III B.TECH CSE-AI

NATURAL LANGUAGE PROCESSING

TEXT BOOK: James Allen, **Natural Language Understanding**, 2nd Edition, 2003, Pearson Education

UNIT–II Grammars and Parsing

Grammars and Parsing- Top- Down and Bottom-Up Parsers, Transition Network Grammars, Feature Systems and Augmented Grammars, Morphological Analysis and the Lexicon, Parsing with Features, Augmented Transition Networks, Bayes Rule, Shannon game, Entropy and Cross Entropy.

Grammars and Parsing:

Natural language has an underlying structure usually referred to under the heading of Syntax. The fundamental idea of syntax is that words group together to form so-called constituents i.e. groups of words or phrases which behave as a single unit. These constituents can combine together to form bigger constituents and eventually sentences.

The instance, *John*, *the man*, *the man with a hat* and *almost every man* are constituents (called Noun Phrases or NP for short) because they all can appear in the same syntactic context (they can all function as the subject or the object of a verb for instance). Moreover, the NP constituent the man with a hat can combine with the VP (Verb Phrase) constituent run to form a S (sentence) constituent.

Grammar: It is a set of rules which checks whether a string belongs to a particular language a not.

A program consists of various strings of characters. But, every string is not a proper or meaningful string. So, to identify valid strings in a language, some rules should be specified to check whether the string is valid or not. These rules are nothing but make **Grammar**.

Example − In English Language, Grammar checks whether the string of characters is acceptable or not, i.e., checks whether nouns, verbs, adverbs, etc. are in the proper sequence.

Grammar

Context-Free Grammar(CFG):

Context-free grammar (CFG) is a type of formal grammar used in natural language processing (NLP) and computational linguistics. It provides a way to describe the structure of a language in terms of a set of rules for generating its sentences.

Formally, Context-Free Grammar (G) can be defined as − It is a 4-tuple (V,T,P,S)

- V is a set of Non-Terminals or Variables. Examples N,NP, V, VP, PP, Det etc
- T is a set of terminals. ∑={names, things, places & verbs}
- P is a set of Productions or set of rules
- S is a starting symbol

The following Grammar productions are used in NLP parsers.

1. S -> NP VP
2. NP -> ART N
3. NP -> ART ADJ N
4. VP -> V
5. VP -> V NP

Grammar rule	Example
S → NPVP	I + want a morning flight
NP → Pronoun	I
NP → Proper-Noun	Los Angeles
NP → Det Nominal	a flight
Nominal → Nominal Noun	morning flight
Nominal → Noun	flights
VP → Verb	do
VP → Verb NP	want + a flight
VP → Verb NP PP	leave + Boston + in the morning
VP → Verb PP	leaving + on Thursday
PP → Preposition NP	from + Los Angeles

Derivations:

A derivation is a sequence of rule applications that derive a terminal string $w = w1 \ldots wn$ from S
• For example:

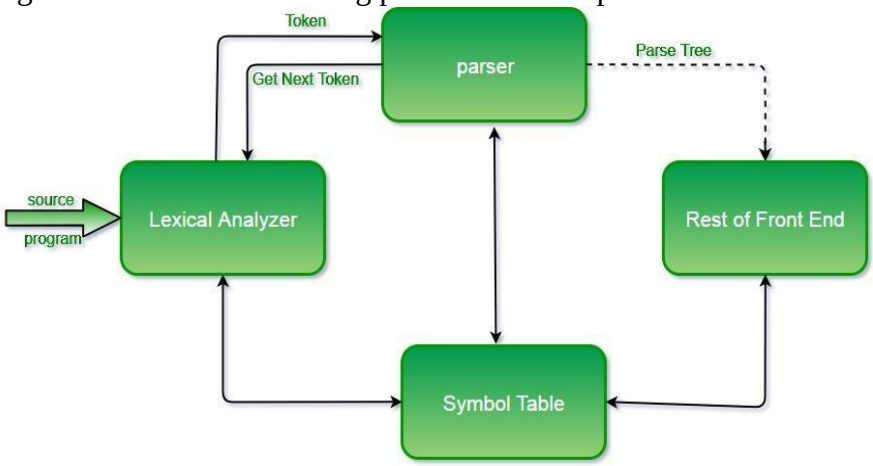
S→NP VP
Pro VP
I VP
I Verb NP
I prefer NP
I prefer Det Nom
I prefer a Nom
I prefer a Nom Noun
I prefer a Noun Noun
I prefer a morning Noun
I prefer a morning flight

Parsing:

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser.

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

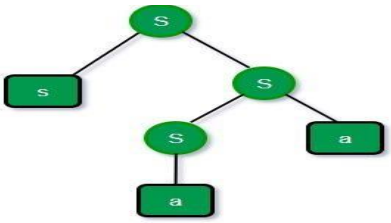
The following diagram describes the working procedure of the parser.

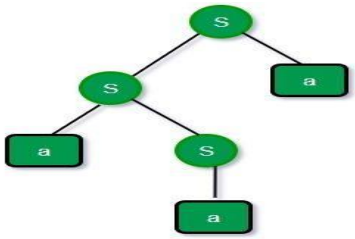


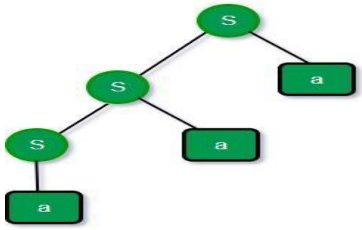
Ambiguity

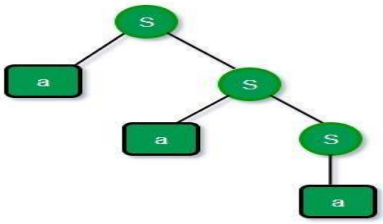
A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
Eg- consider a grammar
S -> aS | Sa | a

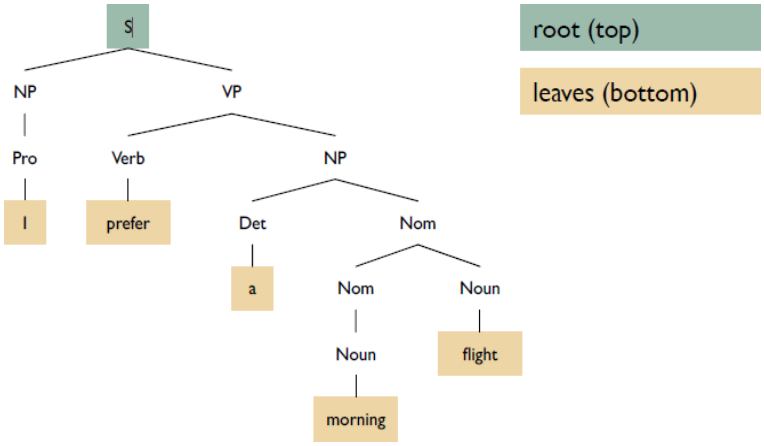
Now for string aaa, we will have 4 parse trees, hence ambiguous











Basic concepts of parsing:

Two problems for grammar *G* and string *w*:

- Recognition: determine if *G* accepts *w*
- Parsing: retrieve (all or some) parse trees assigned to *w* by *G*

Two basic search strategies:

- Top-down parser: start at the root of the tree
- Bottom-up parser: start at the leaves

