Natural Language Processing (NLP)

"Natural" languages are human languages, such as English, German, or Chinese.

- Understanding text (in machine-readable form).

```
What customers ordered widgets in May?
```

- Understanding continuous speech: perception as well as language understanding.
- Language generation (written or spoken).
- Machine translation, e.g., German to English:[1]

```
Vor dem Headerfeld befindet sich eine
Praeambel von 42 Byte Laenge fuer den
Ausgleich aller Toleranzen.
-->
A preamble of 42 byte length for the
adjustment of all tolerances is found
in front of the header field.
```

[1] METAL system, University of Texas Linguistics Research Center.

Why Study Natural Language?

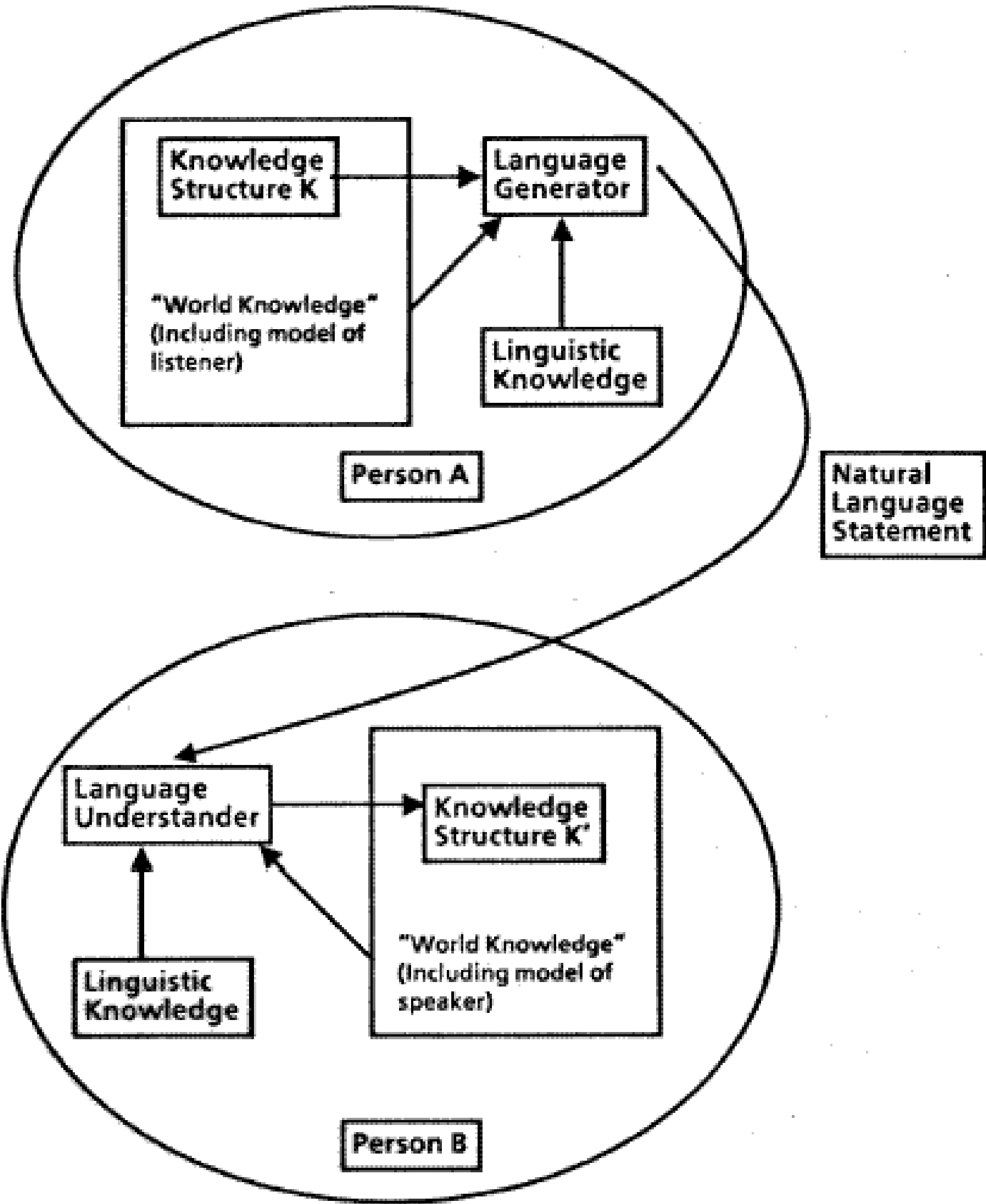
Theoretical:

- Understand how language is structured:
  the right way to do linguistics.
- Understand the mental mechanisms necessary to support language use, e.g. memory:
  language as a window on the mind.

Practical:

- Easier communication with computers for humans:
  - Talking is easier than typing
  - Compact communication of complex concepts
- Machine translation
- Someday intelligent computers may use natural language to talk to each other!

Model of Natural Language Communication



Minimality of Natural Language

William Woods postulated that natural language evolved because humans needed to *communicate complex concepts over a bandwidth-limited serial channel*, i.e. speech.

All of our communication methods are serial:

- a small number of basic symbols (characters, phonemes)
- basic symbols are combined into words
- words are combined into phrases and sentences.

Claude Shannon's *information theory* deals with transmission of information with the smallest possible number of bits. Likewise, natural language is strongly biased toward minimality:

- Never say something the listener already knows.
- Omit things that can be inferred.
- Eliminate redundancy.
- Shorten!

Zipf's Law

Zipf's Law says that *frequently used words are short.* This is true across all human languages.

More formally, $length \propto -log(frequency)$.

If a word isn't short, people who use it frequently will shorten it:

facsimile transmission	fax
latissimus dorsae	lat
Mediterranean	Med
robot	bot

Areas of Natural Language

The study of language has traditionally been divided into several broad areas:

- **Syntax:** The rules by which words can be put together to form legal sentences.
- **Semantics:** Study of the ways statements in the language denote *meanings*.
- **Pragmatics:** Knowledge about the world and the social context of language use.

```
Q: Do you know the time?
A: Yes.
```

Computer Language Understanding

In general, natural language processing involves a *translation* from the natural language to some *internal representation* that represents its *meaning*. The internal representation might be predicate calculus, a semantic network, or a frame representation.

There are many problems in making such a translation:

- **Ambiguity:** There may be multiple ways of translating a statement.
  - **Lexical Ambiguity:** most words have multiple meanings.

    ```
    The pitcher broke his arm.

    The pitcher broke.
    ```

  - **Grammatical Ambiguity:** Different ways of parsing (assigning structure to) a given sentence.

    ```
    One morning I shot an elephant
      in my pajamas.
    How he got in my pajamas
      I don't know.
    ```

Problems in Understanding Language ...

- **Incompleteness:** The statement is usually only the bare outline of the message. The missing parts must be filled in.

  ```
  I was late for work today.
  My car wouldn't start.
  The battery was dead.
  ```

- **Anaphora:**[2] Words that refer to others.

  ```
  John loaned Bill his bike.
  ```

- **Metonymy:** Using a word associated with the intended concept.

  ```
  The White House denied the report.
  ```

- **Semantics:** Understanding what was meant from what was said.
  - Only *differences* from assumed knowledge are stated explicitly.
  - Reasoning from general knowledge about the world is required for correct understanding.
  - A *vast* amount of world knowledge is needed.

