systematically generates successive sentences of the language in some order.

Representations of Languages

If we can recognize the sentences of a language over alphabet V with either an algorithm or a procedure, then we can generate the language, since we can systematically generate all sentences in V*, test each sentence to see if it is in the language, and output in a list only those sentences in the language. One must be careful in doing so. For if we generate the sentences in order and use a procedure which does not always halt for testing the sentences, we never get beyond the first sentence for which the procedure does not halt.

The way to get around this problem is to organize the testing in such a manner that the procedure never continues to test one sentence forever.

This organization requires that we introduce several constructions.

Representations of Languages

Assume that there are p symbols in V. We can think of the sentences of V* as numbers represented in base p, plus the empty sentence e. We can number the sentences in order of increasing length and in "numerical" order for sentences of the same length. In Fig. (a) we have the enumeration of the sentences of{a, b, c}*.

We have implicitly assumed that a, b, and c correspond to 0, 1, and 2, respectively. (This argument shows that the set of sentences over an alphabet is countable.)

1	E
2	a
3	b
4	c
5	aa
6	ab
7	ac
8	ba
9	bb
•	°

Fig. (a) The enumeration of sentences in {a, b, c}*

Representations of Languages

Let P be a procedure for testing a sentence to see if the sentence is in a language L. We assume that P can be broken down into discrete steps so that it makes sense to talk about the ith step in the procedure for any given sentence. Before giving a procedure to enumerate the sentences of L, we first give a procedure to enumerate pairs of positive integers.

We can map all ordered pairs of positive integers (x, y) onto the set of positive integers as shown in Fig. (b) by the formula

$$z = \{[(x + y - 1)(x + y - 2)]/2\} + y .$$

x↓ y→	1	2	3	4	5
1	1	3	6	10	15
2	2	5	9	14	
3	4	8	13	.	
4	7	12			.
5	11				

Fig. (b) Mapping of ordered pairs of integers onto the integers

Representations of Languages

We can enumerate ordered pairs of integers according to the assigned value of z. Thus the first few pairs are (1, 1), (2, 1), (1, 2), (3, 1), (2, 2), ... Given any pair of integers (i, j), it will eventually appear in the list. In fact, it will be the [(i+J - 1) (i+j - 2)] / 2 + jth pair enumerated.

We can now give a procedure for enumerating the strings of L.

Enumerate ordered pairs of integers. When the pair (i, j) is enumerated, generate the ith sentence in V* and apply the first j steps of procedure P to the sentence. Whenever it is determined that a generated sentence is in L, add that sentence to the list of members of L. If word i is in L, it will be so determined by P in j steps, for some finite j. When (i, j) is enumerated, word i will be generated. This procedure will indeed enumerate all sentences in L.

Representations of Languages

If we have a procedure for generating the sentences of a language, then we can construct a procedure for recognizing the sentences of the language, but not necessarily an algorithm. To determine if a sentence x is in L, simply enumerate the sentences of L and compare x with each sentence. If x is generated, the procedure halts, having recognized x as being in L. Of course, if x is not in L, the procedure will never terminate.

A language whose sentences can be generated by a procedure is said to be recursively enumerable. Alternatively, a language is said to be recursively enumerable if there is a procedure for recognizing the sentences of the language. A language is said to be recursive if there exists an algorithm for recognizing the language. In formal language theory, the class of recursive languages is a proper subset of the class of recursively enumerable languages.

Representations of Languages

There are languages which are not even recursively enumerable. That is, there are languages for which we cannot even effectively list the sentences of the language. Asking the question, "What is language theory ?," we can say that language theory is the study of sets of strings of symbols, their representations, structures, and properties. Beyond this, we shall leave the question to be answered in a course on formal grammars.

Grammars

There is one class of generating systems of primary interest to us, systems known as grammars. The concept of a grammar was originally formalized by linguists in their study of natural languages. Linguists were concerned not only with defining precisely what is or is not a valid sentence of a language, but also with providing structural descriptions of the sentences. One of their goals was to develop a formal grammar capable of describing English.

It was hoped that if, for example, one had a formal grammar to describe the English language, one could use the computer in ways that require it to "understand" English. Such a use might be language translation or the computer solution of word problems. To date, this goal is for the most part unrealized in its totality.

