CS 343 Artificial Intelligence

Gordon S. Novak Jr.
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

(512) 471-9569

novak@cs.utexas.edu
http://www.cs.utexas.edu/users/novak

Artificial Intelligence as Science

Intelligence should be placed in the context of biology:

> Intelligence *connects perception to action* to help an organism survive.

Intelligence is computation in the service of life, just as metabolism is chemistry in the service of life.

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory, and computation. Many points on the spectrum of intelligence-versus-cost are viable, from insects to humans.

AI seeks to *understand the computations required for intelligent behavior* and to produce computer systems that exhibit intelligence.

Aspects of intelligence studied by AI include perception, motor control, communication using human languages, reasoning, planning, learning, and memory.

Scientific Goals of AI

AI seeks to understand the working of the mind in mechanistic terms, just as medicine seeks to understand the working of the body in mechanistic terms.

> The mind is what the brain does.
>
> – Marvin Minsky

The *strong AI* position is that any aspect of human intelligence could, in principle, be mechanized.

A.I. as Engineering

How can we make computer systems more intelligent?

- *Perception* to get input directly from the real world.
- *Autonomy* to perform tasks that currently require human operators without human intervention or monitoring.
- *Flexibility* in dealing with variability in the environment.
- *Ease of use*: computers that are able to understand what the user wants from limited instructions in natural languages.
- *Learning* from experience.

Perception

Machine Vision:

It is easy to interface a TV camera to a computer and get an image into memory; the problem is *understanding* what the image represents. Vision takes *lots* of computation; in humans, roughly 10% of all calories consumed are burned in vision computation.

Speech Understanding:

Speech understanding is available now. Some systems must be trained for the individual user and require pauses between words. Understanding continuous speech with a larger vocabulary is harder.

Touch (*tactile* or *haptic*) Sensation:

Important for robot assembly tasks.

Robotics

Although industrial robots have been expensive, robot hardware can be cheap: Radio Shack has sold a working robot arm and hand for $15. The limiting factor in application of robotics is not the cost of the robot hardware itself.

What is needed is perception and intelligence to tell the robot what to do; "blind" robots are limited to very well-structured tasks (like spray painting car bodies).

Natural Language Understanding:

Natural languages are human languages such as English. Making computers understand English allows non-programmers to use them with little training. Applications in limited areas (such as access to data bases) are easy.

```
(askr '(where can i get ice cream in berkeley))
```

Natural Language Generation:

Easier than NL understanding. Can be an inexpensive output device.

Machine Translation:

Usable translation of text is available now. Important for organizations that operate in many countries.

> In a not too far future develops for eleven-year old David in a research lab the first intelligent robot with human feelings in the shape. But its "foster parents" are overtaxed with the artificial spare child and suspend it. Posed on itself alone David tries to fathom its origin and the secret of its existence.

Planning

Planning attempts to order actions to achieve goals.

Planning applications include logistics, manufacturing scheduling, planning manufacturing steps to construct a desired product.

There are huge amounts of money to be saved through better planning.

Expert Systems

Expert Systems attempt to capture the knowledge of a human expert and make it available through a computer program. There have been many successful and economically valuable applications of expert systems.

Benefits:

- Reducing skill level needed to operate complex devices.
- Diagnostic advice for device repair.
- Interpretation of complex data.
- "Cloning" of scarce expertise.
- Capturing knowledge of expert who is about to retire.
- Combining knowledge of multiple experts.
- Intelligent training.

Theorem Proving

Proving mathematical theorems might seem to be mainly of academic interest. However, many practical problems can be cast in terms of theorems. A general theorem prover can therefore be widely applicable.

Examples:

- Automatic construction of compiler code generators from a description of a CPU's instruction set.
- J Moore and colleagues proved correctness of the floating-point division algorithm on AMD CPU chip.

Symbolic Mathematics

Symbolic mathematics refers to manipulation of *formulas*, rather than arithmetic on numeric values.

- Algebra
- Differential and Integral Calculus

Symbolic manipulation is often used in conjunction with ordinary scientific computation as a generator of programs used to actually do the calculations. Symbolic manipulation programs are an important component of scientific and engineering workstations.

```
>(solvefor
   '(= v (* v0 (- 1 (exp (- (/ t (* r c)))))))
   't)

(= T (* (- (LOG (- 1 (/ V V0)))) (* R C)))
```

Game Playing

Games are good vehicles for research because they are well formalized, small, and self-contained. They are therefore easily programmed.

Games can be good models of competitive situations, so principles discovered in game-playing programs may be applicable to practical problems.

Prof. Peter Stone does research on strategies for bidding at auctions.

Characteristics of A.I. Programs

- **Symbolic Reasoning:** reasoning about objects represented by symbols, and their properties and relationships, not just numerical calculations.
- **Knowledge:** General principles are stored in the program and used for reasoning about novel situations.
- **Search:** a "weak method" for finding a solution to a problem when no direct method exists. Problem: *combinatoric explosion* of possibilities.
- **Flexible Control:** Direction of processing can be changed by changing facts in the environment.

Symbolic Representation

Most of the reasoning that people do is non-numeric. AI programs often do some numerical calculation, but focus on reasoning with symbols that represent objects and relationships in the real world.

- Objects.
- Properties of objects.
- Relationships among objects.
- Rules about classes of objects.

Examples of symbolic processing:

- Understanding English:

```
(show me a good chinese restaurant in los altos)
```

- Reasoning based on general principles:

```
if:   the patient is male
then: the patient is not pregnant
```

- Symbolic mathematics:

```
If y = m*x+b, what is the derivative
of y with respect to x?
```

Knowledge Representation

It is necessary to represent the computer's knowledge of the world by some kind of data structures in the machine's memory. Traditional computer programs deal with large amounts of data that are structured in simple and uniform ways. A.I. programs need to deal with complex relationships, reflecting the complexity of the real world.

Several kinds of knowledge need to be represented:

- **Factual Data:** Known facts about the world.
- **General Principles:** "Every dog is a mammal."
- **Hypothetical Data:** The computer must consider hypotheticals in order to reason about the effects of actions that are being contemplated.

Semantic Networks / Frames

A *semantic network* or *frame system* represents knowledge by *nodes* and *labeled arcs* among nodes.

Logic

Facts can also be represented in a logical formalism. In principle, logic and semantic network representations can be equivalent.

```
(huey huey-457)

(all x (if (huey x) (helicopter x)))

(all x (if (huey x) (payload x 4000)))

(all x (if (helicopter x) (can-fly x)))
```

Logic ...

Mathematical logic formalizes certain kinds of reasoning in terms of operations on mathematical formulas. It is important for people working in A.I. to know logic for several reasons:

- **Theory:** Logic has a sound mathematical foundation; things can be proved about it.
- **Applications:** For certain classes of applications (e.g., proving correctness of programs) logic is the representation of choice.
- **Comparison with Other Methods:** Other representation methods are often reducible to logic. Knowing logic helps in understanding other methods and may help prevent reinvention of old techniques.

A.I. is the Future of Computing!

Moore's Law predicts that computation will get faster/cheaper by a factor of 2 every 1.5 years; this means a factor of 1000 in 15 years. What will we do with all that cheap computation? Traditional applications of computers (with a few exceptions, such as large simulations) will not absorb enough additional computing power to make them major growth areas.

In the past, computation was scarce and manufacturers could just sell cycles. Now that cycles are cheap, ease of use is a major selling criterion.

Integration of computers with sensors and effectors will be common and essential for competitive position in many markets.

Consumer markets will not be similar to existing computer applications, but will require intelligence to deal with poorly structured tasks (such as cleaning the house).

Advantages of Lisp

- **Recursion:** A program can call itself as a subroutine. *Dynamic type checking* makes recursion more useful.
- **Garbage Collection:** automatically recycles memory.
- **Uniform Representation:** Programs and data are the same.
  - Programs can examine other programs.
  - Programs can write programs.
  - Programs can modify themselves (learn).
  - Data structures can contain programs; programs can contain data structures.
- **Interaction:** User can combine program writing, compilation, testing, debugging, running in a single interactive session.

If you want to do AI, and you don't start with Lisp, you will have to reinvent it.

You can't be a true Ninja code warrior unless you master the ancient art of Lisp!

Lisp Code

Lisp code is based on a few simple rules:

- Parentheses enclose a function name and its arguments: `(sqrt x)`
- *All* operations are done by *function calls*: `(+ x y)`
- The assignment operator is called `setq`:
  `(setq area (* pi (expt radius 2)))`
- Every function call returns a value.

```
(defun abs (x)
  (if (>= x 0)
      x
      (- x)))
```

- Local variables are declared by a `let`:
  `(let (x y) (setq x 3) ... )`

Quick Lisp

```
int myfn (a, b)                 (defun myfn (a b)
  { int i = 3; float x;           (let ((i 3) x)
    ...     }                        ...  ))

{ statement; ... }              (progn statement ... )

i = j + 2;                      (setq i (+ j 2))

sqrt(x)                         (sqrt x)

if ( i > j && j > k )           (if (and (> i j)
                                         (> j k))
   statement1                       statement1
   else statement2;                 statement2)

for (i=0; i< n; i++) ...        (dotimes (i n) ...)

while ( i < n ) statement;      (while (< i n)
                                  statements)

printf("%d\n", i);              (print i)
```

Lisp Data

- *Symbols* may contain letters, numerals, and some special characters
  `+ - * / @ $ % ^ & _ < > ~ . ?`

  `MASS CS343 WIDGET-ALIGNMENT-SCREW ?X`

- *Numbers*: floating-point, complex, integer (including *bignums* or big numbers), rational: `1/3`.
- *Atoms* include symbols and numbers.
- *S-expressions* (symbolic expressions) are defined *recursively* as follows:
  - An atom is an S-expression.
  - If $x_1 ... x_n$ are S-expressions, then $(x_1 ... x_n)$, called a *list* of $x_1 ... x_n$, is an S-expression.

```
ONTOGENY
(THIS IS A LIST)
(* PI (EXPT R 2))
(ALL X (IF (HUMAN X) (MORTAL X)))
()      ((()))      (((()())()))
```

The empty list `()` is equivalent to the symbol `NIL`.

Quotation

A symbol in Lisp can have a *value*, also called its *binding*. There must be a way to distinguish the symbol *itself* from its *value*. In English these are not formally distinguished.

> The President is the chief executive.
>
> The President has a wife named Laura.

In Lisp, we denote the symbol itself by *quoting* it with a single-quote symbol.

```
(GET 'PRESIDENT 'DUTIES)

(GET PRESIDENT 'SPOUSE)
```

Internally, the ' symbol becomes the pseudo-function `QUOTE`:

```
(GET (QUOTE PRESIDENT) (QUOTE DUTIES))
```

Quoting a list makes a constant data structure:

```
(SETQ FORMULA '(= AREA (* PI (EXPT R 2))))
```

Variable Values in Lisp

We can think of a symbol as a data structure that includes a *value cell* containing a *pointer* to the value of the atom. The value of the symbol can be set using the function `SET`:

```
(SET 'PRESIDENT 'JEFFERSON)
```

If we now *evaluate* `PRESIDENT`, we get the value `JEFFERSON`.

Since the first argument of `SET` is usually quoted, there is a special function `SETQ` that does this automatically.

```
(SETQ PRESIDENT 'JEFFERSON)
```

```
(SETQ RADIUS 5.0)
```

```
(* PI (EXPT RADIUS 2))
```

Constructing Lists

An important feature of Lisp for AI applications is the ability to construct new symbolic structures of arbitrary size and complexity at runtime.

A new list structure with a fixed number of elements can be made using the function `LIST`. To make a list of items x_1 ... x_n, use the form:

`(LIST` x_1 `...` x_n`)`

Each argument of `LIST` is evaluated unless it is quoted.

```
(LIST 'X 'Y 'Z)  =  (X Y Z)

(LIST (+ 2 3) (* 2 3))  =  (5 6)

(SETQ MAN 'ADAM)
(SETQ WOMAN 'EVE)
(LIST MAN 'LOVES WOMAN) = (ADAM LOVES EVE)

(LIST (LIST 'A 'B) (LIST 'C 'D))
                    =  ((A B) (C D))
```

Extracting Parts of Lists

Parts of lists can be extracted by `CAR` and `CDR`:

- `CAR` or `FIRST` returns the *first element* of a list.
- `CDR` or `REST` returns the *rest* of a list after the first element.

```
(CAR '(A B C))           =  A
(CAR '((A) B C))         =  (A)
(CAR (CAR '((A) B C)))   =  A
(CAR 'A)     Error: A is not a list.
(CAR NIL)                =  NIL


(CDR '(A B C))           =  (B C)
(CDR '((A) B C))         =  (B C)
(CDR (CDR '(A B C)))     =  (C)
(CDR (CAR '((A) B C)))   =  ()  =  NIL
(CDR 'A)     Error: A is not a list.
(CDR NIL)                =  NIL
```

Combinations of CAR and CDR

Since combinations of `CAR` and `CDR` are frequently used, all combinations up to four uses of `CAR` and `CDR` are defined as functions of the form `CxxxR`:

```
(CAAR X)     = (CAR (CAR X))

(CADR X)     = (CAR (CDR X))

(CADDR X)    = (CAR (CDR (CDR X)))
```

It's worth memorizing common combinations:

`CAR`	or	`FIRST`	=	First element of a list
`CADR`	or	`SECOND`	=	Second element
`CADDR`	or	`THIRD`	=	Third element
`CADDDR`	or	`FOURTH`	=	Fourth element

Functions `FIRST` through `TENTH` are defined in Common Lisp.

Constructing List Structure

The basic function that constructs new list structure is the function `CONS`.

If `Y` is a list, then we can think of `(CONS X Y)` as adding the new *element* `X` to the front of the list `Y`.

```
(CONS 'A '(B))        =  (A B)

(CONS 'A NIL)         =  (A)

(CONS 'A '())         =  (A)

(CONS '(A) '(B))      =  ((A) B)

(CONS 'A 'B)          =  (A . B)
```

The following axioms always hold:

1. `(CAR (CONS x y))` = `x`
2. `(CDR (CONS x y))` = `y`

List Manipulation Functions

`APPEND` makes a new list consisting of the members of its argument lists. `APPEND` takes any number of arguments.

```
(APPEND '(A) '(B))          =  (A B)

(APPEND '(A B) '(C D))      =  (A B C D)

(APPEND '(A) '(B) '(C))     =  (A B C)
```

`REVERSE` makes a new list that is the reverse of the top level of the list given as its argument.

```
(REVERSE '(A B))            =  (B A)

(REVERSE '((A B)(C D)))     =  ((C D)(A B))
```

`LENGTH` returns the length of the top level of the list given as its argument.

```
(LENGTH '(A))       =  1
(LENGTH '(A B))     =  2
(LENGTH '((A B)))   =  1
```

Substitution

The function `SUBST` makes a new S-expression tree (not just a list) with a specified substitution.

`(SUBST x y z)` means "substitute `x` for `y` in `z`".

```
(SUBST 'JONES 'NAME '(DEAR MR NAME))

            =  (DEAR MR JONES)


(SUBST 5.0 'RADIUS
           '(* 3.14159 (EXPT RADIUS 2)))

            =  (* 3.14159 (EXPT 5.0 2))


(SUBST 'SOCRATES '?X
       '(IF (HUMAN ?X) (MORTAL ?X)))

            =  (IF (HUMAN SOCRATES)
                   (MORTAL SOCRATES))
```

Evaluation

Evaluation is the process by which Lisp determines the value of an expression. Expressions are evaluated using the following recursive algorithm:

1. If the expression to be evaluated is a number, `T`, or `NIL`, its value is the expression itself.
2. If the expression is `(QUOTE x)`, its value is `x`.
3. If the expression is a symbol, its value is the value of the symbol (the symbol's *binding*).
4. Otherwise, the expression must be a list that represents a function call:
   (a) Evaluate each argument of the function call, in left-to-right order.
   (b) Call the function with the resulting values of the arguments.
   (c) The value returned by the function is the value of the expression.

Requesting Evaluation

Evaluation of an s-expression can be explicitly requested using the function `EVAL`. `(EVAL x)` gives the result of evaluating the *value* of the expression `x` in the current execution context. That is, `EVAL` performs an *extra* level of evaluation. The argument of `EVAL` is evaluated once, and should return as its value some Lisp code; then `EVAL` causes that Lisp code to be evaluated.

```
(EVAL (LIST 'LIST (LIST 'QUOTE 'A)))
                        = (A)

(EVAL (SUBST 10.0 'RADIUS
          '(* 3.14159 (EXPT RADIUS 2))))

                        =  314.159
```

`EVAL` and its relatives allow Lisp code to be part of a data structure; the code can be retrieved from the data structure and executed by calling `EVAL`.

Building Lists Incrementally

Often one wants to build up a list of things incrementally; this is usually done using `CONS`. First, note that for any `x`,

`(CONS x NIL)` = `(LIST x)` :

```
(CONS 'A NIL)     =  (A)

(CONS '(A) NIL)   =  ((A))
```

Second, if `LST` is a list, then `(CONS x LST)` will make a new list with `x` as its first element, followed by the other elements of `LST`.

```
(CONS 'A '(B C))     =  (A B C)

(CONS '(A) '(B C))   =  ((A) B C)
```

Building Lists Incrementally ...

Therefore, a list `LST` can be built up as follows:

1. Initially, set `LST` to `NIL` or `'()` (a list of no elements).
2. For each new element `x`, set `LST` to `(CONS x LST)` or use `(PUSH x LST)`.