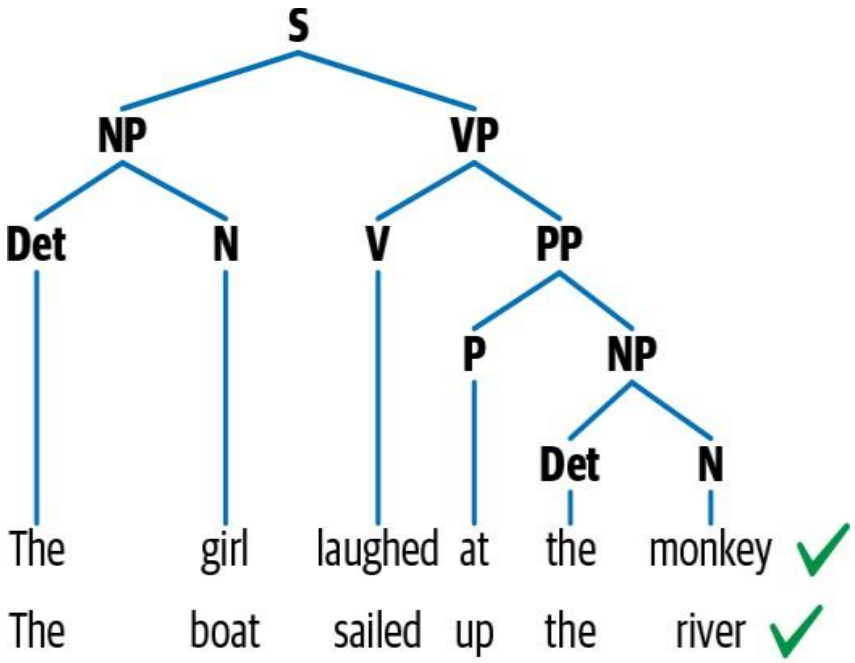


Figure 1-6. Syntactic structure of two syntactically similar sentences

Top-Down Parser

A parsing algorithm can be described as a procedure that searches through various ways of combining grammatical rules to find a combination that generates a tree that could be the structure of the input sentence. In other words, the algorithm will say whether a certain sentence is accepted by the grammar or not. The top-down parsing method is related in many artificial intelligence (AI) search applications.

A **top-down parser** starts with the S symbol and attempts to rewrite it into a sequence of terminal symbols that matches the classes of the words in the input sentence. The state of the parse at any given time can be represented as a list of symbols that are the result of operations applied so far, called the symbol list.

For example, the parser starts in the state (5) and after applying the rule S -> NP VP the symbol list will be (NP VP). If it then applies the rule NP ->ART N, the symbol list will be (ART N VP), and so on.

1. S -> NP VP
2. NP -> ART N
3. NP -> ART ADJ N
4. VP -> V
5. VP -> V NP

The parser could continue in this fashion until the state consisted entirely of terminal symbols, and then it could check the input sentence to see if it matched. The lexical analyser will produce list of words from the given sentence. A very small lexicon for use in the examples is

cried: V
dogs: N, V
the: ART

Positions fall between the words, with 1 being the position before the first word. For example, here is a sentence with its positions indicated:

1 The 2 dogs 3 cried 4

A typical parse state would be

((N VP) 2)

indicating that the parser needs to find an N followed by a VP, starting at position two. New states are generated from old states depending on whether the first symbol is a lexical symbol or not. If it is a lexical symbol, like N in the preceding example, and if the next word can belong to that lexical category, then you can update the state by removing the first symbol and updating the position counter. In this case, since the word *dogs* is listed as an N in the lexicon, the next parser state would be

((VP) 3)

which means it needs to find a VP starting at position 3. If the first symbol is a nonterminal, like VP, then it is rewritten using a rule from the grammar. For example, using rule 4 in the above Grammar, the new state would be

((V) 3)

which means it needs to find a V starting at position 3. On the other hand, using rule *5*, the new state would be

((V NP) 3)

A parsing algorithm that is guaranteed to find a parse if there is one must systematically explore every possible new state. One simple technique for this is called backtracking. Using this approach, rather than generating a single new state from the state ((VP) 3), you generate all possible new states. One of these is picked to be the next state and the rest are saved as backup states. If you ever reach a situation where the current state cannot lead to a solution, you simply pick a new current state from the list of backup states. Here is the algorithm in a little more detail.

A Simple Top-Down Parsing Algorithm

The algorithm manipulates a list of possible states, called the possibilities list. The first element of this list is the current state, which consists of a symbol list - and a word position In the sentence, and the remaining elements of the search state are the backup states, each indicating an alternate symbol-list—word-position pair. For example, the possibilities list

(((N) 2) ((NAME) 1) ((ADJ N) 1))

indicates that the current state consists of the symbol list (N) at position 2, and that there are two possible backup states: one consisting of the symbol list (NAME) at position 1 and the other consisting of the symbol list (ADJ N) at position 1.

Step	Current State	Backup States	Comment
1	**((S) 1)**		initial position
2	**((NP VP) 1)**		rewriting S by rule I
3	**((ART N VP) 1)**		rewriting NP by rules 2 &3
		((ART ADJ N VP) 1)	
4	**((N VP) 2)**		matching ART with *the*
		((ART ADJ N VP) 1)	
5	**((VP) 3)**		matching N with *dogs*
		((ART ADJ N VP) 1)	
6	**((V) 3)**		rewriting VP by rules 5—8
		((V NP) 3)	
		((ART ADJ N VP) 1)	
7			the parse succeeds as V is
			matched to *cried*, leaving
			an empty grammatical symbol
			list with an empty sentence

Figure 3.5 Top-down depth-first parse of 1 The 2 dogs 3 cried 4

The algorithm starts with the initial state ((S) 1) and no backup states.

1. Select the current state: Take the first state off the possibilities list and call it C. If the possibilities list is empty, then the algorithm fails (that is, no successful parse is possible).

2. If C consists of an empty symbol list and the word position is at the end of the sentence, then the algorithm succeeds.

3. Otherwise, generate the next possible states.

   If the first symbol on the symbol list of C is a lexical symbol, and the next word in the sentence can be in that class, then create a new state by removing the first symbol from the symbol list and updating the word position, and add it to the possibilitieslist.

   Otherwise, if the first symbol on the symbol list of C is a non-terminal, generate a new state for each rule in the grammar that can rewrite that nonterminal symbol and add them all to the possibilities list.

Consider an example. Using Grammar 3.4, Figure 3.5 shows a trace of the algorithm on the sentence *The dogs cried.* First, the initial S symbol is rewritten using rule 1 to produce a new current state of ((NP VP) 1) in step 2. The NP is then rewritten in turn, but since there are two possible rules for NP in the grammar, two possible states are generated: The new current state involves (ART N VP) at position 1, whereas the backup state involves (ART ADJ N VP) at position 1. In step 4 a word in category ART is found at position 1 of the sentence, and the new current state becomes (N VP). The backup state generated in step 3 remains untouched. The parse continues in this fashion to step *5,* where two different rules can rewrite VP. The first rule generates the new current state, while the other rule is pushed onto the stack of backup states. The parse completes successfully in step 7, since the current state is empty and all the words in the input sentence have been accounted for.

Consider the same algorithm and grammar operating on the sentence
1 The 2 old 3 man 4 cried 5
In this case assume that the word *old* is ambiguous between an ADJ and an N and that the word *man* is ambiguous between an N and a V (as in the sentence *The sailors man the boats).* Specifically, the lexicon is
the: ART
old: ADJ, N
man: N, V
cried: V

The parse proceeds as follows (see Figure 3.6). The initial S symbol is rewritten by rule 1 to produce the new current state of ((NP VP) 1). The NP is rewritten in turn, giving the new state of ((ART N VP) 1) with a backup state of ((ART ADJ N VP) 1). The parse continues, finding *the* as an ART to produce the state ((N VP) 2) and then *old* as an N to obtain the state ((VP) 3). There are now two ways to rewrite the VP, giving us a current state of ((V) 3) and the backup states of ((V NP) 3) and ((ART ADJ N) 1) from before. The word *man* can be parsed as a V. giving the state (04). Unfortunately, while the symbol list is empty, the word position is not at the end of

the sentence, so no new state can be generated and a backup state must be used. In the next cycle, step 8, ((V NP) 3) is attempted. Again *man* is taken as a V and the new state ((NP) 4) generated. None of the rewrites of NP yield a successful parse. Finally, in step 12, the last backup state, ((ART ADJ N VP) 1), is tried and leads to a successful parse.