[2] The singular is *anaphor*.

Morphology

Morphology is the study of word forms. A program called a *morphological analyzer* will convert words to root forms and affixes (prefixes and suffixes); the root forms can then be looked up in the lexicon.

For English, a fairly simple suffix-stripping algorithm plus a small list of irregular forms will suffice.[3]

```
running   -->   run + ing

went      -->   go  + ed
```

If the lexicon needed for an application is small, all word forms can be stored together with the root form and affixes. For larger lexicons, a morphological analyzer would be more efficient. In our discussions of syntax, we will assume that morphological analysis has already been done.

[3]Winograd, T., in *Understanding Natural Language*, Academic Press, 1972, presents a simple algorithm for suffix stripping. A thorough treatment can be found in Slocum, J., "An English Affix Analyzer with Intermediate Dictionary Lookup", Technical Report LRC-81-01, Linguistics Research Center, University of Texas at Austin, 1981.

Lexicon

The lexicon contains "definitions" of words in a machine-usable form. A lexicon entry may contain:

- The root word spelling
- Parts of speech (noun, verb, etc.)
- Semantic markers, e.g., `animate`, `human`, `concrete`, `countable`.
- Case frames that describe how the word is related to other parts of the sentence (especially for verbs).
- Related words or phrases. For example, *United States of America* should usually be treated as a single term rather than a noun phrase.

Modern language processing systems put a great deal of information in the lexicon; the lexicon entry for a single word may be several pages of information.

Lexical Features

These features are the basis of lexical coding.[4]

```
philosopher +N, +common, +anim, +human, +concrete,
                +count
honesty     +N, +common, -concrete, -count,
idea        +N, +common, -concrete, +count
Sebastian   +N, -common, +human, +masc, +count
slime       +N, +common, +concrete, -anim, -count
kick   +VB, +V, +action, +one-trans,
own    +VB, +V, -action, +one-trans,
honest +VB, -V, +action
tipsy  +VB, -V, -action
```

   I told her to kick the ball
* I told her to own the house
* I told her to be tipsy
   The philosopher who ate
   The idea which influenced me
* The philosopher which ate
* The idea who influenced me

[4] slide by Robert F. Simmons.

Size of Lexicon

Although a full lexicon would be large, it would not be terribly large by today's standards:

- Vocabulary of average college graduate: 50,000 words.
- Oxford English Dictionary: 300,000 words.
- Japanese standard set: 2,000 Kanji.
- Basic English: about 1,000 words.

Each word might have ten or so sense meanings on average. (Prepositions have about 100; the word "set" has the most in the Oxford English Dictionary – over 200.)

These numbers indicate that a lexicon is not large compared to today's memory sizes.

Statistical Natural Language Processing

Statistical techniques can help remove much of the ambiguity in natural language.

A *type* is a word form, while a *token* is each occurrence of a word type. *N-grams* are sequences of N words: *unigrams*, *bigrams*, *trigrams*, etc. Statistics on the occurrences of n-grams can be gathered from text *corpora*.[5]

Unigrams give the frequencies of occurrence of words. Bigrams begin to take context into account. Trigrams are better, but it is harder to get statistics on larger groups.

N-gram approximations to Shakespeare:[6]

1. Every enter now severally so, let
2. What means, sir. I confess she? then all sorts, he is trim, captain.
3. Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
4. They say all lovers swear more performance than they are wont to keep obliged faith unforfeited!

[5] *corpus* (Latin for *body*) is singular, *corpora* is plural. A corpus is a collection of natural language text, sometimes analyzed and annotated by humans.

[6] D. Jurafsky and J. Martin, *Speech and Language Processing*, Prentice-Hall, 2000.

Part-of-Speech Tagging

N-gram statistics can be used to guess the part-of-speech of words in text. If the part-of-speech of each word can be *tagged* correctly, parsing ambiguity is greatly reduced.

> 'Twas brillig, and the slithy toves
> did gyre and gimble in the wabe.[7]

A *Hidden Markov Model* (HMM) tagger chooses the tag for each word that maximizes: [8]

$$P(word \mid tag) * P(tag \mid previous\ n\ tags)$$

For a bigram tagger, this is approximated as:

$$t_i = argmax_j P(w_i \mid t_j) P(t_j \mid t_{i-1})$$

In practice, trigram taggers are most often used, and a search is made for the best set of tags for the whole sentence; accuracy is about 96%.

[7] from Jabberwocky, by Lewis Carroll.
[8] Jurafsky, *op. cit.*

AI View of Syntax

We need a compact and general way to describe language:

> How can a *finite* grammar and parser describe an *infinite* variety of possible sentences?

Unfortunately, this is not easy to achieve.

> But the English ... having such varieties of incertitudes, changes, and Idioms, it cannot be in the compas of human brain to compile an exact regular Syntaxis thereof.[9]

[9] James Howell, *A New English Grammar, Prescribing as certain Rules as the Language will bear, for Forreners to learn English*, London, 1662.

Grammar

A *grammar* specifies the legal syntax of a language. The kind of grammar most often used in computer language processing is a *context-free grammar*. A grammar specifies a set of *productions*; *non-terminal symbols* (phrase names or parts of speech) are enclosed in angle brackets. Each production specifies how a nonterminal symbol may be replaced by a string of terminal or nonterminal symbols, e.g., a Sentence is composed of a Noun Phrase followed by a Verb Phrase.

```
<s>      -->  <np> <vp>
<np>     -->  <art> <adj> <noun>
<np>     -->  <art> <noun>
<np>     -->  <art> <noun> <pp>
<vp>     -->  <verb> <np>
<vp>     -->  <verb> <np> <pp>
<pp>     -->  <prep> <np>

<art>    -->   a | an | the
<noun>   -->   boy | dog | leg | porch
<adj>    -->   big
<verb>   -->   bit
<prep>   -->   on
```

Language Generation

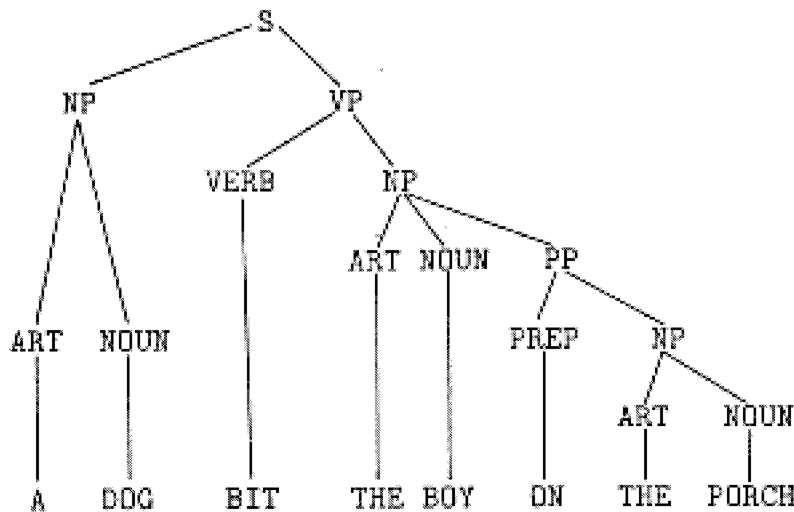
Sentences can be generated from a grammar by the following procedure:

- Start with the sentence symbol, `<S>`.
- Repeat until no nonterminal symbols remain:
  - Choose a nonterminal symbol in the current string.
  - Choose a production that begins with that nonterminal.
  - Replace the nonterminal by the right-hand side of the production.

```
<s>
<np> <vp>
<art> <noun> <vp>
the <noun> <vp>
the dog <vp>
the dog <verb> <np>
the dog <verb> <art> <noun>
the dog <verb> the <noun>
the dog bit the <noun>
the dog bit the boy
```

Parsing

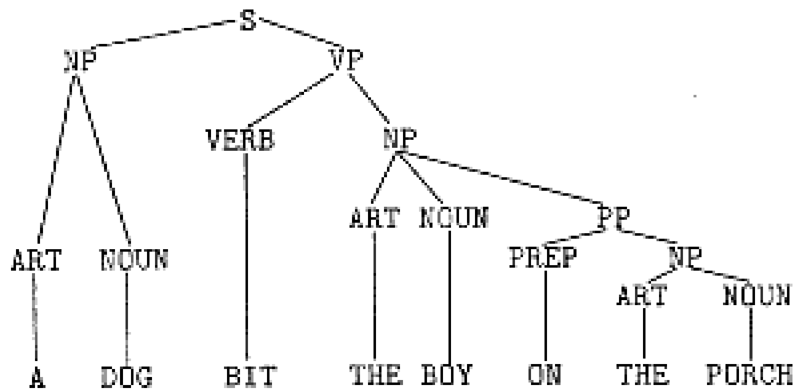
Parsing is the inverse of generation: the *assignment of structure* to a linear string of words according to a grammar; this is much like the "diagramming" of a sentence taught in grammar school.

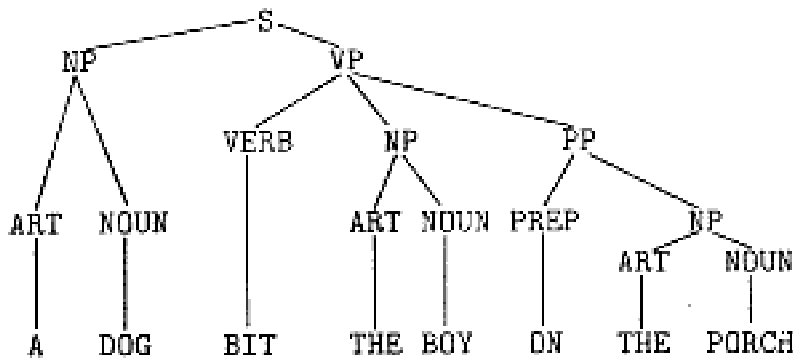


Parts of the *parse tree* can then be related to object symbols in the computer's memory.

Ambiguity

Unfortunately, there may be many ways to assign structure to a sentence (e.g., what does a PP modify?):





Definition: A grammar is ambiguous iff there exists some sentence with two distinct parse trees.

Sources of Ambiguity

- **Lexical Ambiguity:**
  Words often have multiple meanings (*homographs*) and often multiple parts of speech.

bit:	verb:	past tense of bite
	noun:	a small amount
		instrument for drilling
		unit of computer memory
		part of bridle in horse's mouth

- **Grammatical Ambiguity:**
  Different ways of parsing (assigning structure to) a given sentence.

  `I saw the man on the hill with the telescope.`

Lexical ambiguity compounds grammatical ambiguity when words can have multiple parts of speech. Words can also be used as other parts of speech than they normally have.

Foreign Languages

It should be kept in mind that much of the study of computer language processing has been done using English.

The techniques used for English do not necessarily work as well for other languages. Some issues:

- Word order is used more in English than in many other languages, which may use case forms instead.

  gloria in excelsis Deo

- Agreement in number and gender are more important in other languages.

  la casa blanca *the white house*
  el caballo blanco *the white horse*

- Familiar, formal, honorific forms of language.

Formal Syntax

There is a great deal of mathematical theory concerning the syntax of languages. This theory is based on the work of Chomsky; grammars for Sanskrit were developed in India much earlier.

Formal syntax has proved to be better at describing artificial languages such as programming languages than at describing natural languages. Nevertheless, it is useful to understand this theory.

A *recursive language* is one that can be recognized by a program; that is, given a string, a program can tell within finite time whether the string is or is not in the language.

A *recursively enumerable language* is one for which all strings in the language can be enumerated by a program. All languages described by phrase structure grammars are R.E., but not all R.E. languages are recursive.

Notation

The following notations are used in describing grammars and languages:

V^*	a string of 0 or more elements from the set V (*Kleene star* or *Kleene closure*)
V^+	1 or more elements from V
$V^?$	0 or 1 elements from V (*i.e.*, optional)
$a \vert b$	either a or b
$< nt >$	a nonterminal symbol or phrase name
ϵ	the empty string

Phrase Structure Grammar

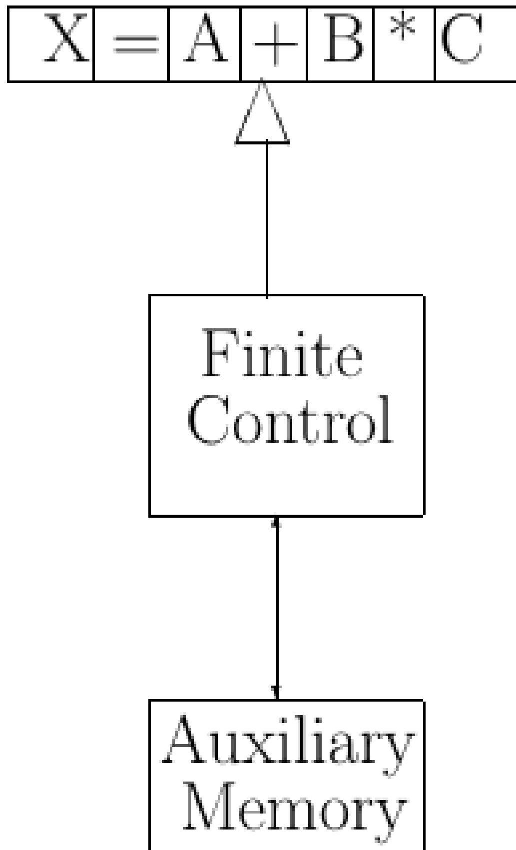
A grammar describes the structure of the sentences of a language in terms of components, or phrases. The mathematical description of phrase structure grammars is due to Chomsky.[10]

Formally, a *Grammar* is a four-tuple $G = (T, N, S, P)$ where:

- **T** is the set of *terminal symbols* or *words* of the language.
- **N** is a set of *nonterminal symbols* or *phrase names* that are used in specifying the grammar. We say $V = T \cup N$ is the *vocabulary* of the grammar.
- **S** is a distinguished element of N called the *start symbol.*
- **P** is a set of *productions*, $P \subseteq V^*NV^* \times V^*$. We write productions in the form $a \rightarrow b$ where a is a string of symbols from V containing at least one nonterminal and b is any string of symbols from V.