Grammars

We do not have a definitive grammar for English, and there is even disagreement as to what types of formal grammar are capable of describing English. However, in describing computer languages, better results have been achieved. For example, the Backus Normal Form used to describe ALGOL is a "context free grammar," a type of grammar with which we shall study and build upon.

We are all familiar with the idea of diagramming or parsing an English sentence. For example, the sentence "The little boy ran quickly" is parsed by noting that the sentence consists of the noun phrase "The little boy" followed by the verb phrase "ran quickly." The noun phrase is then broken down into the singular noun "boy" modified by the two adjectives "The" and "little." The verb phrase is broken down into the singular verb "ran" modified by the adverb "quickly." This sentence structure is indicated in the diagram of Fig. (c).

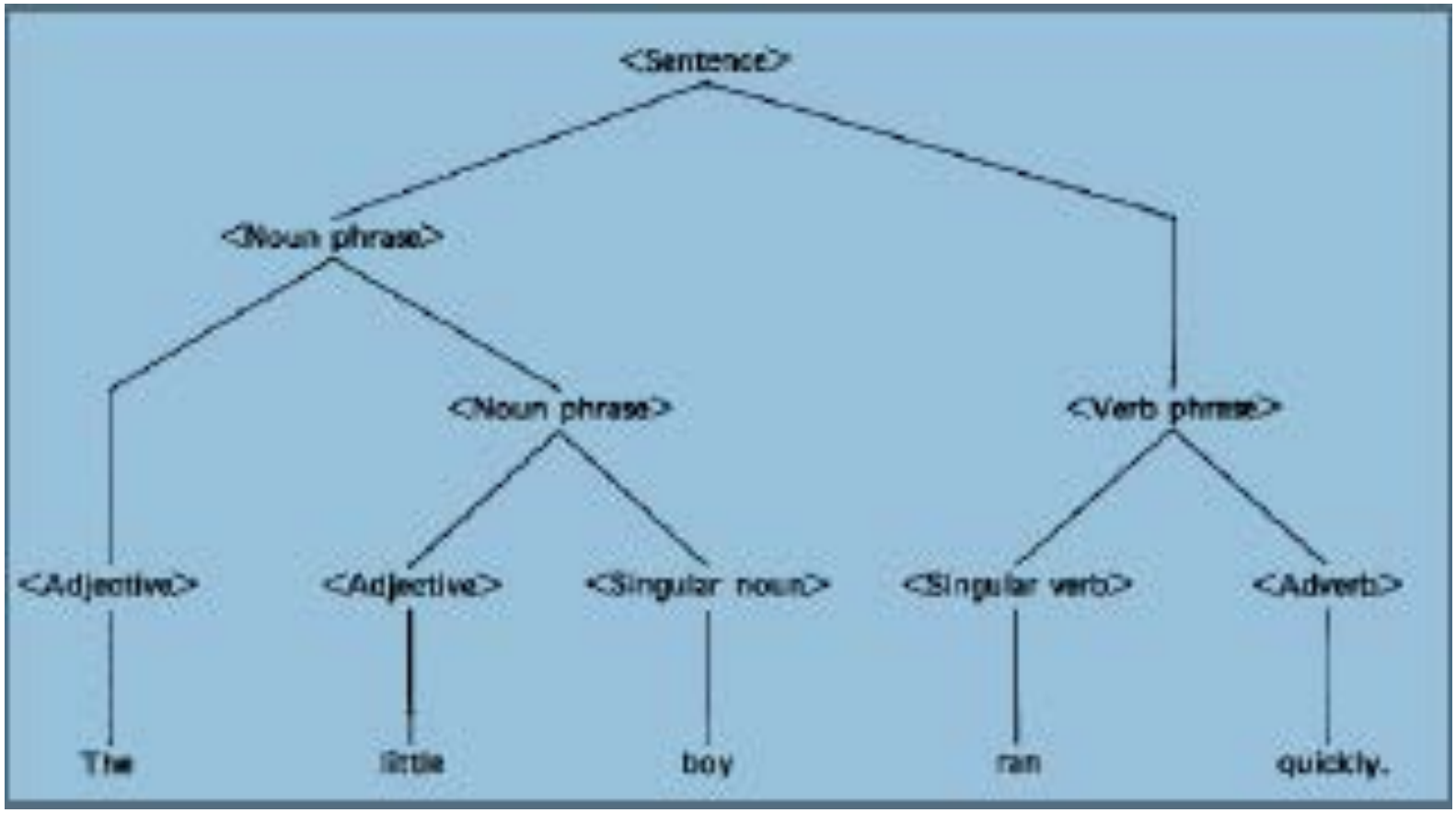


Fig. (c). A diagram of the sentence "The little boy ran quickly."