Parsing as a Search Procedure

You can think of parsing as a special case of a search problem as defined in Al. In particular, the top-down parser in this section was described in terms of the following generalized search procedure. The possibilities list is initially set to the start state of the parse. Then you repeat the following steps until you have success or failure:

1. Select the first state from the possibilities list (and remove it from the list).
2. Generate the new states by trying every possible option from the selected state (there may be none if we are on a bad path).
3. Add the states generated in step 2 to the possibilities list.

Step	Current State	Backup States	Comment
1.	((S) 1)		
2.	((NP VP) 1)		S rewritten to NP VP
3.	((ART N VP) 1)	((ART ADJ N VP) 1)	NP rewritten producing two new states
4.	((N VP) 2)	((ART ADJ N VP) 1)	
5.	((VP) 3)	((ART ADJ N VP) 1)	the backup state remains
6.	((V) 3)	((V NP) 3) ((ART ADJ N VP) 1)	
7.	(( ) 4)	((V NP) 3) ((ART ADJ N VP) 1)	
8.	((V NP) 3)	((ART ADJ N VP) 1)	the first backup is chosen
9.	((NP) 4)	((ART ADJ N VP) 1)	
10.	((ART N) 4)	((ART ADJ N) 4) ((ART ADJ N VP) 1)	looking for ART at 4 fails
11.	((ART ADJ N) 4)	((ART ADJ N VP) 1)	fails again
12.	((ART ADJ N VP) 1)		now exploring backup state saved in step 3
13.	((ADJ N VP) 2)		
14.	((N VP) 3)		
15.	((VP) 4)		
16.	((V) 4)	((V NP) 4)	
17.	(( ) 5)		success!

Figure 3.6 A top-down parse of $_1$ *The* $_2$ *old* $_3$ *man* $_4$ *cried* $_5$

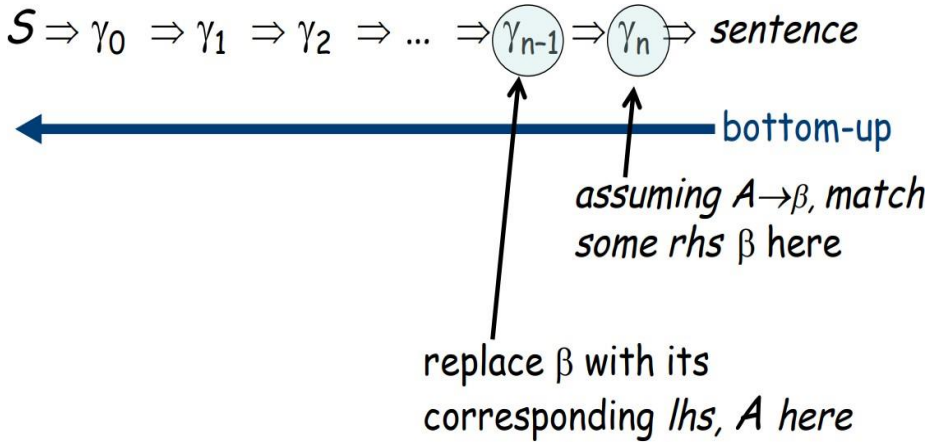
Figure 3.6 A top-down parse of *1 The2 old man 4 cried5*

Bottom-Up Parser

A bottom-up parser builds derivation by working from input sentence back toward the start symbol S. The bottom-up parser is also known as **shift-reduce parser**.

The basic operation in bottom-up parsing is to take a sequence of symbols and match it to the right-hand side of the rules. You could build a bottom-up parser simply by formulating this matching process as a search process. The state would simply consist of a symbol list, starting with the words in the sentence. Successor states could be generated by exploring all possible ways to :

- rewrite a word by its possible lexical categories
- replace a sequence of symbols that matches the right-hand side of a grammar rule by its left-hand side



Bottom-up Chart Parsing Algorithm

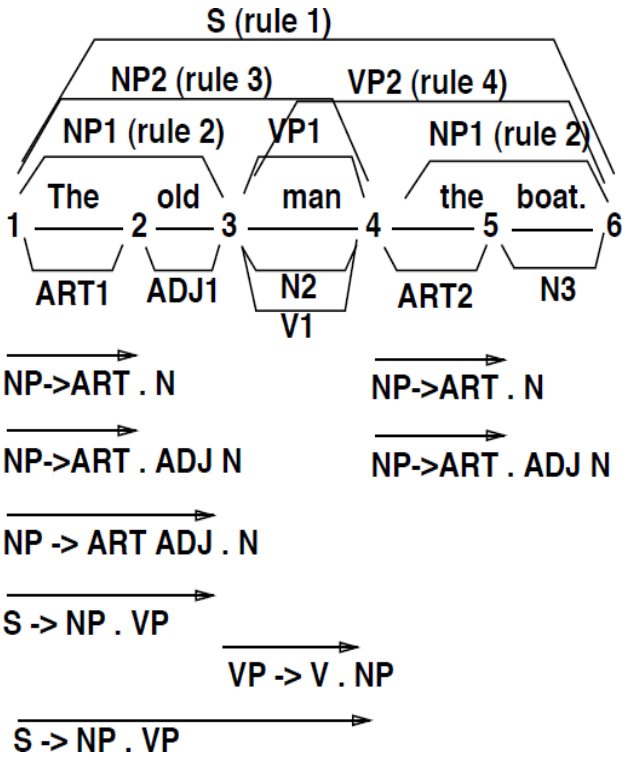
Initialization: For every rule in the grammar of form S -> X1 ... Xk, add an arc labeled S -> o X1 ... Xk using the arc introduction algorithm.

Parsing: Do until there is no input left:

1. If the agenda is empty, look up the interpretations for the next word in the input and add them to the agenda.
2. Select a constituent from the agenda (let's call it constituent C from position p1 to p2).
3. For each rule in the grammar of form **X -> C X1 ... Xn**, add an active arc of form **X -> C o C o X1 .....n** from position p1 to p2.
4. Add C to the chart using the arc extension algorithm above.

Example1.

Grammar and Lexicon
Grammar:
1. S→NP VP
2. NP→ART N
3. NP→ART ADJ N
4. VP →V NP
Lexicon:
the: ART man: N, V
old: ADJ, N boat: N
Sentence: 1 The 2 old 3 man 4 the 5 boat 6



EXAMPLE 2: cats scratch people with claws

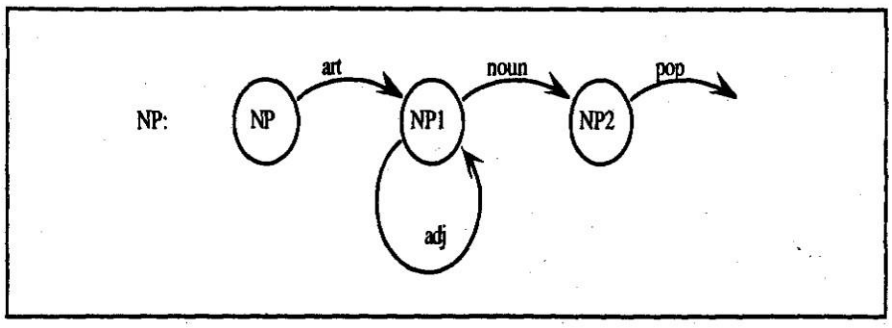
Shift-reduce parsing: one path

	cats scratch people with claws	
cats	*scratch people with claws*	SHIFT
N	*scratch people with claws*	REDUCE
NP	*scratch people with claws*	REDUCE
NP *scratch*	*people with claws*	SHIFT
NP V	*people with claws*	REDUCE
NP V *people*	*with claws*	SHIFT
NP V N	*with claws*	REDUCE
NP V NP	*with claws*	REDUCE
NP V NP *with*	*claws*	SHIFT
NP V NP P	*claws*	REDUCE
NP V NP P *claws*		SHIFT
NP V NP P N		REDUCE
NP V NP P NP		REDUCE
NP V NP PP		REDUCE
NP VP		REDUCE
S		REDUCE

What other search paths are there for parsing this sentence?