The following example illustrates "`CONS`ing up" a list. Note that the element added *last* will be at the *front* of the list; `REVERSE` can be used to reverse the order if desired.

```
                                    LST:
(SETQ LST '())                      NIL = ()

(SETQ LST (CONS 'A LST))            (A)

(SETQ LST (CONS 'B LST))            (B A)

(PUSH 'C LST)                       (C B A)

(SETQ LST (REVERSE LST))            (A B C)
```

Stepping Through A List

`CAR` (first element of a list) and `CDR` (remainder of the list) can be used to step through a list one element at a time:

1. Initially, set a variable `LST` to the list to be processed.
2. If `LST` is `NIL`, quit. Otherwise,

   ```
   (SETQ TOP (CAR LST))  or (SETQ TOP (POP LST))
   (SETQ LST (CDR LST))
   ```

3. Process the element `TOP`; go to step 2.

Such loops are so frequently used that a macro is provided for them:

```
(DOLIST (var list) code)
```

will bind `var` to successive elements of `list` and execute `code` for each element.

```
(DOLIST (NAME NAMELIST)
  (PRINT (SUBST NAME 'TARGET
                '(DEAR MR TARGET))) )
```

Predicates

A *predicate* in Lisp is a function that performs a test and returns either `T` (True) or `NIL` (False). Note: Lisp functions that test predicate values consider `NIL` to be False and *anything else* to be True. Predicate names often end in `P`. Some commonly used predicates are:

`(SYMBOLP x)`	`True if x is a SYMBOL.`
`(ATOM x)`	`True if x is an ATOM (anything besides a CONS).`
`(NULL x)`	`True if x is NIL.`
`(CONSP x)`	`True if x is a CONS cell.`
`(NUMBERP x)`	`True if x is a number.`
`(ZEROP x)`	`True if x is zero; error if x not a number.`
`(MINUSP x)`	`True if x is negative; error if x not a number.`

More Predicates

`(EQ x y)`	`True if x and y are the same pointer value. Always works for symbols.`
`(EQL x y)`	`True if x and y are EQ or are equal numbers.`
`(EQUAL x y)`	`True if x and y have isomorphic structure (‘‘print the same’’).`
`(< x y)`	`True if x < y. Others are >, >=, <=, =, /= (not equal).`

Logical Operators

Predicates can be combined by the logical operators `AND`, `OR`, and `NOT`.

`(NOT x)` returns `T` if `x` is `NIL`; otherwise, it returns `NIL`. `NOT` is therefore the same as `NULL`.

`(AND` x_1 `...` x_n`)` evaluates each of x_1 ... x_n in order. As soon as any x_i returns `NIL`, `AND` returns `NIL` without evaluating any remaining `x`'s. If every x_i returns a non-`NIL` value, the value of `AND` is the value of x_n.

```
(DEFUN SAFE-SQRT (X)
  (AND (NUMBERP X)
       (NOT (MINUSP X))
       (SQRT X)) )
```

`(OR` x_1 `...` x_n`)` evaluates each of x_1 ... x_n in order. As soon as any x_i returns a non-`NIL` value, `OR` returns that value without evaluating any remaining `x`'s. If every x_i returns `NIL`, the value of `OR` is `NIL`.

```
(OR (> CASH 100) (PRINT ‘‘Get money.’’))
```

IF Statement

Common Lisp provides an IF statement:

```
(IF <test> <then-form>)
```

```
(IF <test> <then-form> <else-form>)
```

The `<test>` is evaluated first. If it returns a non-`NIL` value, the `<then-form>` is evaluated and its value is the value of the `IF`; otherwise, the `<else-form>` is evaluated and its value is the value of the `IF`.

Since `NIL` is false and *anything else* is treated as true, a common convention is for a function to return `NIL` if it did not work, or an answer if it did work.

```
(SETQ Y (IF (< X 0.0)
            (* X X)
            (SQRT X)))
```

Note that the `<then-form>` and `<else-form>` are single forms; if multiple things need to be done, they must be enclosed in a `PROGN`.

EQ Predicate

The predicate `EQ` tests for equality of *pointer values* inside the machine. It is faster than `EQUAL`, but less general in its applicability.

`EQ` always works for comparisons where at least one comparand is a symbol, since symbols are always unique structures in memory. Comparisons against constant symbols are usually done with `EQ`; `EQL` is like `EQ`, but also works for numbers.

```
(IF (EQ (CAR FORM) '+) (PRINT 'ADD))
```

In general, `EQ` does *not* work for numbers or for non-atomic s-expressions.

```
(EQ 'A 'A)            =   T
(EQ '(A) '(A))        =   NIL
(EQUAL '(A) '(A))     =   T
(EQ (+ 2 2) 4)        =   T ? (implementation
                              dependent)
(EQUAL (+ 2 2) 4)     =   T
(EQL (+ 2 2) 4)       =   T
```

Membership Testing

A list of items can be used as a representation of a set. The function `(MEMBER x lst)` tests whether the item `x` is a member of the list `lst`. If `x` is a member of `lst`, the value of `MEMBER` is the tail of the list `lst` beginning with the place where `x` was found; otherwise, the value of `MEMBER` is `NIL`. The default test used by `MEMBER` is `EQL`.

```
(SETQ CLUB '(TOM DICK HARRY))

(MEMBER 'DICK CLUB)  =  (DICK HARRY)

(MEMBER 'FRED CLUB)  =  NIL

(MEMBER '(JOHN MARY) COUPLES :TEST #'EQUAL)
```

Association Lists

A simple "database" facility is provided by the *association list*. This is a list of sublists, in which the first element of a sublist is the key value and the remainder of the sublist is the associated data. Association lists are a simple way to implement small databases.

`(ASSOC x l)` searches `l`, a list of lists, for an element that begins with `x`. The result is the element that was found, or `NIL` if no matching element is found.

```
(ASSOC 'TWO '((ONE 1) (TWO 2) (THREE 3)))

                    =  (TWO 2)
```

Note that an additional operation (in this case, `CADR`) is usually required on the result of `ASSOC` to get the desired data.

```
(CADR (ASSOC 'TWO '((ONE 1) (TWO 2) ...)))

                    =  2
```

Use of Recursion in Lisp

A *recursive* program is one that calls itself as a subroutine. Use of recursion in Lisp can result in programs that are powerful, yet simple and elegant. Often, a large problem can be handled by a small program which:

1. Tests for a *basic case* and computes the value of a basic case directly.
2. Otherwise, does *part* of the job and calls itself recursively to do the rest of the job.

A good way to learn recursion, and to gain an appreciation of the beauty of Lisp, is to study definitions of basic Lisp functions written recursively in Lisp. Reference to these definitions can also be used to answer questions about how the functions work in particular cases.

Examples of Recursion in Lisp

```
; Length of a list
(DEFUN LENGTH (L)
  (IF (CONSP L)
      (1+ (LENGTH (REST L)))
      0) )


; Last CONS cell in a list
(DEFUN LAST (L)
  (IF L (IF (REST L)
            (LAST (REST L))
            L)) )


; Membership in a list
(DEFUN MEMBER (X L)
  (IF L (IF (EQL X (FIRST L))
            L
            (MEMBER X (REST L)) )) )
```

More Examples of Recursion

```
; Append two lists
(DEFUN APPEND (X Y)
  (IF X (CONS (FIRST X)
              (APPEND (REST X) Y))
        Y) )


; Reverse a list
(DEFUN REVERSE (L)
  (REVERSE1 NIL L))

(DEFUN REVERSE1 (ANSWER LST)
 (IF LST
     (REVERSE1 (CONS (FIRST LST) ANSWER)
               (REST LST))
     ANSWER))


; Associate key with list of pairs
(DEFUN ASSOC (X L)
  (IF L (IF (EQL X (FIRST (FIRST L)))
            (FIRST L)
            (ASSOC X (REST L)) )) )
```

Binary Tree Recursion

The recursions we have seen so far have involved linear structures, that is, lists in which only the top level was involved. A second class of recursive programs operates on both halves of a `CONS` cell; in general, such functions call themselves *twice*.

```
; Copy a tree structure
(DEFUN COPY-TREE (X)

  (IF (CONSP X)
      (CONS (COPY-TREE (CAR X))
            (COPY-TREE (CDR X)))
      X) )


; Test equality of structure
(DEFUN EQUAL (X Y)

  (COND ((EQL X Y) T)
        ((OR (ATOM X) (ATOM Y)) NIL)
        ((EQUAL (CAR X) (CAR Y))
          (EQUAL (CDR X) (CDR Y)) )) )
```

SUBST is COPY-TREE with Substitution

`SUBST` makes a substitution throughout a structure; `(SUBST x y z)` can be read "substitute `x` for `y` in `z`".

```
(DEFUN SUBST (X Y Z)
  (IF (EQL Y Z)
      X
      (IF (CONSP Z)
          (CONS (SUBST X Y (CAR Z))
                (SUBST X Y (CDR Z)))
          Z) ) )
```

Executing Multiple Statements

Often, one wishes to execute multiple function calls in order.

`(PROGN <statement`$_1$`> ... <statement`$_n$`>)`

will execute `<statement`$_1$`>` through `<statement`$_n$`>` in order; the value of `PROGN` is the value of `<statement`$_n$`>`.

Multiple statements are automatically executed at the "top level" of a function or within a `COND` clause; this is sometimes referred to as an *implicit* `PROGN`.

`(PROG1 <statement`$_1$`> ... <statement`$_n$`>)`

will execute `<statement`$_1$`>` through `<statement`$_n$`>` in order; the value of `PROG1` is the value of `<statement`$_1$`>`.

The LET Construct

The `LET` construct allows the programmer to declare local variables for temporary use and execute multiple forms within the scope of the variable bindings.

```
(LET  ( <variables> )

      <statement-1>
      ...
      <statement-n>  )
```

The `<variables>` are initially bound to `NIL` when the `LET` is entered; their values may be changed using `SETQ`.

Variables may be initialized to values other than `NIL` by specifying a pair, `(<var> <init-form>)`, in the list of variables.

The value of the `LET` is the value of `<statement-n>`, which could be just a variable name.

Example of LET

```
; Density of a sphere
(DEFUN SPHERE-DENSITY (RADIUS WEIGHT)

  (LET ( VOLUME
         (MASS (/ WEIGHT 9.88)) )

    (SETQ VOLUME (* 4/3 PI
                    (EXPT RADIUS 3)))

    (/ MASS VOLUME) ))
```

Iteration Using DOLIST

The control constructs provided by Common Lisp often allow shorter and more understandable code than use of `PROG` and `GO`, as illustrated by the following version of `LENGTH`:

```
(DEFUN LENGTH (L)

  (LET ((COUNT 0))

    (DOLIST (ITEM L) (INCF COUNT))

    COUNT ))
```

`INCF` is a macro that increments a value. `(INCF COUNT)` expands into
`(SETQ COUNT (1+ COUNT))`.

Search

Search programs find a solution for a problem by trying different *sequences* of *actions* (*operators*) until a solution is found.

Advantage:
Many kinds of problems can be viewed as search problems. To solve a problem using search, it is only necessary to code the operators that can be used; search will find the sequence of actions that will provide the desired result. For example, a program can be written to play chess using search if one knows the *rules* of chess; it isn't necessary to know how to play good chess.

Disadvantage:
Most problems have search spaces so large that it is impossible to search the whole space. Chess has been estimated to have 10^{120} possible games. The rapid growth of combinations of possible moves is called the *combinatoric explosion* problem.

Why Search is Necessary

There is no model of the world that is complete, consistent, and computable. Any intelligent system must encounter surprises.

Solutions to problems cannot be precomputed; many problems must be solved dynamically, starting from observed data.

Flexibility to deal with a variable environment requires search.

Ambiguity in interpretation of perceptual data requires search. Interpretation may be locally ambiguous, but global constraints may permit an unambiguous total interpretation.

Creativity can result from searching through many possible designs.

Outline of Search Topics

- State Space Search
  - Combinatoric Explosion Problem
  - Search Strategies
    - Depth-first, Breadth-First
    - Hill Climbing
    - Heuristic Search
    - Iterative Deepening
- Problem Reduction Search
  - Game Tree Search
- Other Topics
  - Genetic Algorithms
  - Example: DENDRAL

State Space Search

A *state space* represents a problem in terms of *states* and *operators* that change states.

A state space consists of:

- A representation of the *states* the system can be in. In a board game, for example, the board represents the current state of the game.
- A set of *operators* that can change one state into another state. In a board game, the operators are the legal moves from any given state. Often the operators are represented as programs that change a state representation to represent the new state.
- An *initial state.*
- A set of *final states*; some of these may be desirable, others undesirable. This set is often represented implicitly by a program that detects terminal states.

Tic-Tac-Toe as a State Space

State spaces are good representations for board games such as Tic-Tac-Toe. The state of a game can be described by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an `X` or `O` or be empty.

- **State:**
  - Player to move next: `X` or `O`.
  - Board configuration:

X		O
	O	
X		X

- **Operators:** Change an empty cell to `X` or `O`.
- **Start State:** Board empty; `X`'s turn.
- **Terminal States:**
  Three `X`'s in a row; Three `O`'s in a row; All cells full.

Search Tree

The sequence of states formed by possible moves is called a *search tree*. Each level of the tree is called a *ply* .

Since the same state may be reachable by different sequences of moves, the state space may in general be a graph. It may be treated as a tree for simplicity, at the cost of duplicating states.

Solving Problems Using Search

- Given an informal description of the problem, construct a formal description as a state space:
  - Define a data structure to represent the *state*.
  - Make a representation for the *initial state* from the given data.
  - Write programs to represent *operators* that change a given state representation to a new state representation.
  - Write program to detect *terminal states*.
- Choose an appropriate search technique:
  - How large is the search space?
  - How well-structured is the domain?
  - What knowledge about the domain can be used to guide the search?

Basic Recursive Algorithm

- If the input is a base case, for which the solution is known, return the solution.
- Otherwise,
  - Do part of the problem, or break it into smaller subproblems.
  - Call the problem solver recursively to solve the subproblems.
  - Combine the subproblem solutions to form a total solution.

In writing the recursive program:

- Write a clear specification of the input and output of the program.
- Assume it works already.
- Write the program to use the input form and produce the output form.

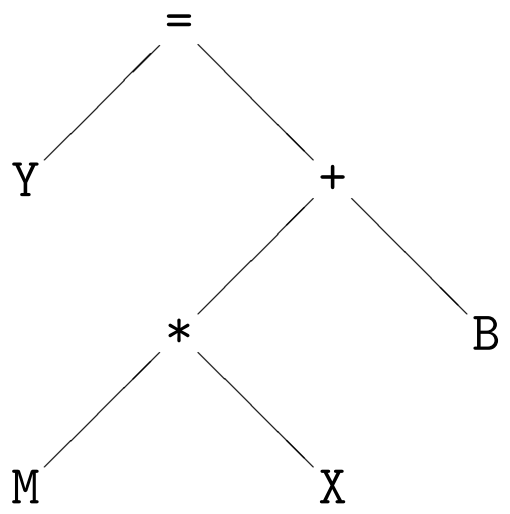
Solving Equations

One way to solve simple equations is to use rules of algebra to move operations to the opposite side until the desired variable is reached.

Equations are represented in Lisp as list structures or trees. The equation $y = m \cdot x + b$ is represented as:

```
(= Y (+ (* M X) B))
```

This is equivalent to the tree:



Solving Equations by Search

We can solve simple equations in this way:

- If only the desired variable is on the left, succeed: return the input equation.

  ```
  (= Y (+ (* M X) B))
  ```

- If only the desired variable is on the right, succeed: return the input equation with the arguments reversed.

  ```
  (= (+ (* M X) B) Y)
  ```

- If only an undesired variable is on the right, fail: return NIL.

  ```
  (= (+ (* M X) B) FOOBAR)
  ```

- Otherwise, try using an algebraic law to eliminate the top operator on the right; usually, there are two possibilities to try. Then try to solve the resulting equation; if either one succeeds, return that answer.

The strategy here is to search through *every possible legal rewriting of the equation* until we get the one we want.

Rewriting Equations

Given an equation `(= ?X (+ ?Y ?Z))`, where each of `?X`, `?Y`, and `?Z` could be any expression, we could use the laws of algebra to rewrite the equation as:

- `(= (- ?X ?Y) ?Z)`
- `(= (- ?X ?Z) ?Y)`

Assuming that the variables `?X`, `?Y`, and `?Z` have values, we could create the first pattern using `LIST`:

```
(LIST '= (LIST '- ?X ?Y) ?Z)
```

or using `backquote`:

- `` `(= (- ,?X ,?Y) ,?Z) ``

Backquote produces the same code as above; it quotes everything unless an item is un-quoted with a comma.

Examples of Equation Solving

```
>(solve '(= x 3) 'x)
(= X 3)

>(solve '(= 3 x) 'x)
(= X 3)

>(solve '(= y (+ x b)) 'x)
  1> (SOLVE (= Y (+ X B)) X)
    2> (TRY-LEFT (= Y (+ X B)))
    <2 (TRY-LEFT (= (- Y X) B))
    2> (SOLVE (= (- Y X) B) X)
    <2 (SOLVE NIL)
    2> (TRY-RIGHT (= Y (+ X B)))
    <2 (TRY-RIGHT (= (- Y B) X))
    2> (SOLVE (= (- Y B) X) X)
    <2 (SOLVE (= X (- Y B)))
  <1 (SOLVE (= X (- Y B)))
(= X (- Y B))

>(solve '(= y (+ (* m x) b)) 'x)
(= X (/ (- Y B) M))
```

Basic Depth-first Search Algorithm

```
; Output: list of operators or FAILURE
(defun search (state)
  (let (op oplist newstate)
    (if (terminalp state)
        (if (goalp state) '() 'failure)
        (progn
          (setq op (choose-op state))
          (setq newstate
                (funcall op state))
          (setq oplist (search newstate))
          (if (eq oplist 'failure)
              'failure
              (cons op oplist)) ) ) ))
```

Complications:

- We may have to try different operators.
  (`newstate` might be a dead end.)
- It may not be possible to apply `op`.
- Applying `op` might violate a constraint.
- We could get into a loop applying `op` and its inverse.

Comments on Search Algorithm

- The program continually goes deeper until it reaches a terminal state, which is either a goal or a failure.
- When the goal is found, `search` returns `'()` as its answer. This is an empty list of operators, since no operators are required to reach the goal.
- At each level as the search unwinds, the operator used at that level is put onto the front of the operator list using `cons`. `cons` adds a new item onto the front of a list:
  `(cons 'a '(b c))` = `(A B C)`

Recursive Depth-First Search

Function `sss(s prev ops)`:

1. If `s` is a goal, return `'()` as the answer. This is a list of no operators, since no operators are required to reach the goal state.
2. If `s` is a failure, or no operators remain, return `'failure`.
3. If the next operator, `op`, is applicable to the input state `s`, compute `new` by applying it to `s`.
   (a) If `new` duplicates one of its ancestor states on `prev`, try the next operator.
   (b) If a search for the goal from the `new` state,
   `(sss new (cons s prev) *ops*)`
   succeeds, return the `cons` of `op` onto the front of its operator sequence, `opseq`.
   `s` $\rightarrow$ `new` $\rightarrow$ ... $\rightarrow$ `goal`
   `op` `opseq`
   (c) Else, try the next operator.
4. Else, try the next operator.