Grammars

We recognize the sentence structure as being grammatically correct. If we had a complete set of rules for parsing all English sentences, then we would have a technique for determining whether or not a sentence is grammatically correct. However, such a set does not exist. Part of the reason for this stems from the fact that there are no clear rules for determining precisely what constitutes a sentence.

The rules we applied to parsing the above sentence can be written in the following form:

<sentence> → <noun phrase> <verb phrase>
<noun phrase> → <adjective> <noun phrase>
<noun phrase> → <adjective> <singular noun>
<verb phrase> → <singular verb> <adverb>
<adjective> → The | little
<singular noun> → boy
<singular verb> → ran
<adverb> → quickly

Grammars

The arrow in the rules indicates that the item to the left of the arrow can generate the items to the right of the arrow. Note that we have enclosed the names of the parts of the sentence such as noun, verb, verb phrase, etc., in brackets to avoid confusion with the English words and phrases "noun," "verb," "verb phrase," etc.

Note that we cannot only test sentences for their grammatical correctness, but can also generate grammatically correct sentences by starting with (sentence) and replacing (sentence) by (noun phrase) followed by (verb phrase). Next we select one of the two rules for (noun phrase) and apply it, and so on, until no further application of the rules is possible. Thus any one of an infinite number of sentences can be derived, e.g., a string of occurrences of "the" and "little" followed by "boy ran quickly" such as "little the the boy ran quickly" can be generated. Most of the sentences do not make sense but, nevertheless, are grammatically correct.

Formal Notion of a Grammar

- Formally, we denote a *grammar* G by (Vn, Vt, P, S). The symbols Vn, Vt, P, and S are, respectively, the *variables, terminals, productions*, and *start symbol*. Vn, Vt, and P are finite sets. We assume that Vn and Vt contain no elements in common; i.e.,

$$Vn \cap Vt = \varepsilon$$

where ε denotes the empty set and, conventionally

$$Vn \cup Vt = V$$

The set of productions P consists of expressions of the form $a \rightarrow b$, where a is a string in V+ and b is a string in V*. Finally, S is always a symbol in Vn.

Formal Notion of a Grammar

- Customarily, we shall use capital Latin alphabet letters for variables. Lower case letters at the beginning of the Latin alphabet are used for terminals. Strings of terminals are denoted by lower case letters near the end of the Latin alphabet, and strings of variables and terminals are denoted by lower case Greek letters.
- Given the grammar G = (Vn, Vt, P, S), we define the language it generates as follows. If $a \rightarrow b$ is a production of P and g and d are strings in V*, then $gad \rightarrow gbd$.
- In English we say gad *directly derives* gbd in *grammar G.* We say that the production $a \rightarrow b$ is applied to the string gad to obtain gbd. Thus G relates two strings exactly when the second is obtained from the first by the application of a single production.

Formal Notion of a Grammar

We define grammars G1 and G2 to be equivalent if L(G1) = L(G2).

Example. Let us consider a grammar G = (VN, VT, P, S), where VN = {S}, VT = {0, 1}, P = {S ~ 0S1, S -+ 01}. Here, S is the only variable, 0 and 1 are terminals. There are two productions, S ⟶ 0S1 and S ⟶ 01. By applying the first production n - 1 times, followed by an application of the second production, we have

$$S \longrightarrow 0S1 \longrightarrow 00S11 \longrightarrow 03S13 \Longrightarrow \ldots \Longrightarrow 0^{n-i}S1^{n-1} \longrightarrow 0^n1^n.\S$$

Furthermore, these are the only strings in L(G). After using the second production, we find that the number of S's in the sentential form decreases by one. Each time the first production is used, the number of S's remains the same. Thus, after using S ⟶ 01, no S's remain in the resulting string.