Transition Network Grammars

The Transition Network Grammars are based on nodes and edges labeled arcs. One of the nodes is specified as the initial state, or start state. Consider the network named NP in the following Grammar with the initial state labeled NP and each arc labelled with a word category.



Grammar 3.16

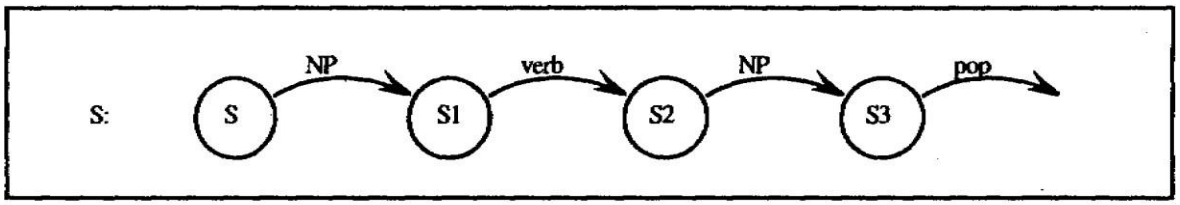
Starting at the initial state, you can traverse an arc if the current word in the sentence is in the category on the arc. If the arc is followed, the current word is updated to the next word. A phrase is a legal NP if there is a path from the node NP to a pop arc (an arc labeled pop) that accounts

for every word in the phrase. This network recognizes the same set of sentences as the following context-free grammar:

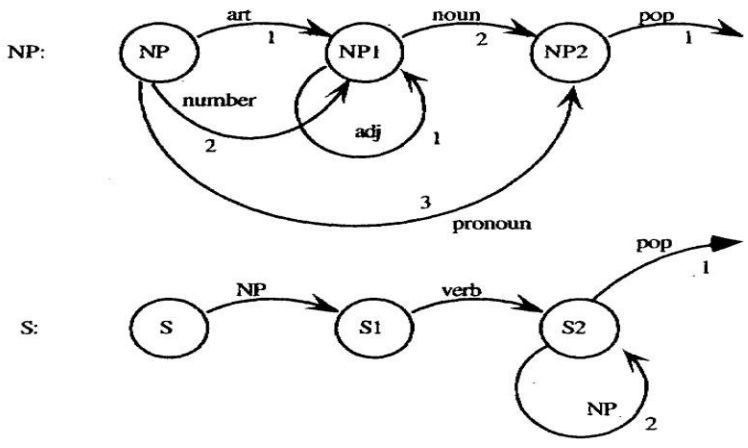
NP -> ART NP1
NP1 -> ADJ NP1
NP1 -> N

Consider parsing the NP *a purple cow* with this network. Starting at the node NP, you can follow the arc labelled art, since the current word is an article— namely, *a.* From node NP1 you can follow the arc labeled adj using the adjective *purple,* and finally, again from NP1, you can follow the arc labeled noun using the noun *cow.* Since you have reached a pop arc, *a purple cow* is a legal NP.

Consider finding a path through the S network for the sentence *The purple cow ate the grass.* Starting at node 5, to follow the arc labeled NP, you need to traverse the NP network. Starting at node NP, traverse the network as before for the input *the purple cow.* Following the pop arc in the NP network, return to the S network and traverse the arc to node S 1. From node S 1 you follow the arc labeled verb using the word *ate.* Finally, the arc labeled NP can be followed if you can traverse the NP network again. This time the remaining input consists of the words *the grass.* You follow the arc labeled art and then the arc labeled noun in the NP network; then take the pop arc from node NP2 and then another pop from node S3. Since you have traversed the network and used all the words in the sentence, *The purple cow ate the grass* is accepted as a legal sentence.



Grammar 3.17



Grammar 3.19

Feature Systems and Augmented Grammars

In natural languages there are often agreement restrictions between words and phrases. For example, the NP "*a men"* is not correct English because the article a indicates a single object while the noun "*men"* indicates a plural object; the noun phrase does not satisfy the number agreement restriction of English. There are many other forms of agreement, including subject-verb agreement, gender agreement for pronouns, restrictions between the head of a phrase and the form of its complement, and so on. To handle such phenomena conveniently, the grammati cal formalism is extended to allow constituents to have features. For example, we might define a feature NUMBER that may take a value of either *s* (for singular) or p (for plural), and we then might write an augmented CFG rule such as

NP -> ART N only when NUMBER1 agrees with NUMBER2

This rule says that a legal noun phrase consists of an article followed by a noun, but only when the number feature of the first word agrees with the number feature of the second. This one rule is equivalent to two CFG rules that would use different terminal symbols for encoding singular and plural forms of all noun phrases, such as

NP-SING -> ART-SING N-SING
NP-PLURAL -> ART-PLURAL N-PLURAL

While the two approaches seem similar in ease-of-use in this one example, consider that all rules in the grammar that use an NP on the right-hand side would now need to be duplicated to include a rule for NP-SING and a rule for NP -PLURAL, effectively doubling the size of the grammar. And handling additional features, such as person agreement, would double the size of the grammar again and again. Using features, the size of the augmented grammar remains the same as the original one yet accounts for agreement constraints.

To accomplish this, a constituent is defined as a feature structure - a mapping from features to values that defines the relevant properties of the constituent. In the examples, feature names in formulas will be written in boldface. For example, a feature structure for a constituent ART1 that represents a particular use of the word a might be written as follows:

ART1: (CAT ART
ROOT a
NUMBER s**)**

This says it is a constituent in the category ART that has as its root the word a and is singular. Usually an abbreviation is used that gives the CAT value more prominence and provides an intuitive tie back to simple context-free grammars. In this abbreviated form, constituent ART1 would be written as

ART1: (ART **ROOT** a **NUMBER** s**)**

Feature structures can be used to represent larger constituents as well. To do this, feature structures themselves can occur as values. Special features based on the integers - 1, 2, 3, and so on - will stand for the first subconstituent, second subconstituent, and so on, as needed. With this, the representation of the NP constituent for the phrase "*a fish*" could be

NP1: (NP NUMBERs
1 (ART **ROOT** a
NUMBER s**)**
2 (N ROOT fish
NUMBER s**))**

Note that this can also be viewed as a representation of a parse tree shown in Figure 4.1, where the subconstituent features 1 and 2 correspond to the subconstituent links in the tree.

The rules in an augmented grammar are stated in terms of feature structures rather than simple categories. Variables are allowed as feature values so that a rule can apply to a wide range of situations. For example, a rule for simple noun phrases would be as follows:

(NP **NUMBER** ?n**) - (**ART **NUMBER** ?n**) (**N **NUMBER** ?n**)**

This says that an NP constituent can consist of two subconstituents, the first being an ART and the second being an N, in which the NUMBER feature in all three constituents is identical. According to this rule, constituent NP1 given previously is a legal constituent. On the other hand, the constituent is not allowed because the NUMBER feature of the N constituent is not identical to the other two NUMBER features.