State Space Search Program

```
(defun sss (s prev ops) ; state, prev states,
 (let ((op (pop ops)))  ; op = next ops to try
  (if (goal? s)         ; if s is a goal,
      '()               ; answer = list of no ops
    (if (or (failure? s) (null op))
        'failure
      (if (applicable? op s)
          (let ((new (apply-op op s)))
                  ; if it duplicates a prev state
            (if (member new prev :test #'equal)
                  ; try the next op
                (sss s prev ops)
                  ; else try to search from new
                (let ((opseq (sss new
                                  (cons s prev)
                                  *ops*)))
                    ; if search failed
                  (if (eq opseq 'failure)
                      ; try the next op
                      (sss s prev ops)
                      ; else cons op onto answer
                      (cons op opseq)))))
          ; try another op
          (sss s prev ops))))))
```

Notes on State Space Search Program

- This program is a generic depth-first state space search program, with detection of duplicate states. It can be applied to a domain by providing the functions `goal?`, `failure?`, `applicable?` and `apply-op`, and the list of operators `*ops*`.
- The program is recursive in two directions:
  - Down: `(sss new (cons s prev) *ops*)`
    This code tries to find a goal from the new state produced by applying one operator to the input state. The input state is added to the list of previous states, and all operators are available to be tried (the global variable `*ops*` is a list of all operators.).
  - Across: `(sss s prev ops)`
    This code tries search from the current state but using the next operator, since the operator that was just tried did not work.

Missionaries and Cannibals

The Missionaries and Cannibals problem illustrates the use of state space search for planning under constraints:

> Three missionaries and three cannibals wish to cross a river using a two-person boat. If at any time the cannibals outnumber the missionaries on either side of the river, they will eat the missionaries. How can a sequence of boat trips be performed that will get everyone to the other side of the river without any missionaries being eaten?

Missionaries/Cannibals Search Graph

Missionaries and Cannibals Representation

State Representation:

1. `BOAT` position: original (`T`) or final (`NIL`) side of the river.
2. Number of `M`issionaries and `C`annibals on the original side of the river.
3. Start is `(T 3 3)`; Goal is `(NIL 0 0)`.

Operators:

`(MM 2 0)`	Two Missionaries cross the river.
`(MC 1 1)`	One Missionary and one Cannibal.
`(CC 0 2)`	Two Cannibals.
`(M 1 0)`	One Missionary.
`(C 0 1)`	One Cannibal.

Functions for Missionaries and Cannibals

```
(defun ml (state) (second state))   ; m on left
(defun mr (state) (- *mtotal* (second state)))
(defun m (s) (if (boat s) (ml s) (mr s)))

(defun applicable? (op s)
  (and (<= (op-m op) (m s)) (<= (op-c op) (c s))))

; Test for failure: c > m > 0 on either side.
(defun failure? (s)
  (or (> (cl s) (ml s) 0)
      (> (cr s) (mr s) 0) ) )

(defun goal? (s)    ; 0 mis and 0 can on left
  (and (= (ml s) 0) (= (cl s) 0) ) )

(defun apply-op (op s)   ; apply op => new state
  (if (boat s)
      (list nil (- (ml s) (op-m op))
                (- (cl s) (op-c op)))
      (list t   (+ (ml s) (op-m op))
                (+ (cl s) (op-c op))) ))
```

Testing Missionaries and Cannibals

```
>(load "/projects/cs381k/miscan.lsp")

>(testmc 1 1)

((MC 1 1))

>(testmc 2 1)

((MM 2 0) (M 1 0) (MC 1 1))

> (testmc 3 3)

((MC 1 1) (M 1 0)  (CC 0 2) (C 0 1)
 (MM 2 0) (MC 1 1) (MM 2 0) (C 0 1)
 (CC 0 2) (M 1 0)  (MC 1 1))

> (testmc 4 4)

FAILURE
```

Combinatoric Explosion

Suppose that each node of a search tree has b descendants; this is sometimes called the *average branching factor*.[1] Then the number of nodes at the bottom of a tree d plies deep is b^d. This number grows exponentially with depth and can quickly become very large as the search becomes deeper; this rapid growth is called the *combinatoric explosion* problem.

The maximum depth, i.e. maximum number of steps between any two states, is called the *diameter* of the problem space.

In chess, there may be 30 possible moves from each state. A chess tree 10 plies deep would require searching 30^{10} or nearly 10^{15} nodes. On a computer that could examine one node in a microsecond, it would take over 18 years of computer time to examine this tree.

[1] Real search trees often have different numbers of descendants from different nodes.

Do Faster Computers Help?

If searching takes too long, the obvious solution would appear to be to get a faster computer (or many parallel computers). Unfortunately, this often does *not* help.

The exponential growth of search trees can outrun computer power. In chess, a computer 1,000 times as fast yields a benefit of only *two* plies deeper search in the same amount of time; a computer 1,000,000 times as fast yields *four* plies.

The Moral:

> Use *knowledge* about the problem domain to reduce the size of the search space.

Irrelevant Operators

Irrelevant operators increase the branching factor b, and this rapidly increases the size of the search space, b^d.

A human chess master considers only 1 to 3 board positions per second, but considers only at the most useful moves. A computer chess program may consider millions of positions per second, but most of those are foolish moves.

Moral: Use *knowledge* about the problem domain to reduce the number of operators considered at any given time.

Search Order

The excessive time spent in searching is *almost entirely spent on failures* (sequences of operators that do not lead to solutions). If the computer could be made to look at promising sequences first and avoid most of the bad ones, much of the effort of searching could be avoided.

Blind search methods try operators in some fixed order, without knowing which operators may be more likely to lead to a solution. Such methods can succeed only for small search spaces.

Heuristic search methods use knowledge about the problem domain to choose more promising operators first.

Search Order

Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.

Depth-First Search

Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.

Advantages:

1. Low storage requirement: *linear* with tree depth.
2. Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages:

1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).
2. Incomplete: without a depth bound, may not find a solution even if one exists.

Bounded Depth-First Search

Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists.

An easy way to solve this problem is to put a maximum depth bound on the search. Beyond the depth bound , a failure is generated automatically without exploring any deeper.

Problems:

1. It's hard to guess how deep the solution lies.
2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of b^{extra}.
3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

Iterative Deepening

Iterative deepening begins a search with a depth bound of 1, then increases the bound by 1 until a solution is found.

Advantages:

1. Finds an optimal solution (shortest number of steps).
2. Has the low (linear in depth) storage requirement of depth-first search.

Disadvantage:

1. Some computer time is wasted re-exploring the higher parts of the search tree. However, this actually is not a very high cost.

Cost of Iterative Deepening

In general, $(b-1)/b$ of the nodes of a search tree are on the bottom row. If the branching factor is $b = 2$, half the nodes are on the bottom; with a higher branching factor, the proportion on the bottom row is higher.

Korf calculates the work done by iterative deepening as $b^d * (1 - 1/b)^{-2}$, where the multiplier approaches 1 as b increases.[2]

My calculation of the work multiplier for iterative deepening is $(b+1)/(b-1)$, which is not far from Korf's result. The multiplier is a constant, independent of depth.

b	multiplier
2	3.00
3	2.00
4	1.67
5	1.50
10	1.22

[2]Korf, Richard E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*. vol. 27, no. 1, pp. 97-112, Sept. 1985.

Using Heuristics to Guide Search

We now turn to methods for using heuristic knowledge to make search more efficient.

Finding a route from one city to another city is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

- State: the current city in which the traveler is located.
- Operators: roads linking the current city to other cities.
- Cost Metric: the cost of taking a given road between cities (distance, time required, dollar cost, a weighted sum of costs, etc.).
- Heuristic Information: the search could be guided by the *direction* of the goal city from the current city, or we could use *airline distance* as an estimate of the distance to the goal.

Hill Climbing

A strategy for climbing a hill in a fog is to move upward.

A heuristic that estimates distance to the goal can be used to guide a hill-climbing search. A discrete depth-first search guided by such a heuristic is called *greedy best-first search*; it can be very efficient. For example, in route finding, hill climbing could be implemented by selecting the next city that is closest to the goal.

Unfortunately, hill-climbing sometimes gets into trouble:

Ramdom restart is one way to recover from local maxima.

Breadth-First Search

Breadth-first search generates new states in the order of their distance from the start state. All states at level i are examined before any states at level $i + 1$ are examined.

Advantages:

1. Guaranteed to find an optimal solution (in terms of shortest number of steps to reach the goal).
2. Can always find a goal node if one exists (complete).

Disadvantages:

1. High storage requirement: *exponential* with tree depth.

Breadth-First Search Algorithm

1. Put the start node `s` on a queue called `open`. `open` contains nodes that are still to be examined.
2. While `open` is non-empty,
   (a) Remove the first node `n` from `open`; put `n` on a list called `closed`. If `n` is a goal node, terminate with success. The solution path is given by the pointers from `n` back to the start node.
   (b) Expand node `n` (generate its successors). For each successor node `m`, if it is neither on `open` nor on `closed`, put a pointer from `m` back to `n` and record the operator used; insert `m` at the end of the `open` queue.
3. `open` is empty; terminate with failure.

Uniform-Cost Search

Uniform-cost search is similar to breadth-first search. We associate with each node n a cost $g(n)$ that measures the cost of getting to node n from the *start* node. $g(start) = 0$. If n_i is a successor of n, then $g(n_i) = g(n) + c(n, n_i)$, where $c(n, n_i)$ is the cost of going from node n to node n_i.

Instead of considering the *first* node on `open`, as in breadth-first search, the *least-cost* node on `open` is expanded.

Advantage:

1. Guaranteed to find the least-cost solution.

Disadvantages:

1. Exponential storage required.
2. `open` list must be kept sorted (as a priority queue).
3. Must change cost of a node on `open` if a lower-cost path to it is found.

Uniform-cost search is the same as Heuristic Search when no heuristic information is available (heuristic function h is always 0).

Ordered Search

Ordered search is like breadth-first search, except that it selects for expansion the node generated so far that looks the most promising according to a chosen heuristic function.

Advantages:

1. If the heuristic is good, then a good path can be found quickly.

Disadvantages:

1. Extra overhead to evaluate the heuristic function.
2. Large storage to store `open` and `closed` node lists.
3. Large CPU time required if heuristic function is inaccurate.

The Ordered Search Algorithm

1. Put the start node `s` on `open`; compute f(`s`).
2. While `open` is non-empty,
   (a) Remove the lowest-cost node `n` from `open`; put `n` on `closed`. If `n` is a goal, terminate with success. The solution path is given by the pointers from `n` back to the start node.
   (b) Expand node `n` (generate its successors). For each successor node `m`,
      i. Compute f(`m`).
      ii. If `m` is neither on `open` nor on `closed`, place `m` on `open` so that `open` remains ordered. Put a pointer from `m` back to `n` and record the operator used.
      iii. If `m` is already on `open` or `closed`, and the path to `m` just found is better than the one previously found, change the cost value and pointer of node `m` to the new values and put `m` on `open`.
3. `open` is empty; terminate with failure.

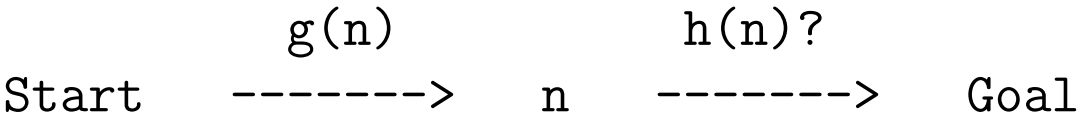
Evaluation Functions

- $h^*(n)$ = cost of the shortest path between node n and a goal node (usually not known).
- $h(n)$ = estimated cost to get to a nearest goal node from n.
- $g(n)$ = cost to get to node n from the start node via the lowest-cost path found so far.

Heuristic Search: A*

Heuristic Search chooses the next node to expand (consider descendants of) based on *lowest estimated total cost* of a path through the node.

Estimated Total Cost $f(n) = g(n) + h(n)$
$g(n)$ = Cost from Start to n [known]
$h(n)$ = Cost from n to Goal [estimated]



The *heuristic function*, h, estimates the cost of getting from a given node n to the goal. If h is good, the search will be highly directed and efficient; for example, airline distance is an excellent heuristic distance estimator for route finding. If h is not so good or has pathologies, inclusion of the known cost g keeps the search from getting stuck or going too far astray.

Heuristic Search for Route Finding

Route finding is a good example for heuristic search because an excellent heuristic function exists: straight-line (or great-circle) distance between points. This heuristic function has the features:

- It is easily computed.
- It is non-overestimating; therefore, the lowest-cost solution is guaranteed to be found.
- It is powerful (good estimate of true cost).

Ordered Search for Route Finding

- $h(n) = 0$: Search proceeds in all directions

- $f(n) = h(n) + g(n)$: Search directed toward goal

- $h = h*$: Search proceeds directly to goal

Effect of Heuristic Function

Admissibility of Heuristic Function

A search algorithm is *admissible* if it always finds an optimal solution path if a solution path exists.

It can be shown[3] that the ordered search algorithm, using the heuristic function

$$f(n) = g(n) + h(n)$$

is admissible iff for all nodes n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual cost of getting from node n to a nearest goal. That is, $h(n)$ must be *non-over-estimating*; heuristic search with such an $h(n)$ is called A^*.

A^* expands the fringe of the search in contours of increasing f values until a goal is reached. A^* is the most efficient admissible graph search possible, in terms of number of nodes examined for a given $h(n)$.

In practice, one might sacrifice admissibility to have a more powerful heuristic function and reduce search time.

[3]Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann Publishers, 1980.

Informed Heuristic Functions

Given two heuristic functions $h_1(n)$ and $h_2(n)$, where both $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$, we say $h_2(n)$ is *more informed* than $h_1(n)$ if $h_2(n) > h_1(n)$ for all nodes n.

Given multiple admissible heuristic functions $h_1(n)$, $h_2(n)$, ..., the function $h(n) = max(h_1(n), h_2(n), ...)$ is also admissible and at least as informed as any of its component heuristic functions.

Since heuristic functions can easily be combined in the above manner, it is possible to learn heuristics from experience. The simplest case is remembering the true cost of solving a particular problem. More generally, features of particular problems can be learned along with costs.

Features of Heuristic Functions

A heuristic function $h(n)$ satisfies the *monotone restriction* if for all nodes n_i and n_j,

$$h(n_i) \leq h(n_j) + c(n_i, n_j)$$

If $h(n)$ satisfies this restriction (similar to the triangle inequality), then a node never has to be moved from `closed` back to `open`. In this case, `closed` could be eliminated, saving storage.

The effectiveness of a heuristic can be expressed as the *effective branching factor*, b^*. b^* is the inferred branching factor that would produce the actual number of nodes searched at the solution depth d.

Heuristic Search Handles Local Maxima

A barrier in the route-finding space creates a local maximum where hill-climbing would get stuck. Heuristic search will widen the search until it gets around the barrier.

$h = 0.9 * distance$, with barrier.

Iterative Deepening A* (IDA*)

Iterative Deepening A* (IDA*)[4] combines a depth-first, iterative-deepening style of search with the use of heuristic functions as in A*.

1. Initially, set $threshold = h(start)$.
2. Perform a depth-first search, failing at any node n for which $g(n) + h(n) > threshold$.
3. If no solution is found, increase $threshold$ to the minimum value of $g(n) + h(n)$ that was over $threshold$ in the previous search, and retry.

Adv.:

- Low storage requirement: $O(depth)$.
- Makes use of heuristic information: fast.
- Optimal solution if $h(n)$ is admissible.

Dis.:

- Redo search work that was done previously.
- If there are multiple paths to a node, IDA^* will re-search the successors of that node (A^* would not).

[4]Korf, R. E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, vol. 27, no. 1 (Sept. 1985).

Beam Search: MA^* and SMA^*

For some problems, A^* produces too large an `open` list. IDA^* may suffer from keeping *too little* information: in a search graph with many paths to a given node, IDA^* must re-search from that node.

One solution is to limit the size of `open`; this is called a *beam search*. The algorithms MA^* and SMA^* remove the worst and oldest node from `open` when a new space is needed, while choosing the best and newest node to expand at each step.

Beam search provides the directedness of depth-first search while maintaining several open lines of search (the width of the beam). For some applications (particularly those dealing with continuous time functions, such as signal understanding), it is necessary to prune less-promising search paths in order to keep storage and time requirements manageable. At the same time, it may be necessary to carry more than one line of search in case the current "best" line is not the overall best path.

Forward vs. Backward Search

It may be advantageous to search forward (from starting state towards the goal) or backwards (from goal to the starting state). Criteria include:

- Number of initial and goal states.
- Direction of greatest branching factor.

In many AI problems, *backward* reasoning is strongly preferred. The reason is that starting from the goal causes many variables to be bound to constants, greatly reducing the branching factor in the backward direction.

Bidirectional search, searching in both directions until the fringes of the searches meet in the middle, can save time: $2 * b^{d/2} << b^d$. However, this may take a lot of storage.

Search Tree vs. Search Graph

Many of the search algorithms can be written so that they consider the search graph to be a tree, even if it actually is a graph.

Advantages:

1. Simpler programming.
2. Saves computer time required to find duplicated states.
3. Saves storage: no need to store previous states.

Disadvantage:

1. Larger search space because the same node (and its descendants) are searched multiple times.

If the problem has a high degree of symmetry or high probability of duplicated states, detecting duplicates may have large savings. If the probability of duplicates is small, treating the graph as a tree is easier and may be faster.

Problem Reduction Search

Problem reduction search is a basic problem-solving technique of AI. It involves reducing a problem to a set of easier subproblems whose solutions, if found, can be combined to form a solution to the hard problem. Such a search is easily written as a recursive program in Lisp:

1. If the given problem is a *primitive subproblem* (one that can be solved by a known technique), return the solution to it.
2. Otherwise, try breaking the given problem into sets of simpler *subproblems*, and call the program recursively to try to solve the subproblems .
3. If a set of subproblems is found such that all the subproblems can be solved, *combine* the subproblem solutions in an appropriate way to form the solution to the current problem.