Since both productions have an S on the left, the only order in which the productions can be applied is S-+ 0S1 some number of times followed by one application of S ~ 01. Thus, $L(G) = \{0^n1^n \mid n \geq 1\}$.

Formal Notion of a Grammar

The example was a simple example of a grammar. It was relatively easy to determine which words were derivable and which were not. In general, it may be exceedingly hard to determine what is generated by the grammar.

Types of Grammars

The Chomsky hierarchy of languages

- A hierarchy of classes of languages, viewed as sets of strings, ordered by their "complexity". The higher the language is in the hierarchy, the more "complex" it is.
- In particular, the class of languages in one class properly includes the languages in lower classes.
- There exists a correspondence between the class of languages and the format of phrase-structure rules necessary for generating all its languages. The more restricted are the rules, the lower in the hierarchy are the languages they generate.

The Chomsky hierarchy of languages

- Recursively enumerable languages
- Context-sensitive languages
- Context-free languages
- Regular languages

The Chomsky hierarchy of languages

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages.
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

The Chomsky hierarchy of languages

- Type-2 grammars (context-free grammars) generate the context-free languages defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages recognized by a non-deterministic pushdown automaton. Context-free languages are the basis for most programming languages.
- Type-3 grammars (regular grammars) generate the regular languages restricting its rules to a single nonterminal on the LHS and a RHS consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the RHS of any rule. These languages are those that can be decided by a finite state automaton. This family of languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

The Chomsky hierarchy of languages

Every regular language is context-free, every context-free language, not containing the empty string, is context-sensitive and every context-sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages which are not context-sensitive, context-sensitive languages which are not context-free and context-free languages which are not regular.

The Chomsky hierarchy of languages

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.'

Grammar	Languages	Automaton	Production rules (constraints)
Type -0	Recursively enumerable	Turing mac hine	α _ β (no restrictions)
Type -1	Context -sensitive	Linear -bounded non - deterministic Turing mac hine	$\alpha A \beta$ _ $\alpha \gamma \beta$
Type -2	Context -free	Non -deterministic pushdown autom aton	A _ γ
Type -3	Regular	Finite state au tom aton	A _ a and A _ a B

Why is it interesting?

- The hierarchy represents some informal notion of the complexity of natural languages
- It can help accept or reject linguistic theories
- It can shed light on questions of human processing of language

What exactly is the question?

- When viewed as a set of strings, is English a regular language? Is it context-free? How about Hebrew?
- Competence vs. Performance
  - This is the dog, that worried the cat, that killed the rat, that ate the malt, that lay in the house that Jack built.
  - This is the malt that the rat that the cat that the dog worried killed ate

Where are natural languages located?

- Chomsky (1957): "English is not a regular language"
- As for context-free languages, "I do not know whether or not English is itself literally outside the range of such analyses"

How not to do it

An introduction to the principles of transformational syntax (Akmajian and Heny, 1976)

"Since there seem to be no way of using such PS rules to represent an obviously significant generalization about one language, namely, English, we can be sure that phrase structure grammars cannot possibly represent all the significant aspects of language structure. We must introduce a new kind of rule that will permit us to do so."

How not to do it

Example: Syntax (Peter Culicover, 1976)

> *In general, for any phrase structure grammar containing a finite number of rules it will always be possible to construct a sentence that the grammar will not generate. In fact, because of recursion there will always be an infinite number of such sentences. Hence, the phrase structure analysis will not be sufficient to generate English.*

How not to do it

Example: Transformational grammar (Grinder & Elgin, 1973)

the girl saw the boy

*the girl kiss the boy

this well-known syntactic phenomenon demonstrates clearly the inadequacy of ... context-free phrase-structure grammars...

How not to do it

The defining characteristic of a context-free rule is that the symbol to be rewritten is to be rewritten without reference to the context in which it occurs. By definition, one cannot write a context-free rule that will expand the symbol *V* into *kiss* in the context of being immediately preceded by the sequence *the girls* and that will expand the symbol *V* into *kisses* in the context of being immediately preceded by the sequence *the girl*. Any set of context-free rules that generate (correctly) the sequences *the girl kisses the boy* and *the girls kiss the boy* will also generate (incorrectly) the sequences *the girl kiss the boy* and *the girls kisses the boy.*

I'd like -
I'd like to know
what this whole show
is all about
before it's out