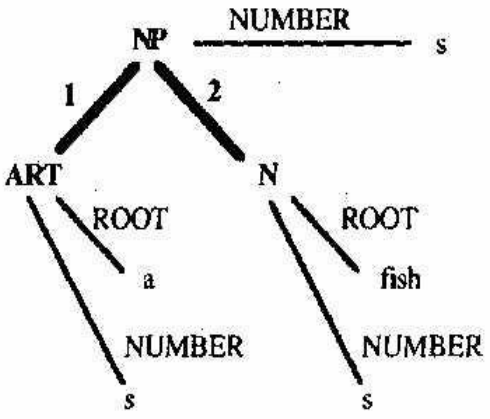


Figure 4.1 Viewing a feature structure as an extended parse tree

* (NP **1** (ART **NUMBER** s)
2 (N **NUMBER** s))
is not allowed by this rule because there is no NUMBER feature in the NP, and the constituent
*(NP **NUMBER** s
1 (ART **NUMBER** s)
2 (N **NUMBER** p))

BOX 4.1 Formalizing Feature Structures

There is an active area of research in the formal properties of feature structures. This work views a feature system as a formal logic. A feature structure is defined as a partial function from features to feature values. For example, the feature structure

ART1: (**CAT** ART
ROOT a
NUMBER *s*)

is treated as an abbreviation of the following statement in FOPC:

$$ART1(CAT) = ART \wedge ART1(ROOT) = a \wedge ART1(NUMBER) = s$$

Feature structures with disjunctive values map to disjunctions. The structure

THE1: (**CAT** ART
ROOT the
NUMBER {s p})

would be represented as

$$THE1(CAT) = ART \wedge THE1(ROOT) = the$$
$$\wedge (THE1(NUMBER) = s \vee THE1(NUMBER) = p)$$

Given this, agreement between feature values can be defined as equality equations.

Morphological Analysis and the Lexicon

Lexical Morphology: Lexical morphology is a specific type of morphology that refers to the **lexemes** in a language. A lexeme is a basic unit of lexical meaning; it can be a single word or a group of words.

Morphological Analysis: Morphological Analysis is the study of lexemes and how they are created. The discipline is particularly interested in **neologisms** (newly created words from existing words(root word)), **derivation**, and **compounding**. In morphological analysis each token will be analysed as follows:

token → lemma(root word) + part of speech + grammatical features

Examples: cats→ cat+N+plur

played →play+V+past

katternas →katt+N+plur+def+gen

Often non-deterministic (more than one solution):

plays →play+N+plur

plays → play+V+3sg

Derivation in Lexical Morphology:

Derivation refers to a way of creating new words by adding **affixes** to the root of a word - this is also known as **affixation**. There are two of affixes: **prefixes and suffixes**.

Prefixes: rewrite (root is write), unfair (root is fair)
Suffixes: wanted, wants, wanting (root is want)

Compounding: Compounding refers to the creation of new words by combining two or more existing words together. Now here are some examples of compounding:

Green + house = Greenhouse
Mother + in + law = Mother-in-law
Motor + bike = Motorbike
Cook + book = Cookbook
Foot + ball = Football

The lexicon must contain information about all the different words that can be used, including all the relevant feature value restrictions. When a word is ambiguous, it may be described by multiple entries in the lexicon, one for each different use.

Most English verbs, for example, use the same set of suffixes to indicate different forms: -s is added for third person singular present tense, -ed for past tense, -ing for the present participle, and so on.

The idea is to store the base form of the verb in the lexicon and use context-free rules to combine verbs with suffixes to derive the other entries. Consider the following rule for present tense verbs:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** 3s) ->
(V **ROOT** ?r **SUBCAT** ?s **VFORM** base) (+S)

where +S is a new lexical category that contains only the suffix morpheme -s. This rule, coupled with the lexicon entry

want:
(V **ROOT** want
SUBCAT {_np_vp:inf _np_vp:inf}
VFORM base)

would produce the following constituent given the input string want -s

want:
(V **ROOT** want

SUBCAT {_np_vp:inf _np_vp:inf}
VFORM pres **AGR** 3s)

Another rule would generate the constituents for the present tense form not in third person singular, which for most verbs is identical to the root form:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** {ls 2s lp 2p 3p})
—> (V **ROOT** ?r **SUBCAT** ?s **VFORM** base)

But this rule needs to be modified in order to avoid generating erroneous interpretations. Currently, it can transform any base form verb into a present tense form, which is clearly wrong for some irregular verbs. For instance, the base form be cannot be used as a present form (for example, *We be at the store). To cover these cases, a feature is introduced to identify irregular forms. Specifically, verbs with the binary feature +IRREGPRES have irregular present tense forms. Now the rule above can be stated correctly:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** (ls 2s lp 2p 3p))
—> (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **IRREG-PRES** -)

The following Grammar 4.5 gives a set of rules for deriving different verb and noun forms using these features. Given a large set of features, the task of writing lexical entries appears very difficult. Most frameworks allow some mechanisms that help alleviate these problems.

Present Tense

1. (V ROOT ?r SUBCAT ?s VFORM pres AGR 3s) →
   (V ROOT ?r SUBCAT ?s VFORM base IRREG-PRES –) +S
2. (V ROOT ?r SUBCAT ?s VFORM pres AGR {1s 2s 1p 2p 3p}) →
   (V ROOT ?r SUBCAT ?s VFORM base IRREG-PRES –)

Past Tense

3. (V ROOT ?r SUBCAT ?s VFORM past AGR {1s 2s 3s 1p 2p 3p}) →
   (V ROOT ?r SUBCAT ?s VFORM base IRREG-PAST –) +ED

Past Participle

4. (V ROOT ?r SUBCAT ?s VFORM pastprt) →
   (V ROOT ?r SUBCAT ?s VFORM base EN-PASTPRT –) +ED
5. (V ROOT ?r SUBCAT ?s VFORM pastprt) →
   (V ROOT ?r SUBCAT ?s VFORM base EN-PASTPRT +) +EN

Present Participle

6. (V ROOT ?r SUBCAT ?s VFORM ing) →
   (V ROOT ?r SUBCAT ?s VFORM base) +ING

Plural Nouns

7. (N ROOT ?r AGR 3p) →
   (N ROOT ?r AGR 3s IRREG-PL –) +S

Grammar 4.5 Some lexical rules for common suffixes on verbs and nouns

Another commonly used technique is to allow the lexicon writing to define clusters of features, and then indicate a cluster with a single symbol rather than listing them all. Figure 4.6 contains a small lexicon. It contains many of the words to be used in the examples that follow. It contains three entries for the word "saw "- as a noun, as a regular verb, and as the irregular past tense form of the verb "see" - as illustrated in the sentences

The saw was broken.

Jack wanted me to saw the board in half.

I saw Jack eat the pizza.