Problem Reduction Search: Flowchart

Problem Reduction Representations

The given problem is reduced to a set of simpler *subproblems*, the solution of which will allow the original problem to be solved.

This reduction is applied recursively until *primitive subproblems* which are immediately solvable are reached. The resulting structure is called an *AND/OR graph* .

- **AND Node:** All subproblems must be solved in order to solve the main problem. An arc is drawn across branches of an AND node .

- **OR Node:** Solution of any subproblem will solve the main problem. A state space search graph consists entirely of OR nodes.

AND/OR Graph Example

Solution to an AND/OR Graph

- A successful terminal node is a solved node.
- If a nonterminal OR node has any solved successors, then it is a solved node.
- If all of the successors of a nonterminal AND node are solved, then it is a solved node.
- The problem is solved if the start node is solved.

A *solution graph* is a subgraph of solved nodes that demonstrates that the start node is solved.

Search of an AND/OR Graph

An AND/OR Graph is searched recursively until the root node is solved. To solve a node, attempt to solve each successor of a nonterminal node sequentially.

- If any successor of an AND node fails, the AND node fails immediately.
- If any successor of an OR node succeeds, the OR node succeeds immediately.

The search time may be improved by ordering consideration of the nodes:

- To solve an AND node, try first to solve those successors that are most likely to fail (so the search can fail early).
- To solve an OR node, try first to solve those successors that are most likely to succeed (so the search can succeed early).

Counterexamples

In theorem proving, it may be possible to use a *model* of the theorem to be proved to prune the search space.

H. Gelernter[5] used a diagram of a geometry theorem as a filter. If a subgoal to be proved is true in the diagram, that may be just a coincidence. However, if a subgoal is false in the diagram, it cannot be part of any general theorem; thus, the subgoal can be failed at once.

[5]H. Gelernter, "Realization of a Geometry-Theorem Proving Machine", in E. A. Feigenbaum and J. Feldman, *Computers and Thought*, McGraw-Hill, 1963, pp. 134-152.

Searching Game Trees

The games we will consider are:

- *two-person:* there are two players.
- *perfect information:* both players have complete information about the state of the game. (Chess has this property, but poker does not.)
- *zero-sum:* if we count a win as +1, a tie as 0, and a loss as -1, the sum of scores for both players is always zero.

Examples of such games are chess, checkers, and tic-tac-toe.

Game Trees

The sequence of states formed by possible moves is called a game tree ; each level of the tree is called a *ply*. We call the two players Max (us) and Min (the opponent).

The game tree must be an AND/OR tree. We can make any move we wish; thus, if any move is a winner, we win, and we have an OR node. However, we can win on an opponent's move only if we can win for every possible move of the opponent; thus, the opponent's move is an AND node.

Each node of a game tree represents the problem of winning for Max from the corresponding board position. A winning strategy corresponds to a solution tree .

Static Evaluation Functions

A *static evaluation function* evaluates a position without doing any search.

Example 1: Chess

e(p) = (sum of values of Max's pieces)
- (sum of values of Min's pieces)
+ k * (degree of control of the center)

Example 2: Tic-tac-toe

e(p) = nrows(Max) - nrows(Min)
where nrows(i) is the number of complete rows, columns, or diagonals that are still open for player i.

Minimax Evaluation

We could play in hill-climbing fashion by making the move with the highest static value; but this produces weak play by failing to take the future into account.

A better way is to search to a depth bound, evaluate positions statically at the bound, and "back up" the values to the top using the *minimax algorithm*:

- The value of an OR node (Max's move) is the maximum of its successors' values. (Max can choose any move, hence will choose the best one.)
- The value of an AND node (Min's move) is the minimum of its successors' values. (Min will choose the move that is best for Min (worst for Max).)

Alpha-Beta Search

As Minimax search proceeds, we could mark each node with an inequality, $\geq max$ at an OR node or $\leq min$ at an AND node, representing that the final value is bounded by the values seen so far.

If two nodes in the hierarchy have incompatible inequalities (no possible overlap), then we know that the node below will not be chosen, and we can stop search.

Implementing Alpha-Beta Search

Alpha-beta search is easily implemented by adding the α and β parameters to a depth-first minimax search. The alpha-beta search simply quits early and returns the current value when the α or β threshold is exceeded.

Alpha-beta search performs best if the best moves (for Max) and worst moves (for Min) are considered first; in this case, the search complexity is reduced to $O(b^{d/2})$.

For games with high symmetry (e.g. chess), a *transposition table* (*cf.* Closed list) containing values for previously evaluated positions can greatly improve efficiency.

Alpha-Beta Search Example

Game Tree Search

Bounded depth-first search is usually used, with the alpha-beta algorithm, for game trees. However:

1. The depth bound may stop search just as things get interesting (e.g. in the middle of a piece exchange in chess [6]). For this reason, the depth bound is usually extended to the end of an exchange.
2. The search may tend to postpone bad news until after the depth bound: the *horizon effect* .

[6] Hsu *et al.*, "A Grandmaster Chess Machine", *Scientific American*, vol. 263, no. 4 (Oct. 1990), pp. 44-50.

Samuel's Checkers Program

Arthur Samuel wrote an early (1959) program that played Checkers at championship level. This program had several interesting features:

1. The program learned its heuristic function from experience playing humans and itself. The heuristic function was a weighted average of several features (e.g. piece advantage, mobility, control of the center). Coefficients of features were "rewarded" or "punished" depending on whether the program won or lost.
2. If the program wins (or loses), it is difficult to determine which move(s) were responsible and should be rewarded or punished. This is called the *credit assignment problem* .
3. Low-pass filtering of coefficients was used to smooth out the effects of unjust rewards and punishments.

General Problem Solver (GPS)

GPS (Newell, Shaw, Simon) was an attempt to construct a general problem solving mechanism that could solve problems in a new area given domain-specific knowledge about that area.

GPS is a generalized state-space search mechanism. It has the following components:

1. A set of states and operators that change states. Specification of start and goal states.
2. A procedure for identifying *differences* between states.
3. A *table of connections* that connects observed differences with operators that may be relevant for reducing those differences.

GPS works by determining the difference between the current state and the goal state and selecting operators relevant to reducing that difference. Hopefully this will give direction to the search and make it more efficient than a blind state-space search.

Problems with GPS

GPS did solve problems in several different domains, but was not as useful as had been hoped. Problems included:

1. Keeping a complete world state at each step takes a lot of storage. This restricted GPS to small problems.
2. The "difference" between the current state and the goal may not be an adequate guide to action. (Consider chess: the goal of checkmating the opponent doesn't tell how to play the opening.)
3. Accomplishing a subgoal might undo a higher-level goal. This could lead to loops.
4. We often want to achieve multiple goals simultaneously. ("Climb Mt. Everest and live to tell about it.")

Searching in Abstraction Spaces

The effective depth of a search tree may be reduced by doing one search in an abstract space to get a rough solution, then doing another search in the actual space to refine the solution.

Example: Route Finding Search for a path between two locations first on a coarse map that includes only major cities and interstate highways; then refine paths to get to the interstate highways.

Advantage: Can make an otherwise very large search tractable.

Disadvantage: The path found may be sub-optimal, depending on how good the abstraction is.

Genetic Algorithms

Genetic algorithms[7] mimic biological processes of evolution ; they can be used to find solutions in domains where:

- A solution consists of values for several independent variables.
- Some variables may have discrete values; others may have continuous numerical values.
- A candidate solution can be evaluated by some evaluation function.

Example: An animal might have variables `color`, `percent-body-fat`, `length-of-legs`.

It would be possible to use search on the discrete variables and hill-climbing on the continuous variables, but often the search space is too large.

[7] John R. Koza, Martin A. Keane and Matthew J. Streeter, "Evolving Inventions", ¡I¿ Scientific American¡/I¿, vol. 288, no. 2, Feb. 2003¡/A¿]

John R. Koza, Martin A. Keane and Matthew J. Streeter, "Evolving Inventions", *Scientific American*, vol. 288, no. 2, Feb. 2003

Outline of Genetic Algorithm

The genetic algorithm operates on a population of candidate solutions.

1. Initialize the set of candidate solutions.
2. Evaluate each solution in the population.
3. Solutions that are good are allowed to reproduce, with mutations (changes to variable values) in some copies; the amount of reproduction can depend on how good a solution is. (Koza uses 9% clones of the best in the old population, 90% recombinations, and 1% mutations.)
4. Poor solutions can be eliminated to keep the total population constant.
5. If the population is dominated by one solution, stop and return that solution; otherwise, go to step 2.

Compare: hill climbing, beam search.

Constraint Satisfaction Problems

Constraint satisfaction problems (CSP) involve assigning values to variables X_i, each of domain D_i, such that all of a set of constraints C_j are satisfied.

- Cryptarithmetic: `SEND + MORE = MONEY`
- n Queens
- Map coloring
- Crossword puzzles, Scrabble, logic puzzles
- Boolean satisfiability (SAT)
- Scheduling
- VLSI chip layout
- Building design

CSP as Backtracking Search

When the domains D_i are finite, CSP problems can be solved easily using search: at each level of the search tree, try each possible value of one variable; fail if a constraint is violated.

While a simple search always works, the time required is combinatoric. It is possible to do much better by making use of heuristics based on the structure of the problem:

- Choose which variable to instantiate next
- Choose which value to try first
- Propagate constraints

Constraint Network

Constraints can be expressed as a network, where nodes are linked if they are related by a binary constraint: $Start(job_2) \geq Start(job_1) + 5$.

Higher-order constraints can be expressed by multiple binary constraints (by introducing additional variables) or represented as a hypergraph.

Example: [8]

In this example, the nodes for states are linked if the states are adjacent on the map; links represent $\neq$ constraints.

[8] Russell & Norvig, Fig. 5.1.

Optimizing Search

- Choose the most constrained variable first.
  - Smallest set of remaining possible values
  - Largest number of arcs to other nodes

  The most constrained variable (smallest number of possible choices) gives the smallest branching factor at the top of the tree; a choice for it will constrain other variables, reducing their branching factors also.
- Choose the least constraining variable value. A more constraining variable value is more likely to have all failures below it.

Constraint Propagation

As in Waltz filtering, constraint propagation can greatly reduce the size of a CSP search:

- Initialize nodes to sets of all values in their respective domains.
- When the possible values of a node constrain the possible values of its neighbors, remove impossible values from the sets for the neighbors.

  An arc (X, Y) is *arc-consistent* if for every value $x \in X$, there is some value $y \in Y$ that satisfies the constraint represented by the arc. *k-consistency* follows multiple arcs to check for consistency (arc consistency is 2-consistency).

Some search may be left after constraint propagation, but the amount of search can be greatly reduced.

For scheduling problems, possible value sets can be represented as ranges $[min, max]$; constraints can allow the range limits to be reduced.

Other Techniques and Kinds of CSP

- Dependency-directed Backtracking can help prevent *thrashing* due to repeated choice of incompatible variable values.
- Linear Programming (LP) can solve optimization problems where the constraints are linear inequalities.
- Hill climbing: Generate a complete assignment of variables (which probably violates some constraints), improve it by modifying variables one at a time using a heuristic. The *min-conflicts* heuristic changes a variable to the value that minimizes the number of constraint violations that involve that variable.

Difficulty of CSP

Most CSP's are easy to solve.

- Easy problems: mild constraints, many solutions.
- Impossible problems: harsh constraints, no solutions, search fails quickly.

The most difficult CSP's are those in a narrow range where the problem is both difficult and just barely solvable.

DENDRAL (Buchanan and Feigenbaum)

Dendral performs structure elucidation in organic chemistry based on data from a mass spectrometer (which bombards chemical samples with electrons, breaking atomic bonds). Fragments are collected by charge/mass, and relative abundance is plotted as a histogram. The goal is to uniquely identify the chemical structure.

Given:

- Empirical formula for a chemical compound, e.g. $C_8H_{16}O$
- Mass spectrum for that compound, e.g.

Determine: structure of compound: 3-octanone

```
          O
          |
CH3-CH2-C-CH2-CH2-CH2-CH2-CH3
```

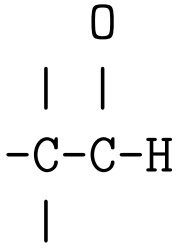
Underlying State Space:

- States: Molecular substructures containing atoms from the Empirical Formula.
- Start State: One atom
- Operators: Add an atom to some unbound valence binding of the current molecular structure.
- Goal States: Molecules which:
  - Have all bindings filled
  - Use all the atoms in the input formula
  - Are chemically stable
  - Are plausible for the given mass spectrum.

Problem: This search space is much too large to search exhaustively.

Ways to Reduce Search Space: Heuristics

- Careful design of the move generator so that equivalent structures are not generated again:
  CH3-CH2- vs. -CH2-CH3
- Rules that detect particular substructures in the mass spectrogram predict some parts of the molecular structure; these parts are put on a GOODLIST and are generated first.
  Rule: M-44 is a high peak, and 44 is a high peak
  Prediction: aldehyde:



- A BADLIST contains substructures which are unstable or could not be present because their signatures do not appear in the mass spectrogram: H-C-N=O, O-O, S-S-S are forbidden substructures because they are unstable. Any move which results in a BADLIST substructure is immediately detected and eliminated, pruning the whole search tree below that node.

Goal Testing: The restrictions on the generation of structures by the search process are so strict that only a few dozen potential goal nodes are generated in most cases. Another set of rules (which simulate the behavior of molecules in a mass spectrometer) is used to predict the mass spectrum for each potential goal structure. All possible structures whose predicted spectrum reasonably matches the input spectrum are returned as answers.

Uses of DENDRAL:

- Determining unknown molecular structures.
- Checking previously published structures.
- META-DENDRAL infers new rules from the structures found by DENDRAL. These rules have been published, thus contributing to the science of mass spectrometry.

Search as a Basic Technique

Search should be regarded as a basic technique of Computer Science, useful for many areas where no nice algorithm exists.

Example:
William Wulf used search to automatically generate code generators for new CPUs for an Ada compiler.

- **Goal:** an instruction sequence that accomplishes what the generated code is supposed to do (e.g., add `A` to `B` and put the result in `C`).
- **Operators:** Instructions of the target CPU.

Search was used to find a minimum-cost sequence of machine instructions that would accomplish the desired task.

Where Search Should Fit in an AI System

We can identify two extremes in the use of search:

- Traditional programming (no search): *inflexible.*
- Doing everything with search: *too slow.*

The ideal is to:

- Replace search by *knowledge* when possible.
  - Heuristic function is one kind of knowledge.
  - Experts vs. novices: Experts "just do the right thing"; novices search.
- Fall back on search when knowledge is insufficient.

Knowledge Representation and Reasoning

Much intelligent behavior is based on the use of knowledge; humans spend a third of their useful lives becoming educated. There is not yet a clear understanding of how the brain represents knowledge.

There are several important issues in knowledge representation:

- how knowledge is *stored*;
- how knowledge that is applicable to the current problem can be *retrieved*;
- how *reasoning* can be performed to derive information that is implied by existing knowledge but not stored directly.

The storage and reasoning mechanisms are usually closely coupled.

Representation Hypothesis

A central tenet of A.I. is the *representation hypothesis* that intelligent behavior is based on:

- *representation* of input and output data as symbols in a physical symbol system[9]
- *reasoning* by processing symbol structures, resulting in other symbol structures.

A central problem of A.I. is to determine what the symbolic representations and reasoning processes should be.[10]

[9]Newell, A., Physical Symbol Systems, *Cognitive Science*, 1980, *4*, 135-183.
[10]Diagram by John Sowa, from "The Challenge of Knowledge Soup," 2005.

Computation as Simulation

It is useful to view computation as simulation, *cf.*: *isomorphism of semigroups*.[11]

```
(princ-to-string (+ (read-from-string "2")
                    (read-from-string "3")))
"5"
```

[11]Preparata, F. P. and Yeh, R. T., *Introduction to Discrete Structures*, Addison-Wesley, 1973, p. 129.

Alternatives to the Representation Hypothesis

Not everyone accepts the representation hypothesis. Some alternatives include:

- Analog information, as in homeostatic control (e.g., regulation of body temperature).
- Special-purpose hardware, such as line detectors in early vision processing.
- Neural networks with weights on the connections between neurons. It may be difficult or impossible to identify "symbols" in such a network; the knowledge is represented by the totality of the set of weights.
- Holograms and various mystical alternatives.

Critics include Brooks,[12] McDermott, Dreyfus, Searle.

[12]Brooks, Rodney A., "Intelligence without representation", *Artificial Intelligence* 47 (1991), 139-159.

Kinds of Knowledge

Several kinds of information need to be represented:

Long-term Knowledge: This is accumulated knowledge about the world. It can include simple data, general rules (every person has a mother), programs, and heuristic knowledge (knowledge of what is likely to work). The collection of long-term knowledge is often called a *knowledge base* (KB). Human long-term memory seems unlimited, but writing to it is slow.

Current Data: A representation of the facts of the current situation. Human short-term memory is very limited (7 ± 2 items).[13]

Conjectures: Courses of action or reasoning that are being considered but are not yet final.

These will be represented in a *knowledge representation language*. Questions that are not directly in the KB may be answered by *inference*.

[13]Miller, George A., "The magical number seven, plus or minus two: some limits on our capacity for processing information", *Psychological Review* vol. 63, pp. 81-97, 1956.

Knowledge Representation System

A knowledge representation system will include ways to store knowledge, ways to add new knowledge, and ways to query the knowledge. We can think of the interface as being two procedures:

- $Tell(fact)$
- $Ask(question)$

$Tell$ and Ask will be front-end programs that access a database of facts in some knowledge representation language.

Either $Tell$ or Ask (or both) may do *inference* :

- $Ask(question)$ may do *backward* inference when the answer is implied but not explicit in the KB.
- $Tell(fact)$ may do *forward* inference to derive additional facts from what it is told.

Knowledge Representation

Knowledge representation is a central problem because A.I. attempts to deal with much more complex and less well-structured data than most computer programs.

There are two major classes of knowledge representation methods that are used in A.I.:

- **Logic:** methods based on mathematical logic, i.e. first-order predicate calculus.
- **Frames:** methods based on networks of nodes, which represent objects or concepts, and labeled arcs, which represent relations among nodes.

The Logic and Frame methods have sometimes been viewed as competitive, and their proponents have fought. Each method has some valuable features. More recently, some have produced representation systems that combine the good features of both.