[10]See, for example, Aho, A. V. and Ullman, J. D., *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972; Hopcroft, J. E. and Ullman, J. D., *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.

Recognizing Automaton



The *Finite Control* (a program with finite memory) reads symbols from the input tape one at a time, storing things in the *Auxiliary Memory.*

The recognizer answers *Yes* or *No* to the question "Is the input string a member of the language?"

Regular Languages

Productions: $A \rightarrow xB$
$A \rightarrow x$
$A, B \in N$
$x \in T^*$

- Only *one* nonterminal can appear in any derived string, and it must appear at the right end.
- Equivalent to a *deterministic finite automaton* (simple program).
- Parser never has to back up or do search.
- Linear parsing time.
- Used for simplest items (identifiers, numbers, word forms).
- Any *finite* language is regular.
- Any language that can be recognized using finite memory is regular.

Context Free Languages

Productions: $A \to \alpha$
$A \in N$
$\alpha \in V^*$

- Since left-hand-side of each production is a single nonterminal, every derivation is a tree.
- Many good parsers are known. Parsing requires a recursive program, or equivalently, a stack for temporary storage.
- Parsing time is $O(n^3)$.
- Used for language elements that can contain themselves, e.g.,
  - Arithmetic expressions can contain sub-expressions: $A + B * (C + D)$.
  - A noun phrase can contain a prepositional phrase, which contains a noun phrase:
    a girl with a hat on her head.

What Kind of Language is English?

- English is Context Free.[11]
- English is not Context Free.[12]
- English is Regular:
  - English consists of finite strings from a finite vocabulary.
  - English is recognized by people with finite memory.
  - There is no evidence that peoples' parsing time is more then $O(n)$.

A better question to ask is:

> *What is a good way to describe English for computer processing?*

[11] Gazdar, G., "NLs, CFLs, and CF-PSGs", in Sparck Jones, K. and Wilks, Y., Eds., *Automatic Natural Language Processing*, Ellis Horwood Ltd., West Sussex, England, 1983.

[12] Higginbotham, J., "English is Not a Context Free Language", *Linguistic Inquiry 15*, 119-126, 1984.

Parsing

A *parser* is a program that converts a *linear* string of input words into a *structured* representation that shows how the phrases (substructures) are related and shows how the input could have been derived according to the grammar of the language.

Finding the correct parsing of a sentence is an essential step towards extracting its meaning.

Natural languages are harder to parse than programming languages; the parser will often make a mistake and have to fail and back up: parsing is search. There may be hundreds of ambiguous parses, most of which are wrong.

Parsers are generally classified as *top-down* or *bottom-up*, though real parsers have characteristics of both.

There are several well-known context-free parsers:

- Cocke-Kasami-Younger (CKY or CYK) *chart parser*
- Earley algorithm
- Augmented transition network

Top-down Parser

A *top-down* parser begins with the Sentence symbol, `<S>`, expands a production for `<S>`, and so on recursively until words (terminal symbols) are reached. If the string of words matches the input, a parsing has been found.[13]

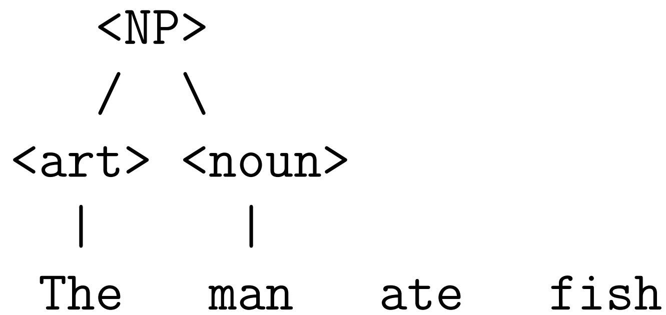
This approach to parsing might seem hopelessly inefficient. However, *top-down filtering*, that is, testing whether the next word in the input string could begin the phrase about to be tried, can prune many failing paths early.

For languages with *keywords*, such as programming languages or natural language applications, top-down parsing can work well. It is easy to program.

[13] See the Language Generation slide earlier in this section.

Bottom-up Parsing

In *bottom-up* parsing, words from the input string are reduced to phrases using grammar productions:



This process continues until a group of phrases can be reduced to `<S>`.

Augmented Transition Networks

An ATN [14] is like a finite state transition network, but is augmented in three ways:

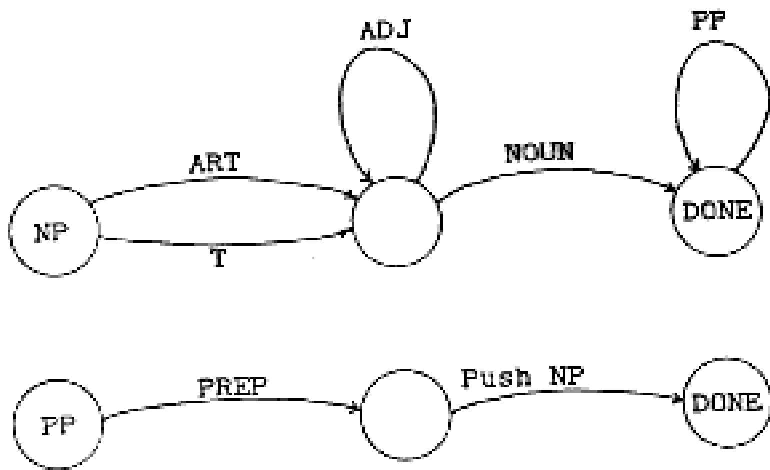
1. **Arbitrary tests** can be added to the arcs. A test must be satisfied for the arc to be traversed. This allows, for example, tests on agreement of a word and its modifier.
2. **Structure-building actions** can be added to the arcs. These actions may save information in *registers* to be used later by the parser, or to build the representation of the meaning of the sentence. Transformations, e.g., active/passive, can also be handled.
3. **Phrase names**, as well as part-of-speech names, may appear on arcs. This allows a grammar to be called as a subroutine.

The combination of these features gives the ATN the power of a Turing Machine, i.e., it can do anything a computer program can do.

[14] Woods, W. A., "Transition Network Grammars for Natural Language Analysis", *Communications of the ACM*, Oct. 1970

Augmented Transition Networks

A grammar can be written in network form. Branches are labeled with parts of speech or phrase names. Actions, such as constructing a database query, can be taken as arcs are traversed.