With the lexicon in Figure 4.6 and Grammar 4.5, correct constituents for the following words can be derived: been, being, cries, cried, crying, dogs, saws (two interpretations), sawed, sawing, seen, seeing, seeds, wants, wanting, and wanted. For example, the word cries would be transformed into the sequence cry +s, and then rule 1 would produce the present tense entry from the base form in the lexicon.

```
a:      (CAT ART
         ROOT A1
         AGR 3s)
be:     (CAT V
         ROOT BE1
         VFORM base
         IRREG-PRES +
         IRREG-PAST +
         SUBCAT (_adjp _np))
cry:    (CAT V
         ROOT CRY1
         VFORM base
         SUBCAT _none)
dog:    (CAT N
         ROOT DOG1
         AGR 3s)
fish:   (CAT N
         ROOT FISH1
         AGR (3s 3p)
         IRREG-PL +)
happy:  (CAT ADJ
         SUBCAT _vp:inf)
he:     (CAT PRO
         ROOT HE1
         AGR 3s)
is:     (CAT V
         ROOT BE1
         VFORM pres
         SUBCAT (_adjp _np)
         AGR 3s)
Jack:   (CAT NAME
         AGR 3s)
man:    (CAT N1
         ROOT MAN1
         AGR 3s)
men:    (CAT N
         ROOT MAN1
         AGR 3p)
saw:    (CAT N
         ROOT SAW1
         AGR 3s)
saw:    (CAT V
         ROOT SAW2
         VFORM base
         SUBCAT _np)
saw:    (CAT V
         ROOT SEE1
         VFORM past
         SUBCAT _np)
see:    (CAT V
         ROOT SEE1
         VFORM base
         SUBCAT _np
         IRREG-PAST +
         EN-PASTPRT +)
seed:   (CAT N
         ROOT SEED1
         AGR 3s)
the:    (CAT ART
         ROOT THE1
         AGR (3s 3p))
to:     (CAT TO)
want:   (CAT V
         ROOT WANT1
         VFORM base
         SUBCAT (_np _vp:inf _np_vp:inf))
was:    (CAT V
         ROOT BE1
         VFORM past
         AGR (1s 3s)
         SUBCAT (_adjp _np))
were:   (CAT V
         ROOT BE
         VFORM past
         AGR (2s 1p 2p 3p)
         SUBCAT (_adjp _np))
```

Figure 4.6 A lexicon

Parsing with Features

The parsing algorithms are used for context-free grammars and also extended to handle augmented context-free grammars. This involves generalizing the algorithm for matching rules to constituents. For instance, the chart-parsing algorithms can be used an operation for extending active arcs with a new constituent. A constituent X could extend an arc of the form

C -> C1 ... Ci o X ... Cn

to produce a new arc of the form

C -> C1 ... Ci X o ... Cn

A similar operation can be used for grammars with features, but the parser may have to instantiate variables in the original arc before it can be extended by X. The key to defining this matching operation precisely is to remember the definition of grammar rules with features. A rule such as:

1. (NP **AGR** ?a) -> o (**ART AGR** ?a) (N **AGR** ?a)

says that an NP can be constructed out of an ART and an N if all three agree on the AGR feature. It does not place any restrictions on any other features that the NP, ART, or N may have. Thus, when matching constituents against this rule, the only thing that matters is the AGR feature. All other features in the constituent can be ignored. For instance, consider extending arc 1 with the constituent

2. (**ART ROOT** A **AGR** 3s)

To make arc 1 applicable, the variable ?a must be instantiated to 3s, producing

3. (NP **AGR** 3s) -> o (**ART AGR** 3s) (N **AGR** 3s)

This arc can now be extended because every feature in the rule is in constituent 2:

4. (NP **AGR** 3s) -> (**ART AGR** 3s) o (N **AGR** 3s)

Now, consider extending this arc with the constituent for the word dog:

5. (N **ROOT** DOG1 **AGR** 3s)

This can be done because the AGR features agree. This completes the arc

6. (NP **AGR** 3s) —> (**ART AGR** 3s) (N **AGR** 3s)

This means the parser has found a constituent of the form (NP AGR 3s).

This algorithm can be specified more precisely as follows: Given an arc A, where the constituent following the dot is called NEXT, and a new constituent X, which is being used to extend the arc,

(a.) Find an instantiation of the variables such that all the features specified in NEXT are found in X.

(b.) Create a new arc A', which is a copy of A except for the instantiations of the variables determined in step (a).

(c.) Update A' as usual in a chart parser.

For instance, let A be arc 1, and X be the ART constituent 2. Then NEXT will be (ART AGR ?a). In step a, NEXT is matched against X, and you find that ?a must be instantiated to 3s. In step b, a new copy of A is made, which is shown as arc 3. In step c, the arc is updated to produce the new arc shown as arc 4.

Figure 4.10 contains the final chart produced from parsing the sentence He wants to cry using Grammar 4.8 & Grammar 4.7.

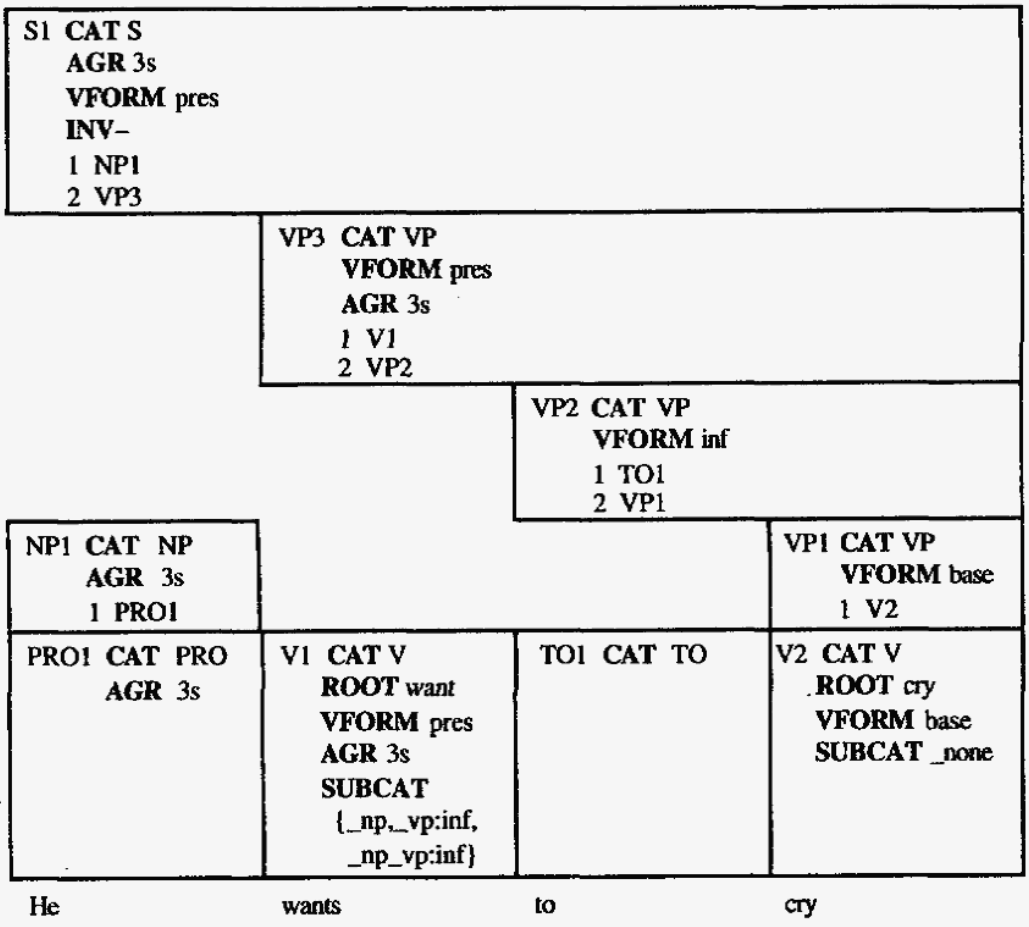


Figure 4.10 The chart for *He wants to cry.*

1. (S **INV** – **VFORM** ?v{pres past} **AGR** ?a) →
   (NP **AGR** ?a) (*VP* ***VFORM*** *?v* [*pres past*] ***AGR*** *?a*)
2. (NP **AGR** ?a) → (ART **AGR** ?a) (*N* ***AGR*** *?a*)
3. (NP **AGR** ?a) → (*PRO* ***AGR*** *?a*)
4. (VP **AGR** ?a **VFORM** ?v) → (*V* ***SUBCAT*** *_none* ***AGR*** *?a* ***VFORM*** *?v*)
5. (VP **AGR** ?a **VFORM** ?v) → (*V* ***SUBCAT*** *_np* ***AGR*** *?a* ***VFORM*** *?v*) NP
6. (VP **AGR** ?a **VFORM** ?v) →
   (*V* ***SUBCAT*** *_vp:inf* ***AGR*** *?a* ***VFORM*** *?v*) (VP **VFORM** inf)
7. (VP **AGR** ?a **VFORM** ?v) →
   (*V* ***SUBCAT*** *_np_vp:inf* ***AGR*** *?a* ***VFORM*** *?v*) NP (VP **VFORM** inf)
8. (VP **AGR** ?a **VFORM** ?v) →
   (*V* ***SUBCAT*** *_adjp* ***AGR*** *?a* ***VFORM*** *?v*) ADJP
9. (VP **SUBCAT** inf **AGR** ?a **VFORM** inf) →
   (*TO* ***AGR*** *?a* ***VFORM*** *inf*) (VP **VFORM** base)
10. ADJP → *ADJ*
11. ADJP → (*ADJ* ***SUBCAT*** *_inf*) (VP **VFORM** inf)