Symbolic Representation

A.I. programs primarily use *symbolic* representations: collections of symbols that represent:

- Objects.
- Properties of objects.
- Relationships among objects.
- Rules about classes of objects.

Retrieval: Matching Problem

A central issue in knowledge representation is *retrieval*: it is unlikely that the current problem will exactly match the stored form of knowledge. How can we tell what part of our knowledge is relevant?

Knowledge:

```
JAPAN BOUGHT 2,000,000 BARRELS OF OIL FROM IRAN.
```

Possible Questions:

```
DID JAPAN BUY 2,000,000 BARRELS OF OIL FROM IRAN?

DID JAPAN BUY 1,000,000 BARRELS OF OIL FROM IRAN?

DID JAPAN BUY OIL FROM IRAN?

TO WHOM DID IRAN SELL OIL?

WHAT ARE MAJOR JAPANESE IMPORTS?

WHAT US ALLIES WOULD BE CRITICALLY AFFECTED
BY BLOCKAGE OF MIDDLE EAST OIL SHIPMENTS?
```

Knowledge Representation Methods

Predicate Calculus

- Strengths:
  - Logical power
  - Mathematical foundation

  Weaknesses:
  - Speed
  - Rigidity

Frames

- Strengths:
  - Speed
  - Defaults
  - Procedural attachment
- Weaknesses:
  - Logical power

Knowledge Representation: Hard Problems

Defaults

- Necessary: A.I. programs seldom get good, complete data.
- Sometimes wrong: may have to retract assumptions, conclusions based on them.

Time-Varying Data

- How to keep a current model of the world as *some* things change (the *frame problem*).

Uncertainty

- "60% of patients with symptoms A, B, and C have disease D."

Logic for Artificial Intelligence

Mathematical logic is an important area of AI:

- Logic is one of the major knowledge representation and reasoning methods.
- Logic serves as a standard of comparison for other representation and reasoning methods.
- Logic has a sound mathematical basis.
- The PROLOG language is based on logic.
- Those who fail to learn logic are doomed to reinvent it.

The form of logic most commonly used in AI is *First-Order Predicate Calculus* (*FOPC* or just *PC*).

Logical Representation

Mathematical logic requires that certain strong conditions be satisfied by the data being represented:

- **Discrete Objects:** The objects represented must be discrete individuals: people, trucks, but not 1000 gallons of gasoline.
- **Truth or Falsity:** Propositions must be entirely true or entirely false; inaccuracy or degrees of belief are not representable.
- **Non-Contradiction:** Not only must data not be contradictory, but facts derivable by rules must not contradict.

These strict requirements give logic its power, but make it difficult to use for many practical applications.

Propositional Logic

Formulas in propositional logic are composed of:

- *Atoms* or *propositional variables* : P, Q, S
- Connectives (in order of precedence):

Negation	$\neg$ or $\sim$	not
Conjunction	$\wedge$	and
Disjunction	$\vee$	or
Implication	$\rightarrow$ or $\supset$	implies or if ... then
	$\leftrightarrow$	iff (if and only if)

- Constants: True $True$ or filled-in box
  False ("box")

Interpretation in Propositional Logic

An *interpretation* of a propositional logic formula is an assignment of a value (true or false) to each atom. There are 2^n possible interpretations of a formula with n atoms. Although this is large, it is finite; thus, every question about propositional logic is *decidable*.

Terminology:

- A formula is *valid* if it is true under every possible interpretation. Otherwise, it is *invalid*.
- A formula is *consistent* or *satisfiable* if it is true under some interpretation. If it is false under every interpretation, it is *inconsistent* or *unsatisfiable*.

Clearly, a formula G is valid iff $\neg\, G$ is inconsistent.

If a formula F is true under an interpretation I, I is a *model* for F.

Two formulas F and G are *equivalent* if they have the same values under every interpretation: $F \leftrightarrow G$.

Equivalent Formula Laws

- Implication:
  $F \to G = \neg F \vee G$
  $F \leftrightarrow G = (F \to G) \wedge (G \to F)$

- De Morgan's Laws:
  $\neg(F \vee G) = \neg F \wedge \neg G$
  $\neg(F \wedge G) = \neg F \vee \neg G$

- Distributive:
  $F \vee (G \wedge H) = (F \vee G) \wedge (F \vee H)$
  $F \wedge (G \vee H) = (F \wedge G) \vee (F \wedge H)$

Inference Rules

- Modus Ponens: $\frac{P,\ P \to Q}{Q}$

Ways to Prove Theorems

Given a set of facts (*ground literals*) and a set of rules, a desired theorem can be proved in several ways:

- **Truth Table:** Write $Premises \rightarrow Conclusion$ and show that this sentence is true for every interpretation. This is also called *model checking*.
- **Algebra:** Write $Premises \rightarrow Conclusion$ and reduce it to $True$ using laws of Boolean algebra.
- **Backward Chaining:** Work backward from the desired conclusion by finding rules that could deduce it; then try to deduce the premises of those rules.
- **Forward Chaining:** Use known facts and rules to deduce additional known facts. If the desired conclusion is deduced, stop.
- **Resolution:** This is a proof by contradiction. Using ground facts, rules, and the *negation* of the desired conclusion, try to derive "box" (false or contradiction) by resolution steps.

Backward Chaining

Suppose that we have formulas:[14]
A
B
D
$A \wedge B \rightarrow C$
$C \wedge D \rightarrow E$

A conclusion E can be proved recursively:

1. First check whether the desired conclusion is in the database of facts. If so, return True.
2. Otherwise, for each rule that has the desired conclusion (right-hand side), call the algorithm recursively for each item in the premise (left-hand side). If all of the premises are true, return True.
3. Otherwise, return False.

In this example, we would know that E is true if we knew that C and D were true; we would know that C is true if we knew A and B; A and B are in the database, so C must be true; and D is in the database, so E is true.

With careful implementation, backchaining can run in linear time.

[14]Backward chaining only works for *Horn clauses*, which have at most one positive literal.

Backchaining Example[15]

The function `(backchain goal rules facts)` tries to prove a goal given sets of rules and facts. `goal` is a symbol (atom or propositional variable). `facts` is a list of atoms that are known to be true. `rules` is a list of rules of the form `(`*concl* $prem_1 ... prem_n$`)`; each rule states that the conclusion is true if all of the premises are true. For example, a rule $A \wedge B \rightarrow C$ would be written `(c a b)`.

`backchain` works as follows: if the goal is known to be a fact, return true. Otherwise, see if some rule has the goal as conclusion and has premises that are true (using backchain).

```
(defun backchain (goal rules facts)
 (or (member goal facts)
     (some #'(lambda (rule)
               (and (eq (car rule) goal)
                    (every #'(lambda (subgoal)
                               (backchain subgoal
                                    rules facts))
                           (cdr rule))))
           rules)) )

>(backchain 'e '((c a b) (e c d)) '(a b d))
T
```

[15] file `backch.lsp`

Forward Chaining

Suppose that we have formulas such as the following:
A
B
D
$A \wedge B \rightarrow C$
$C \wedge D \rightarrow E$

From A and B , it is possible to derive C and save it as a new fact; then from C and D , E can be derived.

The forward chaining algorithm is as follows:

When a new fact is presented to the database manager,

1. Add the new fact to the database.
2. For each rule that has the new fact as part of its premise, if the rest of the premise is true, add the conclusion to the database.

This cycle is repeated until activity stops or until the desired fact is added to the database.

With careful implementation, forward chaining can run in linear time.

Forward Chaining Example[16]

```
(defvar *db*)          ; atomic facts
(defvar *rules*)

(defun assrt (fact)    ; assert a new fact
  (or (member fact *db*)  ; already known
      (progn
        (push fact *db*)  ; add fact to *db*
        (dolist (rule *rules*)
          (if (and (member fact (cdr rule))
                   (every #'(lambda (x)
                              (member x *db*))
                          (cdr rule)))
              (assrt (car rule)) ) ) ) ))

(setq *db* '())
(setq *rules* '((c a b) (e c d)))
>(assrt 'a)
>(assrt 'b)
>*db*
(C B A)
>(assrt 'd)
>*db*
(E D C B A)
```

[16] file `forwch.lsp`

Resolution

Suppose that we have formulas such as the following:
A
B
D
$\neg A \vee \neg B \vee C$ (same as $A \wedge B \rightarrow C$)
$\neg C \vee \neg D \vee E$ (same as $C \wedge D \rightarrow E$)

A desired conclusion, say E, is negated to form the hypothetical fact $\neg E$; then the following algorithm is executed:

1. Choose two clauses that have *exactly one* pair of literals that are complementary (have different signs).
2. Produce a new clause by deleting the complementary literals and combining the remaining literals.
3. If the resulting clause is empty ("box"), stop; the theorem is proved by contradiction. (If the negation of the theorem leads to a contradiction, then the theorem must be true.)

This assumes that the premises are consistent.

Resolution for Propositional Calculus[17]

```
; rep.: p -> q is (~p v q) or  ((not p) q) .
; example: (resolve '((not p) q) '(r (not q)))

(defun resolve (ca cb)
  (let (tmp)
    (if (setq tmp (resolver ca cb))
        (unless (resolver cb ca) tmp)
        (resolver cb ca)) ))

; resolve neg. of ca with pos of cb:

(defun resolver (ca cb)
  (let (pairs)
    (dolist (lit ca)
      (if (and (consp lit)
               (eq (car lit) 'not)
               (member (cadr lit) cb))
          (push lit pairs)) )
    (if (and pairs (null (cdr pairs)))
        (or (union (remove (car pairs) ca)
                   (remove (cadar pairs) cb)
                   :test #'equal)
            'box)) ))
```

[17] file `resolv.lsp`

Normal Forms

A *literal* is an atom or negation of an atom: P or $\neg P$.

A formula F is in *conjunctive normal form* (CNF) if F is of the form $F = F_1 \wedge F_2 \wedge ... \wedge F_n$ where each F_i is a disjunction of literals. Example: $(\neg P \vee Q) \wedge (P) \wedge (\neg Q)$

CNF is used for resolution, so it is the one we will use. There is also a *disjunctive normal form*.

Logical Consequence

We say that a set of formulas $F_1, F_2, ..., F_n$ *entails* a formula G, written $F_1 \wedge F_2 \wedge ... \wedge F_n \models G$, or that G is a *logical consequence* of the F_i, iff G is true in any interpretation where $F_1 \wedge F_2 \wedge ... \wedge F_n$ is true. This is equivalent to saying that $F_1 \wedge F_2 \wedge ... \wedge F_n \rightarrow G$ is valid.[18]

$F_1 \wedge F_2 \wedge ... \wedge F_n \rightarrow G$ is valid iff $F_1 \wedge F_2 \wedge ... \wedge F_n \wedge \neg G$ is inconsistent.

Proof: $\neg(F \rightarrow G) = \neg(\neg F \vee G) = (F \wedge \neg G)$.

Note that *anything* is a logical consequence of (or an inconsistent set of clauses).

Proof: $\rightarrow X = \neg \vee X = True \vee X = True$.

Thus, if the result that is derived from a set of clauses is to be meaningful, the set of clauses must be consistent.

We could prove that a formula G follows from formulas F_i by truth table, or by algebraically reducing one of the logical consequence formulas to true (false) as in the example above.

[18]The fact that $\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid is called the *deduction theorem*.

Resolution for Propositional Calculus

Resolution[19] is a method of proof by contradiction.

Given a set of premises $F_1 \wedge F_2 \wedge ... \wedge F_n$ and a desired conclusion G, we prove that $F_1 \wedge F_2 \wedge ... \wedge F_n \wedge \neg G$ is inconsistent. This is done by adding new clauses H_i, each of which is a logical consequence of the existing clauses, to the set. If we can add as a new clause, then the whole formula collapses to false and is thus inconsistent.

Theorem: If H is a logical consequence of F, $F \wedge H = F$.
Proof: If F is false, then both sides are false. If F is true, then H must be true, so $F \wedge H$ is true.

Thus, we are justified in adding any logical consequence of our set of formulas to the set without changing its truth value.

[19] J. A. Robinson, A machine oriented logic based on the resolution principle, *Journal of the ACM*, **12**, No. 1, 1965.

Resolution Step for Propositional Calculus

A *clause* is a disjunction of literals (atoms or negations of atoms).

Select two clauses C_1 and C_2 that have *exactly one* atom that is positive in one clause and negated in the other.

Form a new clause, consisting of all literals of both clauses except for the two complementary literals, and add it to the set of clauses.

Theorem: The new clause produced by resolution is a logical consequence of the two parent clauses.

Proof: Let the parent clauses be $C_1 = L \vee C_1'$ and $C_2 = \neg L \vee C_2'$; the resolvent is $H = C_1' \vee C_2'$. Suppose that C_1 and C_2 are true in an interpretation I.
Case 1: L = true in I. Then since $C_2 = \neg L \vee C_2'$, C_2' must be true in I and H is true in I.
Case 2: L = false in I. Then since $C_1 = L \vee C_1'$, C_1' must be true in I and H is true in I.

Resolution Step

If: $C_1 \wedge C_2 \rightarrow H$
then: $C_1 \wedge C_2 \wedge H = C_1 \wedge C_2$
but not: $C_1 \wedge C_2 = H$

Example:
$C_1 = (A \vee B)$
$C_2 = (\neg B \vee C)$
$H = (A \vee C)$

A	B	C	$C_1 = A \vee B$	$C_2 = \neg B \vee C$	$C_1 \wedge C_2$	$H = A \vee C$	$C_1 \wedge C_2 \wedge H$
0	0	0	0	1	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	1	0	0	1	0
1	1	1	1	1	1	1	1

When $C_1 \wedge C_2$ is false, it is a "don't care" for H.

Examples of Resolution Step

1. $F_1 = P \vee R$, $F_2 = \neg P \vee Q$.
   Resolvent: $R \vee Q$.
2. $F_1 = \neg P \vee Q \vee R$, $F_2 = \neg Q \vee S$.
   Resolvent: $\neg P \vee R \vee S$.
3. $F_1 = \neg P \vee Q$, $F_2 = \neg P \vee R$.
   Resolution not possible: no complementary literals.
4. $F_1 = \neg P \vee Q \vee S$, $F_2 = P \vee \neg Q \vee R$.
   Resolution not useful: two complementary literals.

Example of a proof by resolution:
Show that Q is a logical consequence of $P \wedge (P \rightarrow Q)$.

1.	P	premise
2.	$\neg P \vee Q$	premise
3.	$\neg Q$	negated conclusion
4.	Q	from 1 and 2
5.		from 3 and 4: Q.E.D.

Example: Propositional Calculus Resolution

Premises:

1. If it rains, the aquaphobes will not vote.
2. John will win only if the aquaphobes and vegetarians vote.
3. Either John or Peter will win, but not both.
4. (conclusion): If it rains, Peter will win.

1. $R \to \neg A$	1.	$\neg R \vee \neg A$
2. $J \to A \wedge V$	2.a.	$\neg J \vee A$
	2.b.	$\neg J \vee V$
3. $(J \vee P) \wedge \neg(J \wedge P)$	3.a.	$J \vee P$
	3.b.	$\neg J \vee \neg P$
4. $\neg(R \to P)$	4.a.	R
	4.b.	$\neg P$
	from 3.a., 4.b.:	5. J
	from 2.a., 5:	6. A
	from 6, 1:	7. $\neg R$
	from 4.a., 7:	8. Q.E.D.

Satisfiability Checking

Many problems in CS can be reduced to checking *satisfiability* of a propositional calculus formula.

Two efficient algorithms for satisfiability checking (SAT):

- The Davis-Putnam or DPLL algorithm uses heuristics for *early termination* (determining the value from a partially specified model), *pure symbols* (those that have the same sign in all clauses) and *unit clauses* (those with a single literal).

  The CHAFF inplementation of DPLL solves hardware verification problems with a million variables.

- The WalkSAT algorithm uses a combination of hill climbing (selecting a literal assignment that makes the most clauses true) and random steps.

Predicate Calculus (First-order Logic)

Propositional logic does not allow any reasoning based on general rules, so its usefulness is limited. Predicate calculus generalizes propositional logic with variables, quantifiers, and functions.

Formulas are constructed from:

- Predicates have arguments, which are terms: $P(x, f(a))$. Predicates are true or false.
- Terms refer to objects in the application domain:
  - Variables: x, y, z
  - Constants: $John, Mary, 3, a, b$. Note that a constant is generally capitalized in English: *Austin* can be a constant, but *dog* cannot.
  - Functions: $f(x)$ whose arguments are terms.
- Quantifiers: $\forall$ ("for all") and $\exists$ ("there exists" or "for some") quantify variables: $\forall x$, $\exists y$. If a variable is in the scope of a quantifier, it is *bound*; otherwise, it is *free*.

Overview of Predicate Calculus Resolution

Resolution provides a complete, uniform proof procedure for predicate calculus that is easily mechanized. A major benefit of resolution is that the complex formulas of predicate calculus are reduced to a few simple forms.

In order to perform resolution, we must first convert the given formulas into *standard form*:

1. First, a formula is converted into *prenex normal form*, in which all quantifiers are at the left.
2. *Skolemization* eliminates existential quantifiers: existential variables are replaced by constants or functions.
3. Universal quantifiers are eliminated; all remaining variables are assumed to be universally quantified.
4. The resulting formulas are converted to conjunctive normal form.

Prenex Normal Form

A predicate calculus formula is in prenex normal form if all the quantifiers are at the left.

A formula can be transformed to prenex normal form using the following identities. The form Qx is used for either $\forall x$ or $\exists x$.

$QxF[x] \vee G = Qx(F[x] \vee G)$, x not used in G.
$QxF[x] \wedge G = Qx(F[x] \wedge G)$, x not used in G.
$\neg \forall x F[x] = \exists x \neg F[x]$
$\neg \exists x F[x] = \forall x \neg F[x]$
$\forall x F[x] \wedge \forall x G[x] = \forall x(F[x] \wedge G[x])$
$\exists x F[x] \vee \exists x G[x] = \exists x(F[x] \vee G[x])$
$Q_1 x F[x] \vee Q_2 x G[x] = Q_1 x Q_2 z F[x] \vee G[z]$
$Q_1 x F[x] \wedge Q_2 x G[x] = Q_1 x Q_2 z F[x] \wedge G[z]$

Order of Quantifiers

The order in which quantifiers appear is very important; it must be maintained when converting a formula to prenex normal form.