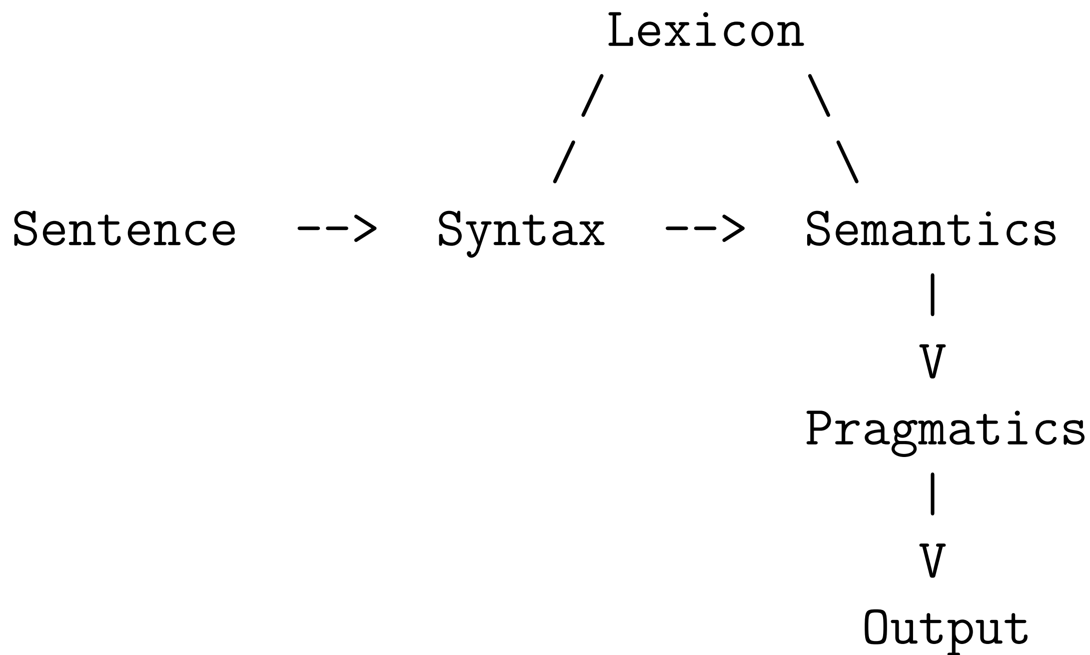


ATN's are more readable than lists of productions.

ATN interpreter and compiler packages exist; one can also write an ATN-like program directly in Lisp.

Separability of Components

An idealized view of natural language processing has the components cleanly separated and sequential:



Unfortunately, such a clean separation doesn't work well in practice.

Problems with Separability

- **Lexicon:**
  - New uses of words.

    You can verb anything. – William Safire
  - Metaphor: `The computer is down.`
- **Syntax:**
  - Ambiguity: *hundreds* of syntactically possible interpretations of ordinary sentences.
  - Agreement:

    `Bill and John love Mary.`
  - Elision: omission of parts of a sentence.

    `He gave John fruit and Mary candy.`
- **Discourse:**
  - The meaning of a sentence depends on context.

Combining Syntax and Semantics

There are several advantages to combining syntactic and semantic processing:

- **Removal of Ambiguity:** It is better to eliminate an incorrect parsing before it is generated, rather than generating all possible interpretations and then removing bad ones.
  - Computer time is saved.
  - Eliminating one bad partial interpretation eliminates many bad total interpretations.
- **Reference:** It is often advantageous to relate the sentence being parsed to the model that is being constructed during the parsing process. "John holds the pole at one end [*of the pole*]."
- **Psychological Plausibility:** People can deal with partial and even ungrammatical language.

  All your base are belong to us.

  This sentence no verb. – D. Hofstadter

How to Combine Syntax & Semantics

- **Grammar and Parser:** no place to include program operations.

  Note that in natural language processing we often want the parsing that is chosen for ambiguous sentences to depend on semantics.

- **Program Alone:** *ad hoc*, likely to be poorly structured.

- **Augmented Transition Network:** best of both worlds.

Natural Language as an AI Problem

Natural language understanding is a classical AI Problem:

- **Minimal Input Data:** the natural language statement does not *contain* the message, but is a minimal specification to allow an intelligent reader to *construct* the message.
- **Knowledge Based:** the interpretation of the message is based in large part on the knowledge that the reader already has.
- **Reference to Context:** the message implicitly refers to a context, including what has been said previously.
- **Local Ambiguity:** many wrong interpretations are superficially consistent with the input.
- **Global Constraints:** there are many different kinds of constraints on interpretation of the input.
- **Capturing the Infinite:** a language understanding system must capture, in finite form, rules sufficient to understand a potentially infinite set of statements.

Reference

Reference is the problem of determining which objects are referred to by phrases.

> *A* pole supports *a* weight at *one* end.

Determiners:

- **Indefinite:** *a*

```
      Make a new object.
```

- **Definite:** *the*, *one*, etc.

```
      Find an existing object;
else, find something closely related
      to an existing object;
else, make a new one.
```

In reading the above sentence, we create a new pole object and a new weight object, but look for an existing *end*: one end of the existing pole.

Referent Identification

Referent identification is the process of identifying the object(s) in the internal model to which a phrase refers.

> *Paul and Henry* carry a *sack* on a pole. If the *load* is 0.5 m from Paul, what force does *each boy* support?

load is not a synonym for *sack*; instead, it describes the role played by the sack in this context.

Unification of *Paul and Henry* with *each boy* conveys new information about the ages of Paul and Henry.

the *left* end ... the *other* end

the *100 lb* boy

the *heavy* end

English

English is a context-free language (more or less).

English has a great deal of ambiguity, compared to programming languages. By restricting the language to an *English subset* for a particular application domain, English I/O can be made quite tractable.

Some users may prefer an English-like interface to a more formal language.

Of course, the best way to process English is in Lisp.

Expression Trees to English [15]

```
(defn op  [x] (first x))
(defn lhs [x] (second x))
(defn rhs [x] (third x))

(defn op->english [op]
  (list 'the
      (second (assocl op '((+ sum)
                           (- difference)
                           (* product)
                           (/ quotient)
                           (sin sine)
                           (cos cosine)))) 'of))

(defn exp->english [x]
  (if (cons? x)                     ; operator?
      (append
        (op->english (op x))
        (append (exp->english (lhs x))
                (if (null? (rest (rest x)))
                    '()             ; unary
                    (cons 'and
                      (exp->english (rhs x))))))
      (list x) ) )                  ; leaf: operand
```

[15] file `expenglish.clj`

Generating English

```
%clojure
>(load-file "cs378/expenglish.clj")

>(exp->english 'x)

(X)

>(exp->english '(+ x y))

(THE SUM OF X AND Y)

>(exp->english '(/ (cos z) (+ x (sin y))))

(THE QUOTIENT OF THE COSINE OF Z AND
     THE SUM OF X AND THE SINE OF Y)
```

Simple Language Processing: ELIZA

Weizenbaum's ELIZA program simulated a psychotherapist; it achieved surprisingly good performance simply by matching the "patient's" input to patterns:

```
Pattern:   (I HAVE BEEN FEELING *)
Response:  (WHY DO YOU THINK YOU
             HAVE BEEN FEELING *)
```

The * matches anything; it is repeated in the answer.

```
Patient: I have been feeling depressed
           today.
Doctor:  Why do you think you have been
           feeling depressed today?
```

Problems:

- Huge number of patterns needed.
- Lack of real understanding:

```
Patient: I just feel like jumping
           off the roof.
Doctor:  Tell me more about the roof.
```

Spectrum of Language Descriptions

ELIZA and a general grammar represent two extremes of the language processing spectrum:

- **ELIZA:**
  Too restricted. A large application, PARRY – an artificial paranoid – was attempted, but failed to get good enough coverage even with 10,000 patterns.
- **General English Grammar:**
  Too ambiguous. Hundreds of interpretations of ordinary sentences.