Grammar 4.8 The expanded grammar showing all features

1. S[–inv] → (NP **AGR** ?a) (*VP* [{*pres past*}] ***AGR*** *?a*)
2. NP → (ART **AGR** ?a) (*N* ***AGR*** *?a*)
3. NP → *PRO*
4. VP → *V*[*_none*]
5. VP → *V*[*_np*] NP
6. VP → *V*[*_vp:inf*] VP[inf]
7. VP → *V*[*_np_vp:inf*] NP VP[inf]
8. VP → *V*[*_adjp*] ADJP
9. VP[inf] → *TO* VP[base]
10. ADJP → *ADJ*
11. ADJP → *ADJ*[*_vp:inf*] VP[inf]

Head features for S, VP: **VFORM**, **AGR**
Head features for NP: **AGR**

Grammar 4.7 A simple grammar in abbreviated form

Augmented Transition Networks

The ATN (augmented transition network) is produced by adding new features to a recursive transition network. Features in an ATN are traditionally called registers. Constituent structures are created by allowing each network to have a set of registers. Each time a new network is pushed, a new set of registers is created. As the network is traversed, these registers are set to values by actions associated with each arc. When the network is popped, the registers are assembled to form a constituent structure, with the CAT slot being the network name.

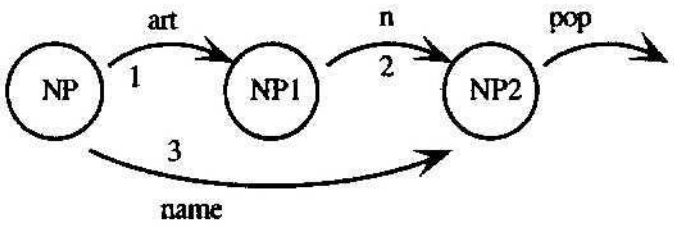
Consider Grammar 4.11 is a simple NP network. The actions are listed in the table below the network. ATNs use a special mechanism to extract the result of following an arc. When a lexical arc, such as arc 1, is followed, the constituent built from the word in the input is put into a special variable named "*". The action

DET := *

then assigns this constituent to the DET register. The second action on this arc,

AGR := AGR*

assigns the AGR register of the network to the value of the AGR register of the new word (the constituent in "*"). Agreement checks are specified in the tests. A test is an expression that succeeds if it returns a nonempty value and fails if it returns the empty set or nil.

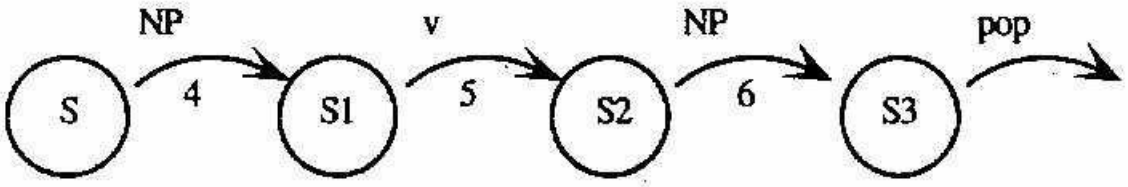


Arc	Test	Actions
1	none	DET := * AGR := AGR*
2	AGR ∩ AGR*	HEAD := * AGR := AGR ∩ AGR*
3	none	NAME := * AGR := AGR*

Grammar 4.11 A simple NP network

If a test fails, its arc is not traversed. The test on arc 2 indicates that the arc can be followed only if the AGR feature of the network has a non-null intersection with the AGR register of the new word (the noun constituent in "*").

Features on push arcs are treated similarly. The constituent built by traversing the NP network is returned as the value "*". Thus in Grammar 4.12, the action on the arc from S to S1,
SUBJ := *
would assign the constituent returned by the NP network to the register SUBJ. The test on arc 2 will succeed only if the AGR register of the constituent in the SUBJ register has a non-null intersection with the AGR register of the new constituent (the verb). This test enforces subject-verb agreement.



Arc	Test	Actions
4	none	SUBJ := *
5	$AGR_{SUBJ} \cap AGR_*$	MAIN-V := * $AGR := AGR_{SUBJ} \cap AGR_*$
6	none	OBJ := *

Grammar 4.12 A simple S network

ATNs Tracing Example:

The following Figure 4.13 Trace tests and actions used with "1 The 2 dog 3 saw 4 Jack 5" With the lexicon in Section 4.3, the ATN accepts the following sentences:
The dog cried.
The dogs saw Jack.
Jack saw the dogs.
Consider an example. A trace of a parse of the sentence "The dog saw Jack" is shown in Figure 4.13. It indicates the current node in the network, the current

Trace of S Network

Step	Node	Position	Arc Followed	Registers Set
1.	S	1	arc 4 succeeds (for recursive call see trace below)	**SUBJ** ← (NP **DET** the **HEAD** dog **AGR** 3s)
5.	S1	3	arc 5 (checks if 3p ∩ 3p)	**MAIN-V** ← saw **AGR** ← 3p
6.	S2	4	arc 6 (for recursive call trace, see below)	**OBJ** ← (NP **NAME** Jack **AGR** 3s)
9.	S3	5	pop arc succeeds	returns (S **SUBJ** (NP **DET** the **HEAD** dog **AGR** 3s) **MAIN-V** saw **AGR** 3p **OBJ** (NP **NAME** Jack **AGR** 3s))

Trace of First NP Call: Arc 4

Step	Node	Position	Arc Followed	Registers Set
2.	NP	1	1	**DET** ← the **AGR** ← {3s 3p}
3.	NP1	2	2 (checks if {3s 3p} ∩ 3p)	**HEAD** ← dog
4.	NP2	3	pop	returns (NP **DET** the **HEAD** dog **AGR** 3s)

Trace of Second NP Call: Arc 6

Step	Node	Position	Arc Followed	Registers Set
7.	NP	4	3	**NAME** ← John **AGR** ← 3s
8.	NP2	5	pop	returns (NP **NAME** John **AGR** 3s)

Figure 4.13 Trace tests and actions used with $_1$ *The* $_2$ *dog* $_3$ *saw* $_4$ *Jack* $_5$

Bayes' Theorem:

Bayes theorem is also known as the Bayes Rule or Bayes Law. It is used to determine the conditional probability of event A when event B has already happened. The general statement of Bayes' theorem is "The conditional probability of an event A, given the occurrence of another event B, is equal to the product of the event of B, given A and the probability of A divided by the probability of event B." i.e.

$$P(A|B) = P(B|A)P(A) / P(B)$$

where,
***P(A)** and **P(B)** are the probabilities of events A and B*
***P(A|B)** is the probability of event A when event B happens*
***P(B|A)** is the probability of event B when A happens*

Bayes Theorem Statement

Bayes' Theorem for n set of events is defined as,

Let $E_1, E_2, \ldots, E_n$ be a set of events associated with the sample space S, in which all the events $E_1, E_2, \ldots, E_n$ have a non-zero probability of occurrence. All the events $E_1, E_2, \ldots, E$ form a partition of S. Let A be an event from space S for which we have to find probability, then according to Bayes' theorem,

$$P(E_i|A) = P(E_i)P(A|E_i) / \sum P(E_k)P(A|E_k)$$
$$\text{for } k = 1, 2, 3, \ldots, n$$

Terms Related to Bayes Theorem

As we have studied about Bayes theorem in detail, let us understand the meanings of a few terms related to the concept which have been used in the Bayes theorem formula and derivation:

Conditional Probability

The probability of an event A based on the occurrence of another event B is termed conditional Probability. It is denoted as **P(A|B)** and represents the probability of A when event B has already happened.