Consider the ambiguous sentence, "Every man loves some woman." This sentence could be interpreted as:

1. For every man, there is some woman (depending on who the man is) whom the man loves. This would be written:

   $\forall x[Man(x) \rightarrow \exists y[Woman(y) \wedge Loves(x, y)]]$

   and Skolemized:

   $Man(x) \rightarrow$
   $[Woman(lover(x)) \wedge Loves(x, lover(x))]$

2. There is some woman (perhaps Marilyn Monroe) who is loved by every man. This would be written:

   $\exists y \forall x[Man(x) \rightarrow [Woman(y) \wedge Loves(x, y)]]$

   and Skolemized:

   $Man(x) \rightarrow [Woman(a) \wedge Loves(x, a)]$

Skolemization

Skolemization eliminates existential quantifiers by replacing each existentially quantified variable with a *Skolem constant* or *Skolem function.*

In effect, we are saying "If there exists (at least) one, give the algebraic name a to it." Having named the existential variable, we can eliminate the quantifier.

In general, an existential variable is replaced by a *Skolem function* of all the universal variables to its left. (A Skolem constant is a function of no variables.)

Each Skolem constant or function that is introduced must be a new one, distinct from any constant or function symbol that has been used already.

Example: $\exists x \forall y \forall z \exists w P(x, y, z, w)$
This is Skolemized as $P(a, y, z, f(y, z))$. $\exists x$ has no universals to its left, so it is Skolemized as a constant, a. $\exists w$ has universals y and z to its left, so it is Skolemized as a function of y and z.

After Skolemizing, universal quantifiers are eliminated; all remaining variables are understood to be universally quantified.

Standard Form

A formula is in *standard form* after it has been converted to prenex normal form, Skolemized, and converted into conjunctive normal form. The result is a conjunction of *clauses*, each of which is a disjunction of literals.

Example: "Every man loves some woman."

$\forall x[Man(x) \rightarrow \exists y[Woman(y) \wedge Loves(x, y)]]$

This is Skolemized as:

$Man(x) \rightarrow [Woman(lover(x)) \wedge Loves(x, lover(x))]$

Converted to CNF, it becomes:

$(\neg Man(x) \vee Woman(lover(x))) \wedge$
$(\neg Man(x) \vee Loves(x, lover(x)))$

Since we know where the $\wedge$ and $\vee$ are in CNF, we can eliminate them and represent clauses in list form in Lisp. Note that we have eliminated all of the algebraic structure in the formulas except for $\neg$.

```
( ( (not (man x)) (woman (lover x)) )
  ( (not (man x)) (loves x (lover x)) ) )
```

Proof by Contradiction

It is a theorem that a standard form of a formula F is inconsistent iff F is inconsistent.

Note that if F is not inconsistent, the standard form is not in general equivalent to F. This is because Skolemization replaces an existentially quantified variable by a single constant or Skolem function; if F is consistent, there might be many instances of the existential, not just one.

Herbrand's Theorem

A formula is inconsistent iff it is unsatisfiable for any interpretation over any domain. Since there are infinitely many domains and interpretations, it is important not to say what the domain and interpretation are, but to treat them algebraically.

Our algebraic domain is called the *Herbrand universe*; it consists of all the Skolem constants and all the Skolem functions applied to all constants, recursively. If there is one constant a and one function $f(x)$, the Herbrand universe will be $\{a, f(a), f(f(a)), ...\}$. The set of all predicates applied to members of the Herbrand universe is called the *Herbrand base*.

Herbrand's Theorem states that if a set of clauses S is unsatisfiable, there is a finite unsatisfiable set S' of ground instances of S, where the ground instances are obtained by substituting members of the Herbrand universe for variables in S.

Resolution for Predicate Calculus

The resolution step is still valid for predicate calculus, i.e., if clause C_1 contains a literal L and clause C_2 contains $\neg L$, then the resolvent of C_1 and C_2 is a logical consequence of C_1 and C_2 and may be added to the set of clauses without changing its truth value.

However, since L in C_1 and $\neg L$ in C_2 may contain different arguments, we must do something to make L in C_1 and $\neg L$ in C_2 exactly complementary.

Since every variable is universally quantified, we are justified in substituting any term for a variable. Thus, we need to find a set of substitutions of terms for variables that will make the literals L in C_1 and $\neg L$ in C_2 exactly the same. The process of finding this set of substitutions is called *unification*. We wish to find the *most general unifier* of two clauses, which will preserve as many variables as possible.

Example: $C_1 = P(z) \vee Q(z), C_2 = \neg P(f(x)) \vee R(x)$. If we substitute $f(x)$ for z in C_1, C_1 becomes $P(f(x)) \vee Q(f(x))$. The resolvent is $Q(f(x)) \vee R(x)$.

Examples of Unification

Consider unifying the literal $P(x, g(x))$ with:

1. $P(z, y)$: unifies with $\{x/z, g(x)/y\}$
2. $P(z, g(z))$: unifies with $\{x/z\}$ or $\{z/x\}$
3. $P(Socrates, g(Socrates))$: unifies, $\{Socrates/x\}$
4. $P(z, g(y))$: unifies with $\{x/z, x/y\}$ or $\{z/x, z/y\}$
5. $P(g(y), z)$: unifies with $\{g(y)/x, g(g(y))/z\}$
6. $P(Socrates, f(Socrates))$: does not unify: f and g do not match.
7. $P(g(y), y)$: does not unify: no substitution works.

Substitutions

A *substitution* t_i/v_i specifies substitution of term t_i for variable v_i. Unification will produce a set of substitutions that make two literals the same.

A substitution set can be represented as either *sequential* substitutions (done one at a time in sequence) or as *simultaneous* substitutions (done all at once). Unification can be done correctly either way.

We will assume a simultaneous substitution, using the Lisp function `sublis`. `(sublis` *alist form* `)` performs the substitutions specified by *alist* in the formula *form*. *alist* is of the form `((` *var* `.` *term* `) ... )`.

Suppose we want to substitute $\{a/x, f(b)/y\}$ in $P(x, y)$. In Lisp form, this is:

```
(sublis '((x . (a)) (y . (f (b))))
        '(p x y))

       = (P (A) (F (B)))
```

Unification Algorithm

The basic unification algorithm is simple. However, it must be implemented with care to ensure that the results are correct.

We begin by making sure that the two expressions have no variables in common. If there are common variables, substitute a new variable in one of the expressions. (Since variables are universally quantified, another variable can be substituted without changing the meaning.)

Imagine moving a pointer left-to-right across both expressions until parts are encountered that are not the same in both expressions. If one is a *variable*, and the other is a *term not containing that variable*,

1. substitute the term for the variable in both expressions,
2. substitute the term for the variable in the existing substitution set [20]
3. add the substitution to the substitution set.

[20] This is necessary so that the substitution set will be simultaneous.

Unification Implementation

1. Initialize the substitution set to be empty. We do this with the set `((t . t))`. A `nil` set indicates failure.
2. Recursively unify expressions:
   (a) Identical items match.
   (b) If one item is a variable v_i and the other is a term t_i not containing that variable, then:
      i. Substitute t_i/v_i in the existing substitutions.
      ii. Add t_i/v_i to the substitution set.
   (c) If both items are functions, the function names must be identical and all arguments must unify. Substitutions are made in the rest of the expression as unification proceeds.

Simple Unification Algorithm

```
(defun unify (u v) (unifyb u v '((t . t))))

; unify terms: subs list or NIL if failure
(defun unifyb (u v subs)          ; works if:
  (or (and (eq u v) subs)         ; identical vars
      (varunify v u subs)         ; u is a var
      (varunify u v subs)         ; v is a var
      (and (consp u) (consp v)    ; both are fns
           (eq (car u) (car v))   ;  with same name
           (unifyc (cdr u) (cdr v) subs))) ) ;args

(defun unifyc (args1 args2 subs) ; unify arg lists
  (if (null args1)               ; if args1 empty
      (if (null args2) subs)     ;    args2 must be
      (and args2 subs            ; unify first args
        (let ((newsubs (unifyb (car args1)
                               (car args2) subs)))
             (unifyc (sublis newsubs (cdr args1))
                     (sublis newsubs (cdr args2))
                     newsubs))) ))

(defun varunify (term var subs) ; unify with var
  (and var (symbolp var) (not (occurs var term))
       (cons (cons var term)
             (subst term var subs))))
```

```
>(unify-test '(p x) '(p (a)))
((X A) (T . T))

>(unify-test '(p (a)) '(p x))
((X A) (T . T))

>(unify-test '(p x (g x) (g (b))) '(p (f y) z y))

((Y G (B))  (Z G (F (G (B))))  (X F (G (B))) (T . T

>(unify-test '(p (g x) (h w)   w)
             '(p   y   (h y) (g (a))))
( (X A)  (W G (A))  (Y G (A))  (T . T))

>(unify-test '(p x (h (a)) (f x)) '(p (g y) y z))

( (Z F (G (H (A))))  (Y H (A))
  (X G (H (A)))      (T . T))

>(unify-test '(p (f x) (g (f (a))) x)
             '(p   y   (g    y)   (b)))
NIL
>(unify-test '(p x) '(p (a) (b)))
NIL
>(unify-test '(p x (f x)) '(p (f y) y))
NIL
```

Soundness and Completeness

The notation $p \models q$ is read "*p entails q*"; it means that q holds in every model in which p holds.

The notation $p \vdash_m q$ means that q can be derived from p by some proof mechanism m.

A proof mechanism m is *sound* if $p \vdash_m q \rightarrow p \models q$.

A proof mechanism m is *complete* if $p \models q \rightarrow p \vdash_m q$.

Resolution for predicate calculus is:

- *sound:* If is derived by resolution, then the original set of clauses is unsatisfiable.
- *complete:* If a set of clauses is unsatisfiable, resolution will eventually derive . However, this is a search problem, and may take a very long time.

We generally are not willing to give up soundness, since we want our conclusions to be valid. We might be willing to give up completeness: if a sound proof procedure will prove the theorem we want, that is enough.

Resolution Strategies

Many different strategies have been tried for selecting the clauses to be resolved. These include:

- *level saturation* or *two-pointer* method: the outer pointer starts at the negated conclusion; the inner pointer starts at the first clause. The two clauses denoted by the pointers are resolved if possible, with the result added to the end of the list of clauses. (There may be multiple resolvents of two clauses.) The inner pointer is incremented to the next clause until it reaches the outer pointer; then the outer pointer is incremented and the inner pointer is reset to the front. The two-pointer method is a breadth-first method that will generate many duplicate clauses.
- *set of support:* One clause in each resolution step must be part of the negated conclusion or a clause derived from it. This can be combined with the two-pointer method by putting the clauses from the negated conclusion at the end of the list. Set-of-support keeps the proof process focused on the theorem to be proved rather than trying to prove everything.

Resolution Strategies ...

- *unit preference:* Clauses are prioritized, with *unit clauses* (those with only one literal) preferred, or more generally, shorter clauses preferred. Our goal is to reach , which has zero literals and is only obtained by resolving two unit clauses. Resolution with a unit clause makes the result smaller.
- *linear resolution:* one clause in each step must be the result of the previous step. This is a depth-first strategy. It may be necessary to back up to a previous clause if no resolution with the current clause is possible.

Resolution Example[21]

1. No used car dealer buys a used car for his family.
2. Some people who buy used cars are absolutely dishonest.
3. Conclusion: Some absolutely dishonest people are not used car dealers.

1. $\forall x(U(x) \rightarrow \neg B(x))$
2. $\exists x(B(x) \wedge D(x))$
3. $\exists x(D(x) \wedge \neg U(x))$

1. $\neg U(x) \vee \neg B(x)$
2. a. $B(a)$ b. $D(a)$
3. $\neg(\exists x(D(x) \wedge \neg U(x)))$
   $\forall x \neg(D(x) \wedge \neg U(x))$
   $\forall x(\neg D(x) \vee U(x))$
   $\neg D(x) \vee U(x)$
4. (1, 2.a.) $\neg U(a)$
5. (3, 4) $\neg D(a)$
6. (2.b, 5.)

[21] Chang and Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

Resolution Example[22]

1. The customs officials searched everyone who entered the country who was not a V.I.P.
2. Some of the drug pushers entered the country, and they were only searched by drug pushers.
3. No drug pusher was a V.I.P.
4. Conclusion: Some of the officials were drug pushers.

1. $\forall x(E(x) \wedge \neg V(x) \rightarrow \exists y(S(x,y) \wedge C(y)))$
2. $\exists x(P(x) \wedge E(x) \wedge \forall y(S(x,y) \rightarrow P(y)))$
3. $\forall x(P(x) \rightarrow \neg V(x))$
4. $\exists x(P(x) \wedge C(x))$

[22]Chang and Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

Resolution as Syntax

Resolution is a *purely syntactic* uniform proof procedure. Resolution does not consider what predicates may *mean*, but only what logical conclusions may be derived from the axioms.

Advantage: Resolution is universally applicable to problems that can be described in first-order logic. Work on the theorem prover can be decoupled from any particular domain.

Disadvantage: Resolution by itself cannot make use of any domain-dependent heuristics. Despite many attempts to improve the efficiency of resolution, it often takes exponential time.

A contradiction in the axiom set allows anything to be "proved". Unfortunately, consistency of the axiom set is not decidable.

Natural Deduction

Natural deduction methods perform deduction in a manner similar to reasoning used by humans, e.g. in proving mathematical theorems.

Forward chaining and *backward chaining* are natural deduction methods. These are similar to the algorithms described earlier for propositional logic, with extensions to handle variable bindings and unification.

Backward chaining by itself is not complete, since it only handles *Horn clauses* (clauses that have at most one positive literal). Not all clauses are Horn; for example, "Every person is male or female" becomes $\neg Person(x) \vee Male(x) \vee Female(x)$ which has two positive literals. Such clauses do not support backchaining.

Splitting can be used with backchaining to make it complete. Splitting makes assumptions (e.g. "Assume x is Male") and attempts to prove the theorem for each case.

Backchaining Theorem Prover

```
   1.  ((FATHER (ZEUS) (ARES)))
   2.  ((MOTHER (HERA) (ARES)))
   3.  ((FATHER (ARES) (HARMONIA)))
   4.  ((PARENT X Y) (MOTHER X Y))
   5.  ((PARENT X Y) (FATHER X Y))
   6.  ((GRANDPARENT X Y) (PARENT Z Y) (PARENT X Z))

>(goal '(father x (harmonia)))
  1> (GOAL (FATHER X (HARMONIA)))
((X ARES))

>(goal '(parent z (harmonia)))
  1> (GOAL (PARENT Z (HARMONIA)))
    2> (GOAL (FATHER Z (HARMONIA)))
    <2 (GOAL ((Z ARES)))
((Z ARES))

>(goal '(grandparent x (harmonia)))
  1> (GOAL (GRANDPARENT X (HARMONIA)))
    2> (GOAL (PARENT Z (HARMONIA)))
      3> (GOAL (FATHER Z (HARMONIA)))
      <3 (GOAL ((Z ARES)))
    <2 (GOAL ((Z ARES)))
    2> (GOAL (PARENT X (ARES)))
      3> (GOAL (FATHER X (ARES)))
      <3 (GOAL ((X ZEUS)))
    <2 (GOAL ((X ZEUS)))
((X ZEUS))
```

```
   ; remove the father of ares
>(setf (get 'father 'ground) '(3))
(3)

>(goal '(grandparent x (harmonia)))
  1> (GOAL (GRANDPARENT X (HARMONIA)))
    2> (GOAL (PARENT Z (HARMONIA)))
      3> (GOAL (FATHER Z (HARMONIA)))
      <3 (GOAL ((Z ARES)))
    <2 (GOAL ((Z ARES)))
    2> (GOAL (PARENT X (ARES)))
      3> (GOAL (FATHER X (ARES)))
      <3 (GOAL NIL)
      3> (GOAL (MOTHER X (ARES)))
      <3 (GOAL ((X HERA)))
    <2 (GOAL ((X HERA)))
  <1 (GOAL ((X HERA)))
((X HERA))
```

Question Answering

Given a data base of facts (ground instances) and rules expressed in logic, we can pose questions in logic and answer them using resolution.

Example:

1. $\forall x \forall y \forall z [Parent(x, z) \quad \wedge \quad Parent(z, y) \quad \rightarrow Grandparent(x, y)]$
2. $\forall x \forall y [Father(x, y) \rightarrow Parent(x, y)]$
3. $\forall x \forall y [Mother(x, y) \rightarrow Parent(x, y)]$
4. $Father(Zeus, Ares)$
5. $Mother(Hera, Ares)$
6. $Father(Ares, Harmonia)$

Consider the question, "Who is a grandparent of Harmonia?" This can be expressed as $\exists x Grandparent(x, Harmonia)$; this is negated to be $\forall x \neg Grandparent(x, Harmonia)$ ("Nobody is the grandparent of Harmonia.").

We then can use resolution to derive , thus proving that Harmonia does have a grandparent. But who is it?

Answer Extraction

Clearly, if the last resolution step matched $\neg Grandparent(x, Harmonia)$ with a positive clause, that positive clause would carry a binding for x that would provide an answer to our question.

The technique of *answer extraction* augments the negated conclusion with an $Answer$ predicate to capture the desired binding. Thus, the negated conclusion becomes:
$\neg Grandparent(x, Harmonia) \vee Answer(x)$

The $Answer$ predicate is "invisible" when counting the number of literals, so that a clause consisting only of the $Answer$ predicate is recognized as .

More generally, the conclusion can be augmented by an "invisible" predicate that is the positive form of the negated question predicate:
$\neg Grandparent(x, Harmonia)$
$\vee Grandparent(x, Harmonia)$
This form can capture multiple variable bindings.

Planning

Planning attempts to find a *sequence of actions* that will accomplish a goal.

Problems:

- Ordering of actions is important.
- Search space may be very large – even infinite.
- Actions interact. Some actions may undo others or make later actions impossible.
- Overall constraints may exist, such as total of a conserved quantity (weight, cost, etc.).
- Minimizing cost of executing the plan is often important.

The Frame Problem

There is a problem[23] for a predicate calculus planning system: how to keep a world model consistent as some things in the world change.