There is a very useful middle ground: *semantic grammar*.

Semantic Grammar

Semantic grammar lies between ELIZA and a more general English grammar. It uses a grammar in which nonterminal symbols have *meaning* in the domain of application.

```
<S>      -->  WHAT <CUST> ORDERED <PART>
                <MODS>
<CUST>   -->  CUSTOMER | CUSTOMERS <LOC>
<LOC>    -->  IN <CITY>
<CITY>   -->  AUSTIN | SEATTLE | LA
<PART>   -->  WIDGETS | WIDGET BRACKETS
<MODS>   -->  IN <MONTH> | BEFORE <MONTH>
<MONTH>  -->  JANUARY | FEBRUARY | MARCH
```

```
WHAT CUSTOMERS IN AUSTIN ORDERED
  WIDGET BRACKETS IN MARCH
```

Advantages:

- More coverage with fewer patterns than ELIZA.
- No ambiguity due to use of semantic phrases.
- Easy to program.

Semantic Grammar: Extended Pattern Matching

In this approach, the pattern-matching that is allowed is restricted to certain semantic categories. A grammar is used to specify the allowable patterns; this allows the restrictions to be specified easily, while allowing more language coverage and easier extension with fewer specified patterns.

Example:

```
<s> --> what is <ship-property> of <ship>?
<ship-property> --> the <ship-prop> | <ship-prop>
<ship-prop> --> speed | length | draft | beam
<ship> --> <ship-name> | the fastest <ship2>
           | the biggest <ship2> | <ship2>
<ship-name> | Kennedy | Kitty Hawk | Constellation
<ship2> --> <countrys> <ship3> | <ship3>
<ship3> --> <shiptype> <loc> | <shiptype>
<shiptype> --> carrier | submarine | ...
<countrys> --> American | French | British
<loc> --> in the Mediterranean | in the Med | ...
```

"What is the length of the fastest French sub in the Med?"

Example Semantics for a Semantic Grammar

Suppose we want to use the semantic grammar given earlier to access a relational database containing information about ships. For simplicity, let us assume a single `SHIP` relation-as follows:

NAME	TYP	OWN	LAT	LONG	SPD	LNG
Kitty Hawk	CV	US	$10^o00'N$	$50^o27'E$	35	1200
Eclair	SS	France	$20^o00'N$	$05^o30'E$	15	50

Consider the query: *What is the length of the fastest French sub in the Med?*

This query is parsed by the top-level production

```
<S> --> What is <ship-property> of <ship>?
```

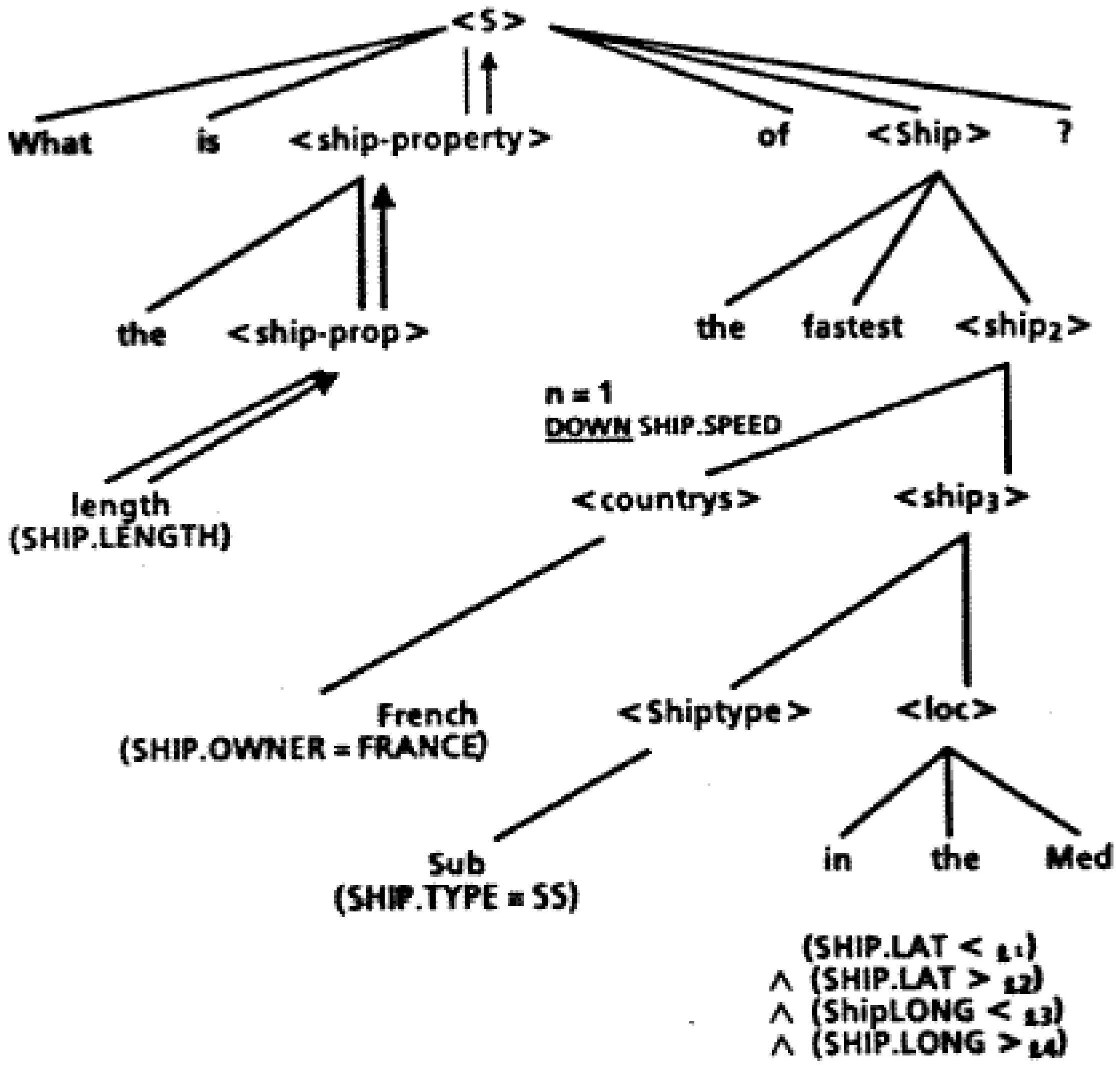
which is conveniently structured in terms of:

1. The data values to be retrieved: `<ship-property>`
2. The data records (tuples) from which to retrieve the data: `<ship>`.