Joint Probability

When the probability of two more events occurring together and at the same time is measured it is marked as Joint Probability. For two events A and B, it is denoted by joint probability is denoted as, **P(A∩B).**

Random Variables

Real-valued variables whose possible values are determined by random experiments are called random variables. The probability of finding such variables is the experimental probability.

Shannon's Spelling Game

Competent spellers are good at recognizing **common spelling patterns**. This enables them to predict how any sound might be spelt because they know that there are only a limited number of options.

For example, if they hear an "o" sound, as in hope, they will consider -oa-; -oe-; or -o- followed, one consonant later, by the magic e

boat toe cope

Shannon's game helps to develop this kind of awareness. It's similar to Hangman except that the letters have to be guessed **in sequence**.

- Start by writing the first letter of a word.
- Then put down dashes to represent the other letters.
- Allow ten guesses for the next letter. If there is no correct guess, put the letter in and go on to the next.
- Continue until the whole word is completed.
- So, for example:
- q.....
  qu....
  que
  ques----
  quest---
  questi--
  questio-
  question

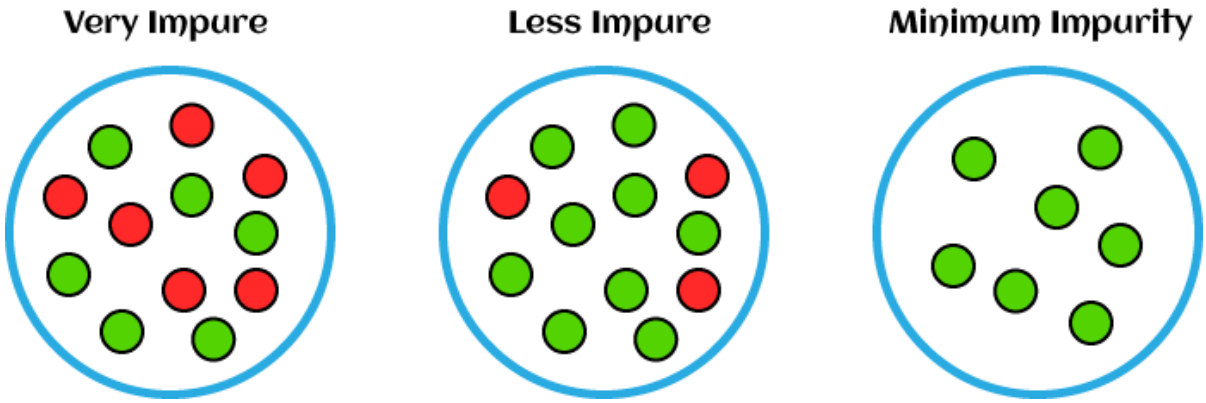
Sometimes it helps to have the alphabet written out in front of the players.

As players become more competent they are able to succeed with far fewer guesses. The game provides an ideal basis for parents to discuss the possible choices at any particular stage.

(The above is based on a page from my website Spelling it Right at www.spellitright.talktalk.net where you will find many other spelling topics)

Entropy and Cross Entropy

Entropy is defined as the randomness or measuring the disorder of the information being processed in Machine Learning. Further, in other words, we can say that **entropy is the machine learning metric that measures the unpredictability or impurity in the system**.



Entropy is a slippery concept in physics, but is quite straightforward in information theory. Suppose you have a process (like a language L that generates words). At each step in the process, there is some probability p that the thing that happened (the event) was going to happen. The amount of **surprisal** is $-\log(p)$ where the logarithm is taken in any base you want (equivalent to changing units). Low probability events have high surprisal. Events that were certain to happen (p=1) have 0 surprisals. Events that are impossible (p=0) have infinity surprisal.

The entropy is the expected value of the surprisal across all possible events indexed by i:

$$H(p) = -\sum_i p_i \log p_i$$

Entropy of a probability distribution p

So, the entropy is the average amount of surprise when something happens.

Entropy always lies between 0 and 1, however depending on the number of classes in the dataset, it can be greater than 1.

Entropy in base 2 is also optimal number of bits it takes to store the information about what happened, by Claude Shannon's source coding theorem. For example if I told you that a full-length tweet of 280 characters had an entropy of 1 bit per character, that means that, by the laws of mathematics, no matter what Twitter does, they will always have to have 280 bits (35 bytes) of storage for that tweet in their database.

In the context of our language model, we'll have to make one tweak. Given that we are interested in sentences *s* (sequences of events) of length *n*, we'll define the entropy rate per word (event) as:

$$H_n(L) = -\frac{1}{n}\sum_{s\in L} L(s)\log L(s)$$

where the sum is over all sentences of length *n* and L(s) is the probability of the sentence. Finally, a technical point: we want to define the entropy of the language L (or language model M) regardless of sentence length *n*. So finally we define

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\sum_{s\in L} L(s)\log L(s)$$

Final definition of entropy for a language (model)

The Shannon-McMillan-Breiman Theorem

Under anodyne assumptions[3] the entropy simplifies even further. The essential insight is that, if we take a long enough string of text, each sentence occurs in proportion to its probability anyways. So there is no need to sum over possible sentences. We get:

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\log L(s)$$

Simplification of entropy with the Shannon-McMillan-Breiman Theorem

This tells us that we can just take a large (n is big) text instead of trying to sample from diverse texts.

Cross-Entropy:

Suppose we mistakenly think that our language model M is correct. Then we observe text generated by the actual language L without realizing it. The *cross-entropy* H(L,M) is what we measure the entropy to be

$$H(L,M) = \lim_{n\to\infty} -\frac{1}{n}\sum_{s\in L} L(s)\log M(s)$$

$$= \lim_{n\to\infty} -\frac{1}{n}\log M(s)$$

Cross entropy for our language model M

Where the second line again applies the Shannon-McMillan-Breiman theorem.
Crucially, this tells us we can estimate the cross-entropy H(L,M) by just measuring log M(s) for a random sample of sentences (the first line) or a sufficiently large chunk of text (the second line).

The Cross-Entropy is Bounded by the True Entropy of the Language

The cross-entropy has a nice property that **H(L)** ≤ **H(L,M)**. Omitting the limit and the normalization 1/n in the proof:

$$H(L) = -\sum L(s)\log\left[L(s)\right]$$

$$= -\sum L(s)\log\left[\frac{L(s)}{M(s)}M(s)\right]$$

$$= -\left(\sum L(s)\log M(s)\right) - \left(\sum L(s)\log\frac{L(s)}{M(s)}\right)$$

$$= H(L,M) - D_{\mathrm{KL}}(L||M)$$

In the third line, the first term is just the cross-entropy (remember the limits and 1/n terms are implicit). The second term is the Kullback-Leibler divergence (or KL-divergence). By Gibbs' inequality the KL-divergence is non-negative and is 0 only if the models L and M are the same. The KL-divergence is sort of like a distance measure.