There are two aspects:

- the *frame problem:* how to propagate information to the next situation when an action is taken. For example, if a box is at position p and the light is turned off, the box is still at p.
- the *ramification problem:* indirect effects of actions. For example, if a book is on the table and we move the table, the position of the book must also be changed.

Most facts in the world will be unchanged for any given action. The frame problem is a serious practical problem, since many solutions to it require vast amounts of storage and/or computation.

[23]McCarthy, J. and Hayes, P. J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, *Machine Intelligence 4*, American Elsevier, 1969. Note that the frame problem has nothing to do with frames.

Planning: Situation Calculus

It is possible to represent a state space in predicate calculus and use a proof procedure to find solutions to problems.

An example of a classical planning problem is the "Monkey and Bananas Problem:" A monkey in a cage wants to get some bananas that hang from the ceiling. The monkey cannot reach the bananas directly. However, there is a box in the cage; by pushing the box underneath the bananas and climbing on the box, the monkey can get the bananas.

Since a state space operator changes the state, we cannot simply represent facts with predicates such as $At(Monkey, a)$. While the monkey may be at a in some state, the monkey can move. Ordinary logic is *monotonic*, i.e., the set of true predicates can only increase; facts cannot be retracted.

One way to represent mutable facts is to add a *situation* or *state* variable to each predicate: $At(Monkey, a, s_0)$. This states that the monkey is at a in state s_0, which remains true even if the monkey moves (which puts it in a new state).

Operators in Situation Calculus

Operators can be represented as functions that change states into new states:
$\forall x \forall y \forall s[\neg On(Monkey, Box, s) \wedge At(Monkey, x, s) \wedge At(Box, x, s) \rightarrow At(Monkey, y, pushbox(x, y, s)) \wedge At(Box, y, pushbox(x, y, s))]$

pushbox is a function from states to states.

If we pose a question of "How can the monkey get the bananas?" and negate it, we will have $\neg Has(Monkey, Bananas, s)$ ("There is no possible state in which the monkey has the bananas.") After proving and using answer extraction, we will have an answer such as:
$Has(Monkey, Bananas,$
$\quad grasp(climbbox(pushbox(b, c, goto(a, b, s_0)))))$

Note that the instruction set of any CPU could be axiomatized as a state space; thus, any computation can be expressed in predicate calculus. In fact, binary Horn clauses have the power of a Turing machine.

Frame Axioms

There is a problem with situation calculus. Suppose the bananas are at c in state s_0. Where are the bananas after the monkey pushes the box, in state $pushbox(b, c, goto(a, b, s_0))$?

It is necessary to write *frame axioms* to describe what does *not* change when each operator is applied. Unfortunately, most things do not change for most operators. Worse, a combinatoric number of axioms is required.

This is an instance of the *frame problem*[24]: the problem of maintaining a valid world model as some things in the world change.

[24]The frame problem is not related to the representational technique of *frames*, which uses the same word.

Planning: STRIPS

STRIPS[25] is a logic-based language designed for robot problem solving. STRIPS maintains a set of *world models*, each of which is a set of ground predicates.

A *STRIPS operator* (analogous to a clause in predicate calculus) consists of:

1. *precondition list:* a set of predicates that must be true before the operator can be applied.
2. *delete list:* a set of predicates to be deleted from the world model when the operator is applied.
3. *add list:* a set of predicates to be added to the world model when the operator is applied.

In many cases, the precondition list and delete list turn out to be the same.

[25] from STanford Research Institute Problem Solver.

STRIPS Operators for Blocks World

STRIPS operators for a robot arm:

1. **pickup(x)**:
   P&D: ontable(x), clear(x), handempty()
   Add: holding(x)
2. **putdown(x)**:
   P&D: holding(x)
   Add: ontable(x), clear(x), handempty()
3. **stack(x, y)**:
   P&D: holding(x), clear(y)
   Add: handempty(), on(x, y), clear(x)
4. **unstack(x, y)**:
   P&D: handempty(), on(x, y), clear(x)
   Add: holding(x), clear(y)

The variables are instantiated to constants by proving the precondition wff.

A *world model* is a set of wffs, mostly ground clauses:

```
on(a,b)
ontable(b)
clear(a)
ontable(c)
clear(c)
```

STRIPS: Operator Application

A STRIPS operator is applied as follows:

1. Show that the preconditions are satisfied (by deriving from the negated Precondition and the world model); this instantiates parameters.
2. Form an offspring world model by first deleting everything on the Delete List from the existing world model, then adding everything on the Add List.

This is just state space search, but it is more general because the robot can perform many possible actions, and we can ask for many different kinds of goals.

Selection of STRIPS Operators

Given a current model of the world and a current goal, STRIPS attempts to prove the goal in the world model by resolution. If the goal is satisfied, the search can move on to the next goal, or succeed if the main goal is satisfied.

If the current goal cannot be satisfied in the current world model, STRIPS takes clauses derived from the goal as new subgoals, or as a *difference* between the world model and the goal (as in GPS).

Example: Suppose the negated goal is $\neg At(Box_1, x) \vee \neg At(Box_2, x)$ and the world model has $At(Box_1, b)$. Resolution gives the new subgoal $At(Box_2, b)$.

An operator relevant to reducing the difference can be found by looking for operators whose Add Lists unify with parts of the subgoal. After selecting an operator, the preconditions of the operator become additional subgoals.

Kinds of Planning

- *Non-hierarchical planning*: All actions can be considered at all times.
- *Hierarchical planning*: Break goal down into subgoals, recursively. Problem: subgoals may interact.
- *Linear planning*: Assume no interactions between subgoals.
- *Nonlinear planning*: Subgoals interact.
  - Paint the ladder and paint the ceiling.
  - Buy a car and a boat for less than $25,000.

Criticality Number

Sacerdoti introduced the notion of *criticality number* to aid in planning:

- Each type of goal is given a criticality number that indicates how difficult it is to achieve.
- Planning is done for the highest-criticality goals first.
- In planning for a given criticality level, preconditions at lower criticality levels are ignored.
- A plan is successively refined by filling in the lower-numbered goals and preconditions.

The basic assumption is that satisfying a lower-criticality goal will never undo a higher-criticality goal.

Plan Monitoring

In the real world, actions do not always have the desired effects. In addition, there are other actors (including nature) that may cause unexpected effects. *Plan monitoring* is needed to make sure that a plan can be carried out. The representation of a plan must include:

- Preconditions of an action. The robot should check whether these are in fact true before initiating the action.
- Expected effects of an action. The robot should check whether these are in fact true after the action.

If things go wrong, either local repair of the plan or complete replanning may be necessary.

Weaknesses of A.I. Planning

Much work on planning in A.I. has been based on assumptions that are very optimistic:

- The robot knows all the relevant facts about the world.
- The world can be described by a small set of predicates.
- The robot is the only actor.
- Actions produce the desired effects.

Planning without such assumptions is a difficult problem on which more research is needed.

Knowledge Rep. in Predicate Calculus

Facts: Facts can be stored in a *propositional database:*

```
(DOG DOG1)
(NAME DOG1 FIDO)
(HOUND DOG1)
(LOVES JOHN MARY)
```

Facts can be retrieved in response to *patterns:*

```
(LOVES JOHN MARY)     Does John love Mary?
(LOVES JOHN ?X)       Whom does John love?
(LOVES ?X MARY)       Who loves Mary?
(LOVES ?X ?Y)         All pairs of lovers.
```

Knowledge: Knowledge is stored as logical axioms that can be used for deduction. For example, the rule that 'all hounds howl' could be represented as:

```
(ALL X (IF (HOUND X) (HOWL X)))
```

or

```
(IF (HOUND ?X) (HOWL ?X))
```

Rules

Rules are typically written in an "*If ... then*" form:

`If <premises> then <conclusion>`

`If <condition> then <action>`

These forms correspond to the logical implication form:

$\forall x P_1(x) \wedge ... \wedge P_n(x) \rightarrow C(x)$

However, the interpretation of rules may or may not correspond to a formal logical interpretation.

Forward Chaining

In forward chaining, if the premises $P_1 ... P_n$ are known, the conclusion C is asserted. For example, if we have the axiom:

$\forall x MAN(x) \rightarrow MORTAL(x)$

and the fact MAN(Socrates) is added to the database of facts, the fact MORTAL(Socrates) will also be added.

Problems:

1. Infinite loops. For example, consider:
   $\forall x NUMBER(x) \rightarrow NUMBER(x+1)$
   $NUMBER(0)$
2. The forward assertions tend to fill up memory with uninteresting facts.

Backward Chaining

In backward chaining, if it is desired to prove the conclusion C of a clause, the system tries to do so by proving the premises $P_1 ... P_n$.

$\forall x CAR(x) \wedge RED(x) \rightarrow EXPENSIVE(x)$

Given this axiom, an attempt to prove that BMW_1 is expensive would be reduced to the subproblems of proving that it is a car and that it is red.

Problems:

1. Infinite loops. For example, consider transitivity:
   $\forall x \forall y \forall z GREATER(x, y) \wedge GREATER(y, z) \rightarrow GREATER(x, z)$
2. The system has to keep reproving (and failing to prove) the same mundane facts.

Importance of Backchaining

Backward chaining, rather than forward chaining, is the method of choice for most search problems. The reason is that backward chaining causes variables to be bound to the constant data of the problem of interest, and thus greatly reduces the size of the search space.

Example:

$\forall x \forall y WIFE(x, y) \rightarrow LOVES(x, y)$

$WIFE(John, Mary)$
$WIFE(Bill, Jane)$
...

Suppose we want to prove LOVES(John,Mary). Backward chaining will bind x and y in the theorem, do a single database lookup of WIFE(John,Mary), and succeed. Forward chaining will assert the LOVES relationship for every WIFE pair in the database until it happens to hit LOVES(John,Mary).

Reason Maintenance

Reason Maintenance or *Truth Maintenance* is a technique for maintaining a propositional database that has a `Retract` operation. We assume that the database has three operations:

- `Assert(P)`: Assert that the proposition `P` is true.
- `Ask(P)`: Ask whether something that unifies with the proposition `P` is true, e.g. `Ask(Loves(?x, Mary))`.
- `Retract(P)`: Remove the proposition `P` from the database.

We assume that the database does forward inference from assertions, so that if `P` and `P` $\rightarrow$ `Q` have been asserted, `Q` will also be asserted.

For efficiency, the database may be indexed for fast lookup on predicates and ground terms as arguments of the predicates.

Implementing Retraction

An obvious way to implement retraction would be to mark propositions that have been asserted by the user; to implement `Retract`, start over with an empty database and `Assert` all marked propositions except the one that was retracted. However, this would be inefficient, since the retraction is likely to have only limited effects.

A *Justification-based Truth Maintenance System* or JTMS keeps for each derived proposition a list of sets of propositions from which it was derived, and for each proposition, a list of propositions it supports. When a proposition is retracted, the system checks each proposition that depends on it, and if the proposition no longer has support, retracts it.

In some systems, it may be likely that a retracted proposition may be re-asserted in the future. Rather than removing a proposition from the database, the proposition can be marked as `IN` if it is currently believed, or `OUT` if it is currently not believed.

Truth Maintenance

The idea of a *truth maintenance system* (*TMS*) grew out of dependency-directed backtracking.

Given a set of boolean variables (or assumptions) Σ and a set of boolean constraints Γ , the general problem is to find a set of assignments to the variables that satisfies all the constraints.

Applications:

- Solve search problems, e.g., schedule meetings so that everyone can attend.
- Determine what derived conclusions are still true when some facts change.
- Determine what part failures could have caused a machine to fail.

Constraints could be expressed as:

- A 'digital electronics' network.
- A set of `IN` and `OUT` nodes that must be satisfied for a node to be true (`IN`).
- Logical relationships among boolean variables.

ATMS

An *Assumption-based Truth Maintenance System* or ATMS keeps for each proposition a set of sets of assumptions under which the proposition is true. A proposition is believed at a given time iff one of the assumption sets supporting it is satisfied.

Closed World Assumption

In some applications it is useful to make the assumption that the database contains all the relevant facts. Therefore, if something cannot be proved to be true, it can be assumed to be false; this is called *negation as failure*.

```
Is there an employee of the company
whose age is greater than 80?
```

It is probably a good assumption that if no such employee is represented in the database, there is none.

The opposite assumption, the Open World Assumption, assumes that if something cannot be proved we do not know its truth status. This is the normal assumption for logic.

PROLOG

PROLOG is a logic-based programming language. A PROLOG statement, $C \leftarrow P_1, ..., P_n$ can be considered to be a rule. Proofs proceed by backchaining.

Problems:

1. Hard to control search.
2. The Horn clause restriction prevents some kinds of rules from being written:
   (a) Rules which conclude a negated conclusion, or have a disjunction (OR) in the conclusion.
   (b) Rules which depend on a fact being *not* true. (Some PROLOGs do this using negation as failure.)
3. Backchaining is not logically complete. For example, it cannot do reasoning by cases.

PROLOG has the advantages that search is built into the language, and that PROLOG programs can run "forward" or "backward".

Non-Monotonic Logic

Ordinary logic is Monotonic, that is, new facts (axioms) can never invalidate previous deductions. The set of derived facts can only increase.

Problem: Real-world problems rarely present us with a complete set of facts. Usually, we make assumptions based on what is "usual"; sometimes these assumptions are incorrect. Non-monotonic logics attempt to allow assumptions to be made, an if necessary retracted, while staying within the framework of logic.

```
BIRD (x): m FLIES (x)
---------------------
      FLIES (x)
```

"IF BIRD (x) and it is consistent to assume FLIES (x), then assume FLIES (x)."

Induction and Abduction

Deduction: apply general principle to infer a fact.[26]

```
Given:  Every bird flies.  Tweety is a bird.
Infer:  Tweety flies.
```

Induction: assume a general principle that subsumes many facts.

```
Given:  Tweety, Polly, and Hooty are birds.
        Tweety, Polly, and Hooty fly.
        Fred is a bat.  Fred flies.
Assume: Every bird flies.
```

Abduction: guess a new hypothesis that explains some fact.

```
Given:  Every bird flies.  Tweety flies.
Guess:  Tweety is a bird.
```

Deduction is sound; induction and abduction are not sound, but essential for intelligence.

[26]Examples from John Sowa, "The Challenge of Knowledge Soup."

Predicate Calculus: Representation Language

There are several problems with Predicate Calculus as a representation language:

1. The level of representation is "frozen" at the level chosen for predicates and objects. Every predication made about a type of object restricts the objects which may be of that type.

   Example: $\forall x(Bird(x) \rightarrow Flies(x))$

   If we have such an axiom (reasonable at first glance), we have restricted the set of things which can be birds to those which fly. We can't talk about dead birds, birds that habitually don't fly, birds which can't fly due to injury, etc.; if we have such instances, we can derive and "prove" anything.

   On the other hand, if we enumerate the enabling conditions for a bird to be able to fly, (a) we couldn't enumerate all of them, and (b) we would have to prove every enabling condition in proving a theorem.

2. Predicate Calculus does not easily deal with:
   - Decomposition of an object into parts: "A car without an engine"
   - Mass nouns: "five gallons of water"
   - Representation at variable levels of detail.
   - Meta-rules: "There's an exception to every rule"
   - Things that are usually (but not always)true: "Most people can drive a car."
   - Knowledge about the state of knowledge: "John thinks Mary believes unicorns exist."
   - Modals: "You should take your umbrella if it might rain."
3. Predicate Calculus requires the Law of the Excluded Middle: if we use a predicate P to represent something, then either P or $\neg P$ must be true; "sort of P" or "probably P" is excluded as a possibility. The "real intelligence" in a reasoning process may be in the judgment of whether a proposition is true for a particular case.

Predicate Calculus as Programming Language

1. New knowledge or methods can be added.

   Advantage: In theory, at least, the program can immediately combine new knowledge with existing knowledge.

   Disadvantage: The "new knowledge" may contradict or subsume existing knowledge without our being aware of it.

2. Predicate Calculus is completely "unstructured". Any two clauses which are unifiable may interact.

3. In order to make a program run in a reasonable length of time, it is usually necessary to restructure clauses to:

   - Order the search so the desired solution will be found rapidly.
   - Reduce the branching factor of the search tree.

When to Use Logic

Logic is a preferred representation and reasoning method in cases where the data are discrete and there is "absolute truth". Such applications include:

- Mathematical theorem proving.
- Proofs of correctness of computer programs.
- Proofs of correctness of logic designs.

Semantic Networks and Frames

An alternative to Logic as a representation formalism is semantic networks and frames.

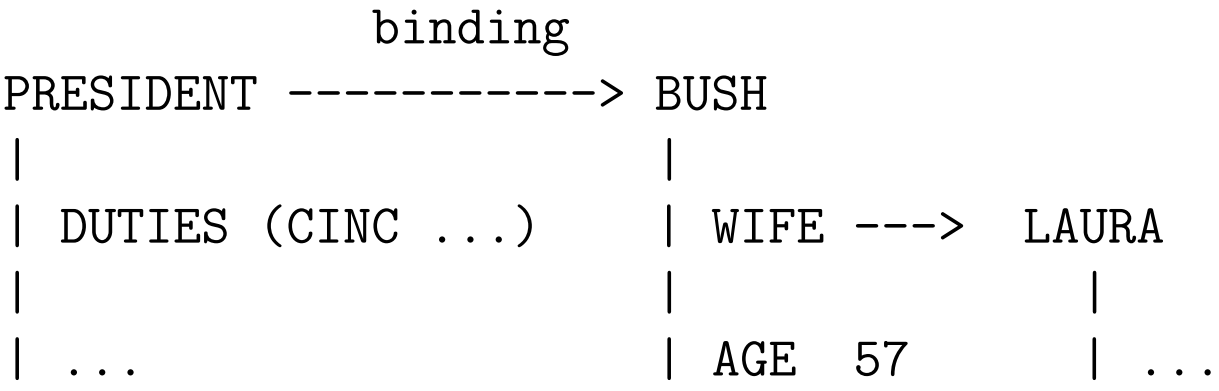
The property list feature of Lisp was used to represent networks of nodes connected by labeled arcs, forming *semantic networks*, by M. Ross Quillian, Robert F. Simmons, and others.

Following Minsky's "frame paper", procedures and inheritance were added to semantic networks to form frame systems.