In each case, the values are *additive* and can be synthesized from the parse tree, as shown below.

Compositional Semantics

GET w (l) (SHIP.LENGTH): DOWN SHIP.SPEED $\wedge$ SHIP.OWNER = FRANCE $\wedge$ SHIP.TYPE = SS $\wedge$ SHIP.LAT $< \ell_1 \wedge$ SHIP.LAT $> \ell_2 \wedge$ SHIP.LONG $< \ell_3 \wedge$ SHIP.LONG $> \ell_4$



The semantics of each phrase is propagated up the tree and combined with the semantics of the other descendant nodes at each higher-level node of the tree.

Additional Language Features

Semantic grammar enables additional features that help users:

- Spelling correction:

```
What is the lentgh of Kennedy?
= length
```

  Because we know from the grammar that a `<ship-prop>` is expected, the list of possible ship properties can be used as input to a spelling corrector algorithm to automatically correct the input.

- Sentence fragments:

```
What is the length of Kennedy?

speed
= What is the speed of Kennedy?
```

  If the input can be parsed as a part of the previous parse tree, the rest of the input can be filled in.

Recursive Descent

Recursive Descent is a method of writing a parsing program in which a grammar rule is written as a function.

Given a grammar rule:

```
S -> NP VP
```

we simply make the left-hand-side nonterminal be the name of the function, and write a series of function calls for the right-hand side.

```
(defn s []
  (np)
  (vp) )
```

There could be an infinite loop if there is *left recursion*, i.e. a rule of the form:

```
A -> A ...
```

Parsing English

In most cases, a parser for a programming language never has to back up: if it sees `if`, the input must be an `if` statement or an error.

Parsing English requires that the parser be able to fail, back up, and try something else: if it sees `in`, the input might be `in Austin` or `in April`, which may be handled by different kinds of grammar rules.

Backup means that parsing is a search process, possibly time-consuming. However, since English sentences are usually short, this is not a problem in practice.

An *Augmented Transition Network* (ATN) framework facilitates parsing of English.

ATN Program [16]

- A global variable `atnsent` points to a list of words that remain in the input sentence:
  `(GOOD CHINESE RESTAURANT IN LOS ALTOS)`
- A global variable `atnword` points to the current word: `GOOD`
- `(wordcat` *category* `)` tests whether `atnword` is in the specified category. It can also translate the word, e.g. `(wordcat 'month)` might return `3` if `atnword` is `MARCH`.
- `(nextword)` moves to the next word in the input
- `(saveptr)` saves the current sentence position on a stack, `atnsavesent`.
- `(success)` pops a saved position off the stack.
- `(fail)` restores a saved position from the stack (restoring `atnsent` and `atnword`) and returns `nil`.

[16] file `atn.clj`

Parsing Functions

The parser works by recursive descent, but with the ability to fail and back up and try another path.

```
(defn locfn []
  (saveptr)
  (let [$1 (and (= atnword (quote in)) atnword)]
    (if $1
        (do (nextword)
            (let [$2 (wordcat (quote city))]
              (if $2
                  (do (nextword)
                      (success)
                      (restrict (quote city) $2))
                  (fail))))
        (fail))))
```

The program performs `(saveptr)` on entry and either `(success)` or `(fail)` before leaving.

Grammar Compiler [17]

It is easy to write a grammar compiler that converts a Yacc-like grammar into the equivalent ATN parsing functions. This is especially easy in Lisp since Lisp code and Lisp data are the same thing.

```
(rulecompr '(loc ->  (in (city))
                (restrict 'city $2))
           'locfn)

(defn locfn []
  (saveptr)
  (let [$1 (and (= atnword (quote in)) atnword)]
    (if $1
        (do (nextword)
            (let [$2 (wordcat (quote city))]
              (if $2
                  (do (nextword)
                      (success)
                      (restrict (quote city) $2))
                  (fail))))
        (fail))))
```

[17]file gramcom.clj

Sentence Pointer Handling

```
; initialize for a new sentence
(defn initsent [sent]
  (def atnsent sent)    ; remainder of sentence
  (def atnsavesent '()) ; saved pos for backup
  (setword))

;   set atnword for current position
(defn setword []
  (def atnword (first atnsent))   ; current word
  (def atnnext (rest atnsent))  )

;   move to next word
(defn nextword []
  (def atnsent atnnext) (setword) true)
```

Sentence Pointer Handling ...

```
;   save the current position
(defn saveptr []
  (def atnsavesent
       (cons atnsent atnsavesent)))   ; push

;   pop the stack on success
(defn success []
  (def atnsavesent (rest atnsavesent)) )   ; pop

;   restore position on failure, return nil
(defn fail    []
  (def atnsent (first atnsavesent))
  (def atnsavesent (rest atnsavesent))
  (setword)
  nil)
```

Lexicon Example

```
(def lexicon
 '((a/an     (a an some))
   (i/you    (i you one))
   (get      (get find obtain))
   (quality  ((good 2.5) ))
   (restword (restaurant (restaurants restaurant)))
   (kindfood (american bakery chinese))
   (city     (palo-alto berkeley los-altos))
   (county   (santa-clara))
   (area     (bay-area))
   (street   (el-camino-real))
   ))
```

Note translation to internal form, e.g., `good -> 2.5`

It is easy to include abbreviations, slang, and special terms. These are good because they are usually short (reducing typing), are usually unambiguous, and users like them.

Word Category Testing

```
;   Test if current word is in category
(defn wordcat [category]
  (if (= category 'number)
      (and (number? atnword) atnword)
      (if (= category 'symbol)
          (and (symbol? atnword) atnword)
          (let [catlst (assocl category lexicon)
               wd     (findwd atnword (second catls
            (if (cons? wd)
                (if (empty? (rest wd))
                    (first wd)
                    (second wd))
                wd) ))))
```

The lexicon and category testing can do multiple tasks:

1. Test if a word has a specified part of speech.
2. Translate to internal form, e.g., `March --> 3`.
3. Check for multi-word items, e.g., United States (not implemented).

Database Access

Database access requires two kinds of information:

1. Which records are to be selected. This takes the form of a set of restrictions that selected records must satisfy.

   ```
   (restrict 'field value)
   ```

2. What information is to be retrieved from the selected records.

   ```
   (retrieve 'field)
   ```

The task of the NL access program is to translate the user's question from English into a formal call to an existing database program.

The components of the query are collected as lists in the global variables `restrictions` and `retrievals`.

Database Access

Our example database program takes queries of the form:

```
(querydb <condition> <action>)
```

The `<condition>` is formed by consing **and** onto the `restrictions`, and the `<action>` is formed by consing `list` onto the `retrievals`.

The condition and action are Clojure code using a variable `tuple`: if the condition is true, the action is executed and its result is collected. Both the condition and action can access fields of the current database record using the call:

```
(getdb (quote <fieldname>))
```

Building Database Access

```
; retrievals   = things to get from database
; restrictions = restrictions on the query

;   Main function: ask
(defn ask [sentence]
  ...
  (s)
  ...
   (let [ans (querydb (cons 'and restrictions)
                      (cons 'list retrievals))]
     (if postprocess
         (eval (subst ans '$$ postprocess))
         ans )) ) )

;   make a database access call
(defn retrieve [field]
  (addretrieval (list 'getdb 'tuple (kwote field))

;   add a restriction to the query
(defn restrict [field value]
  (addrestrict
    (list '= (list 'getdb 'tuple (kwote field))
             (kwote value)) ) )
```

Grammar Rules

A grammar rule has the form:

```
(nonterm -> (right-hand side items) semantics)
```

`nonterm` is a nonterminal symbol that is the left-hand side of the production; the rule says that the left-hand side nonterminal can be composed of the sequence of items on the right-hand side.

The allowable items on the right-hand side are:

`word`	exactly the specified word
`(nonterminal)`	like a subroutine call to a sub-grammar.
`(category)`	a word in the category, e.g. `(month)`
`(number)`	any number
`(symbol)`	any symbol
`?`	preceding item is optional (separate ? from a word by a space)

The `semantics` is clojure code to be executed when the grammar rule is satisfied. The right-hand side items are available as variables `$1`, `$2`, `$3`, etc., similar to what is done in Yacc. For example, consider the rule:

```
(loc -> (in (city))   (restrict 'city $2))
```

In this case, `$2` refers to whatever matches the `(city)` part of the grammar rule.

Restaurant Database Grammar

```
(def grammar
'((command  -> (show me) true)
  (command  -> (what is) true)
  (qual     -> ((quality))
                    (restrictb '>= 'rating $1))
  (qualb    -> (rated above (number))
                    (restrictb '>= 'rating $3))
  (resttype -> ((kindfood))
                    (restrict 'foodtype $1))
  (loc      -> (in (city))   (restrict 'city $2))
  (loc      -> (in (county)) (restrict 'county $2))
  (s -> ((command) (a/an)? (qual)? (resttype)?
         (restword) (qualb)? (loc)?)
              (retrieve 'restaurant) )
  (s -> (how many (qual)? (resttype)? food ?
                  (restword) (loc)?)
        (do (retrieve 'restaurant)
              (postpr '(length (quote $$)))) )
```

Notes on Database Grammar

It is good to write grammar rules that cover multiple sentences, using:

- multiple rules for a nonterminal, to handle similar phrases
- the `?` to make the preceding item optional.

My solution for the restaurant assignment only has 5 rules for the top-level nonterminal `s`.

Multiple actions can be combined using `do`:

```
(do (retrieve 'streetno) (retrieve 'street))
```

Questions such as *how many* or *what is the best* require post-processing. The result of the query (restrictions and retrievals) is available as the variable `$$`:

```
(postpr '(length (quote $$)))
```

In this case, `postpr` specifies post-processing, and `length` is the function that is called; this would answer *how many*.

Restaurant Queries

```
% clojure
user=> (load-file "cs378/restaurant.clj")
user=> (load-files)
user=> (gramcom grammar)

user=> (ask '(where can i get ice-cream in berkeley
 ((x2001-flavors-ice-cream-&-yogur)
 (baskin-robbins)
 (double-rainbow)
 (fosters-freeze)
 (marble-twenty-one-ice-cream)
 (sacramento-ice-cream-shop)
 (the-latest-scoop))

user=> (ask '(show me chinese restaurants
              rated above 2.5 in los-altos))
 ((china-valley)
 (grand-china-restaurant)
 (hunan-homes-restaurant)
 (lucky-chinese-restaurant)
 (mandarin-classic) ...)
```

Physics Problems[18]

```
(def lexicon
 '((propname (radius diameter circumference area
       volume height velocity time
       weight power height work speed mass))
   (a/an     (a an))
   (the/its  (the its))
   (objname  (circle sphere fall lift))   ))

(def grammar '(
  (param     -> ((the/its)? (propname)) $2)
  (quantity  -> ((number)) $1)
  (object    -> ((a/an)? (objname) with (objprops))
                  (cons 'object (cons $2 $4)))
  (objprop   -> ((a/an)? (propname) of ? (quantity))
                  (list $2 $4))
  (objprop   -> ((propname) = (quantity))
                  (list $1 $3))
  (objprops  -> ((objprop) and (objprops))
                  (cons $1 $3))
  (objprops  -> ((objprop))  (list $1))
  (s         -> (what is (param) of (object))
                  (list 'calculate $3 $5))   ))
```

[18] file `physgram.lsp`

Physics Queries

```
% clojure
user=> (load-file "cs378/physics.clj")
user=> (load-files)
user=> (gramcom grammar)

user=> (phys '(what is the area of a circle
               with radius = 2))

(calculate area (object circle (radius 2)))
12.566370614

user=> (phys '(what is the circumference of
               a circle with an area of 100))

(calculate circumference (object circle (area 100))
35.44907701760372

user=> (phys '(what is the power of a lift with
              mass = 100 and height = 6
              and time = 10))

(calculate power (object lift (mass 100)
                              (height 6) (time 10))
588.399
```

Physics Units

An important part of physics problems is units of measurement. These can easily be handled by multiplying the number by the conversion factor that converts the unit to SI (metric) units. For example, an inch is 0.0254 meter, so 40 inches is 1.016 meter.

```
(phys '(what is the area of a circle
         with radius = 40 inches))

(calculate area (object circle (radius 1.016)))

3.2429278661312964
```

The `calculate` form can accept a multiply by a factor, allowing units for the goal of the calculation:

```
(phys '(what is the area in square-inches
        of a circle with radius = 1 meter))

(calculate (* area 6.4516E-4)
           (object circle (radius 1.0)))

4869.478351881704
```

Physics Changes

An interesting kind of question is how one quantity varies in terms of another quantity:

```
(phys '(how does the force of gravitation
         vary with radius))
(varywith gravitation force radius)

inverse-square

(phys '(how does the area of a circle vary
         if radius is doubled))
(varychg circle area ((radius 2.0)))

4.0
```

These questions can be answered easily as follows:

1. Find an equation that relates these variables.
2. Solve the equation for the desired variable.
3. Evaluate the rhs of the solved equation with all variables set to 1.
4. Change the variable(s) that change to 2.0 (or the appropriate value) and evaluate again.
5. The ratio of the two evaluations gives the answer.

Natural Language Interfaces

Interfaces for understanding language about limited domains are easy:

- Database access.
- Particular technical areas.
- Consumer services (e.g., banking).

Benefits:

- Little or no training required.
- User acceptance.
- Flexible.

Specialized language is much easier to handle than general English. The more jargon used, the better: jargon is usually unambiguous.

Problems with NL Interfaces

- **Slow Typing:** A formal query language might be faster for experienced users.
- **Typing Errors:** Most people are poor typists. A spelling corrector and line editor are essential.
- **Complex Queries:** Users may not be able to correctly state a query in English. "All that glitters is not gold."
- **Responsive Answers:**

  ```
  Q: How many students failed CS 381K
        in Summer 2018?
  A: 0
  ```

  Does this mean:

  1. Nobody failed.
  2. CS 381K was not offered in Summer 2018.
  3. There is no such course as CS 381K.
- **Gaps:** Is it possible to state the desired question?