More recently, frame systems have been interpreted as having semantics similar to logic, and they have been seen as more efficient implementations of certain common types of reasoning.

Property List Representation

Lisp provides for every symbol a *property list* that associates named properties with the symbol:



This mechanism has several advantages:

1. New properties can be added at any time; there is no need to pre-declare properties.
2. Only those properties needed for each individual object need to be stored.
3. A property can have a single value or a list of values associated with it.

Property lists are a natural mechanism to use in implementing *semantic networks*.

Property Lists

Each symbol has a *property list* on which semi-permanent properties of the symbol can be stored. Each property has a *property name* or *indicator* and a *value*. Property list values are retrieved and set by two functions:

`(get <symbol> <propname>)`
retrieves the value of the specified property for the specified symbol. If no such property exists, the value returned is `NIL`.

`(setf (get <symbol> <propname>) <value>)`
sets the value of the specified property for the specified symbol, displacing any previous value.

Each symbol has a single property list, which appears the same to all parts of a Lisp program and is *unaffected* by binding.

Advantages of Property Lists

The property list provides an easy way to create a database of relatively permanent facts about objects, e.g., the parts of speech of a word. It is convenient because additional kinds of facts can be added without interfering with existing facts, so long as the property names are different.

```
(setf (get 'car 'part-of-speech) 'noun)
```

Note that the symbol `car` can play several roles: it can hold facts about the English word "car" on its property list; it can be used as a variable name; and it is the name of a system function. These uses of `car` are entirely separate and do not conflict.

The property list provides an easy and safe way to create networks of relationships among objects (sometimes called *semantic networks*) while avoiding potential problems due to circular structures and multiple references to the same entities. These problems are avoided because a symbol is always guaranteed to be a single, unique structure in memory.

Frames

The term *Frame* was introduced in Minsky's paper "A Framework for Representing Knowledge".[27]

A basic idea of frames is that people make use of stereotyped information about typical features of objects, images, and situations; such information is assumed to be structured in large units representing the stereotypes, and these units are what are referred to as "frames".

Frames (or something similar) are important because they allow deep understanding of new situations about which only minimal information is directly available. They represent our understanding of regularities in the universe that allow intelligent action based on minimal clues.

[27]MIT AI Memo 306, 1974; reprinted in Brachman, R. and Levesque, H., *Readings in Knowledge Representation*, Morgan Kaufmann, 1985. Also see Minsky, Marvin, *The Society of Mind*, Simon & Schuster, 1986.

"Frame" Software Packages

There is no clear definition of what a frame implementation is. As a first approximation, a frame can be thought of as an extension of the Lisp property list. Numerous frame-like software packages have been developed, including FRL, KRL, KL-ONE, KM.

Object-oriented languages can be considered to belong to the class of frame languages; CLOS (Common Lisp Object System) is provided in Common Lisp. These languages trace their ancestry to the simulation programming language Simula. Smalltalk is a "pure" object-oriented language.

Typical Features of Frames

A frame can represent an *individual* object or a *class* of similar objects.

Instead of properties, a frame has *slots*. A slot is like a property, but can contain more kinds of information (sometimes called *facets* of the slot):

- The value of the slot; default value in case no value is present.
- A procedure that can be run to compute the value (an *if-needed procedure*).
- Procedures to be run when a value is put into the slot or removed (*if-added* and *if-removed* procedures). These can be used to implement *demons*.
- Data type information; constraints on possible slot fillers.
- Documentation.

Frames can inherit slots from *parent* frames. For example, `FIDO` (an individual dog) might inherit properties from `DOG` (its parent class) or `MAMMAL` (a parent class of `DOG`).

Simple Frame Program

Slots are stored on the property list of a symbol as an alist of facets, e.g. `((value 4))` .

```
(defun getslot (frame slot-name)
  (getslotb frame frame slot-name))

(defun getslotb (orig-frame frame slot-name)
  (let (slot if-needed)
    (if (setq slot (get frame slot-name))
        (or (cadr (assoc 'value slot))
            (if (setq if-needed
                      (assoc 'if-needed slot))
                (funcall (cadr if-needed)
                         orig-frame)))
        (some #'(lambda (super)
                  (getslotb orig-frame super
                            slot-name))
              (get frame 'supers)))))
```

A stored **value** is used if available; else an **if-needed** method is called if available; else an attempt is made to get the value from one of the **supers**.

Example Frame Data

```
(setf (get 'fido 'birth-year)     '((value  2004)))
(setf (get 'fido 'supers)         '(dog))
(setf (get 'dog 'barks)           '((value t)))
(setf (get 'dog 'supers)          '(mammal))
(setf (get 'mammal 'warm-blooded) '((value t)))
(setf (get 'mammal 'legs)         '((value 4)))
(setf (get 'mammal 'age)      '((if-needed agefn)))

(defun agefn (frame)
  (- (current-year) (getslot frame 'birth-year)))

>(getslot 'fido 'birth-year)    ; stored
2002

>(getslot 'fido 'barks)   ; inherited from dog
T

>(getslot 'fido 'legs)    ; inherited from mammal
4

>(getslot 'fido 'age)     ; inherited, if-needed
3
```

Shadowing

Because each frame is examined for a value before superclasses are examined, local values *shadow* inherited values.

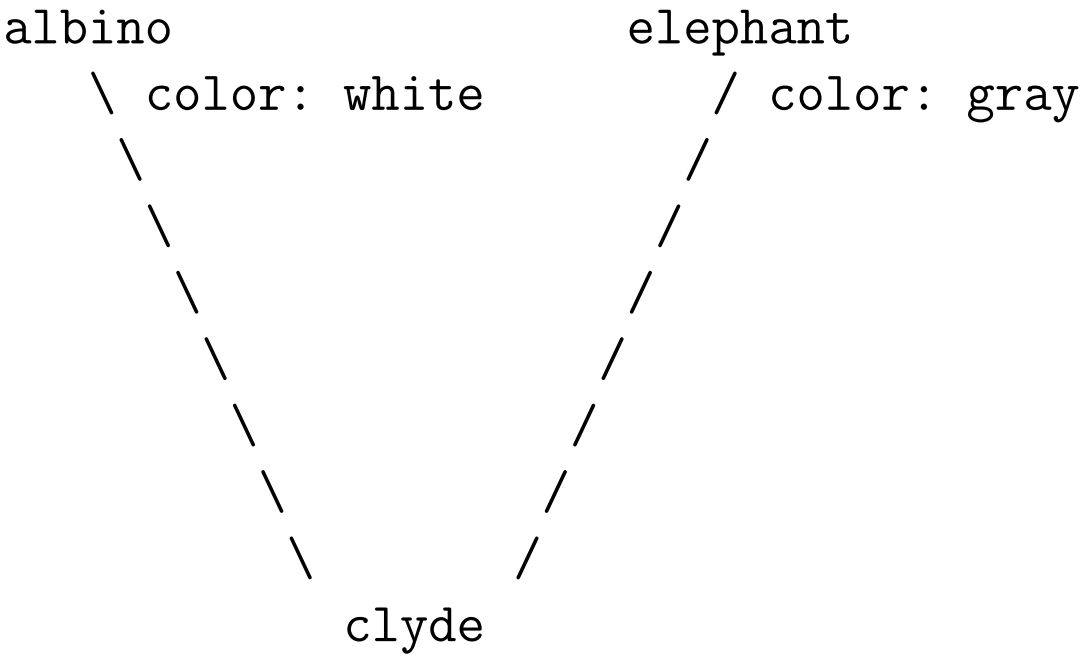
```
>(setf (get 'fido 'legs) '((value 3)))

>(getslot 'fido 'legs)
3
```

In this example, `fido` can have only 3 legs, although the inherited default value is 4.

Multiple Inheritance

Multiple inheritance can cause problems if different values are inherited along different paths.

Suppose `clyde`[28] inherits from both `elephant` and `albino`. Elephants normally have color `gray`, while albinos normally have color `white`. What color is `clyde`?



[28]Example due to Scott Fahlman.

Example Frame [29]

```
(defEntity Thermal-component
  :documentation "a thermodynamic device"
  :subclass-of (Thermal-system)
  :quantities
  ((efficiency :dimension dimensionless
               :documentation "the efficiency of ..."
               :the-*-the-object ("efficiency" "of")
               :abbreviation "e")
   (capacity   :documentation "the capacity of ..."
               :the-*-the-object ("capacity" "of")
               :abbreviation "c")
   (connected-to-external-heat-source-p
     :range-class boolean
     :non-numeric t
     :documentation "the attribute indicating ..."
     :value-names ((true "being heated")
                   (false "not being heated"))
     :the-*-the-object ("predicate indicating ..." "of")
   ))
  :operating-modes
  (thermal-component-op-mode cycle-heat-input)
  )
```

[29] From the How Things Work project at Stanford, Thomas Gruber *et al.* http://www-ksl.stanford.edu/htw/htw-demos.html

```
(defEntity extraction-turbine-system
  :subclass-of (system-model)
  :documentation "the system consisting of ..."
  :attributes
  ((xTbn :type simple-turbine
         :abbreviation tbn
         :The-*-the-object "turbine"
         :documentation "the turbine" )
   (xtbn-cv :type 1-2-control-volume
              :documentation "the control volume ..."
                :the-*-the-object ("control volume ..."
                )
   (expsn-prcs :type steady-flow-expansion
               :abbreviation xpn-prcs
               :The-*-the-object "expansion process"
               :documentation "the expansion process"
               :graphical-representation
                    (:Display-Class ())
              )  )
  :consequences
  ((cv-component (xtbn-cv ?self) (xtbn ?self))
   (component-cv (xtbn ?self) (xtbn-cv ?self) )
   (process-cv expsn-prcs (xtbn-cv ?self))
   )
  )
```

Advantages of Frames

1. A frame collects information about an object in a single place in an organized fashion. (Cf. Logic, which represents information about an object by many small predicates scattered throughout the database.)
2. By relating slots to other kinds of frames, a frame can represent *typical* structures involving an object; these can be very important for reasoning based on limited information.
3. Frames provide a way of associating *knowledge* with objects (via the slot procedures).
4. Frames may be a relatively efficient way of implementing A.I. applications (direct procedure invocation versus search in a logic system).
5. Frames allow data that are stored and computed to be treated in a uniform manner. (E.g., `AGE` might be stored, or might be computed from `BIRTHDAY`.)

Object-oriented programming has much in common with frames.

Disadvantages of Frames

Frames encourage baroque representations; little guide to good structuring of a domain.

Some things that can be represented in logic cannot be represented well – or at all – in frames:[30]

1. Slot fillers must be "real" data. For example, it is not possible to say that John is a butcher *or* a baker, since there is no way to deal with a disjunction in a slot filler.
2. It is not possible to quantify over slots. For example, there is no way to represent "Some student made 100 on the exam".
3. It is necessary to repeat the same information to make it usable from different viewpoints, since methods are associated with slots or particular object types. (For example, it may be easy to answer "whom does John love?" but hard to answer "who loves Mary?".)

[30]Michael Genesereth, lecture presented at Stanford.

When to Use Frames

Frames are good for applications in which the structure of the data and the available information are relatively fixed. The procedures (*methods*) associated with slots provide a good way to associate knowledge with a class of objects.

Frames are *not* particularly good for applications in which deep deductions are made from a few principles (as in theorem proving).

Object-Oriented Programming

The SIMULA language, for discrete event simulation, introduced OOP in 1967. The goal was to efficiently simulate large numbers of similar objects. A Class/Instance representation achieves this goal.

Class: Represents behaviors that are shared by all of its instances.

Instance: Represents the data for a particular individual.

Classes are arranged in a hierarchy, with inheritance of behaviors from higher classes.

Access to Objects

All access to objects is accomplished by sending *messages* to them.

- Retrieving data values:
  `(send obj x)`
- Setting data values:
  `(send obj x:  3)`
- Requesting actions:
  `(send obj print)`

Internal Implementation is Hidden

Messages define a standard interface to objects. Objects may have different internal implementations as long as the message interface is maintained.

Example: Vector: `(send v x)`

Vector type 1: x is stored

Vector type 2: r, θ are stored; x is computed.

The two kinds of vectors appear the same to the outside world.

Advantages of Objects

- **Modularity**
- **Flexibility:** New kinds of objects can be used with existing software if they understand the right messages.
- **Modifiability:** Internal implementations can be changed easily.

OOP is provided in Common Lisp by CLOS (Common Lisp Object System).

Object-Oriented Programming vs. Frames

Object-Oriented and Frame systems have much in common:

- Class/Instance structure.
- Inheritance from higher classes.
- Ability to associate programs with classes and call them automatically.

Because of these similarities, it is not possible to draw a sharp distinction between Object-Oriented systems and Frames. However, there are some differences that are typically found.

Unique Features of Frames

Frame systems typically have the following features in contrast to typical object-oriented systems:

- Richer slots. A slot can contain more kinds of information, e.g., documentation, default value, restrictions on slot fillers, if-added and if-removed methods.
- Slot orientation. There usually are no "messages" that are not associated with slots.
- More complex structure. A frame system could potentially have instance values, default values, and if-needed methods for the same slot. When all these are combined with multiple inheritance paths, the result can be complex.

Unique Features of Object-Oriented Systems

Object-oriented systems typically have the following features in contrast to typical Frame systems:

- Separation between Data and Methods.
- Methods need not be associated with slots. There can be methods that implement actions on the object as a whole.
- Regular structure. The structure of an object-oriented system is typically simpler and cleaner than that of a frame system.

Example: Frame

```
Ship
   latitude     type:       real
                range:      (-90.0 90.0)
   longitude    type:       real
                range:      (-180.0 180.0)
   x-velocity   type:       real
   y-velocity   type:       real
   speed        type:       real
                if-needed:  ship-speed-fn
```

If we wish to have an action **speedup** that doubles the velocities of the ship, it will have to be separate from the frame definition, since it is not associated with a slot.

Example: Object

```
(defclass ship ()
   ((latitude    :type real)
    (longitude   :type real)
    (x-velocity  :type real)
    (y-velocity  :type real)) )

(defmethod speed ((s ship))
  (with-slots (s)
    (sqrt (+ (expt x-velocity 2)
             (expt y-velocity 2))) ))

(defmethod speedup ((s ship))
  (with-slots (s)
     (setf x-velocity (* x-velocity 2))
     (setf y-velocity (* y-velocity 2))
     s ))
```

In this example, a method `speedup` is defined; with frames, this was not possible because `speedup` is not associated with a slot. The frame example showed `range` restrictions on the values of `latitude` and `longitude`; the object representation doesn't handle this.

Benefits of Frame Idea

Frames are large, structured units. A major benefit is that frames provide a large amount of structured information that can be invoked from a small "trigger". This is essential for intelligence:

- An intelligent being *never* gets complete, consistent information about the world.
- An intelligent being *must* jump to conclusions and make assumptions based on what is usual.
- *Sparseness* of the universe makes this work: only a few things are possible.

Problems with Frames

- Inheritance can cause trouble.
- There is more than one way to break down the world into taxonomies.
- Inference tends to become complex and ill-structured.
- Frames are good for representing static situations; they are not so good for representing *fluents*, things that change over time.

Scripts

A Script[31] can be viewed as a frame for a sequence of actions. Understanding natural language often requires knowledge of typical sequences:

```
John went to a restaurant.
He ordered a big steak.
He had forgotten his wallet.
He had to wash dishes.
```

The sequence of sentences mentions only the parts of the story that are "different" from what might otherwise be expected. Understanding such a sequence requires knowledge of a "restaurant script" that specifies typical sequences of actions involved in going to a restaurant.

In contrast to the relatively static slots of a Frame, a Script may have a directed graph of events composing the script.

[31] Schank, R. C. and Abelson. R P., *Scripts, Plans, Goals, and Understanding*, Hillsdale, NJ: Lawrence Erlbaum,1977.

Ontology

In philosophy, *ontology* is the study of being or existence, or the nature of being and kinds of existents; it comes from the Greek *ont* (to be) and *-logy* (theory of).

An *ontology* in AI refers to the set of kinds of objects and relations among objects that are represented in an AI system and used in reasoning; the ontology provides a vocabulary for talking about them.

One can have multiple ontologies, e.g. an ontology for set theory and an ontology for Newtonian mechanics.

`http://www-ksl.stanford.edu/kst/what-is-an-ontology.html`

Some Languages / Inference Engines

- **Algernon:** Kuipers, James Crawford. A frame system with logical semantics. *Socratic completeness:* logical completeness *if* the right questions are asked in the right order.
- **KM:** Porter, Peter Clark. A frame system with logical semantics. Skolems are lazily instantiated. KM can simulate actions, using delete and add lists.
- **CycL:** Logic-like representation language of Cyc. Microtheory: set of assertions that share assumptions.
  `cyc.com`
- **KIF:** Knowledge Interchange Format, a logic-like language for exchanging information among AI systems.
  `http://logic.stanford.edu/kif/kif.html`
  `http://www.ksl.stanford.edu/knowledge-sharing`

```
(defrelation natural (?x) :=
  (and (integer ?x) (>= ?x 0)))
(believes john '(material moon stilton))
(=> (believes john ?p) (believes mary ?p))
(=> (= (father ?x) ?y) (male ?y))
```

- **KL-ONE:** automatic classification of instances to the right place in a taxonomy.

Knowledge Bases or Ontologies

- **Clib:** Porter, Ken Barker, *et al.*: uses KM.
- **Cyc** KB: 47,000 concepts, 306,000 relations.
- **SUMO:** Suggested Upper Merged Ontology, mapped to WordNet.

```
*   Entity
      o Physical
            + Object
                  # SelfConnectedObject
                        * Substance
                        * CorpuscularObject
                        * Food
                  # Region
                  # Collection
                  # Agent
            + Process
      o Abstract
            + SetOrClass
            + Relation
            + Quantity
                  # Number
                  # PhysicalQuantity
            + Attribute
            + Proposition
```

Overview of Knowledge Representation

- Frames are used more than logic for applications.
- Theorists favor logic.
- Newer knowledge representation systems such as KM combine features of frames (for efficiency) and logic.
- There are many difficult problems in knowledge representation.

Situated Action

It is a mistake to try to represent everything in the world and rely too much on reasoning.

An alternative is to use the world as its own best representation and simulation, and rely on perception.

Rules, of the form:

```
In situation x, do y.
```

seem to be good models of how many organisms, including humans, work.

The Expert System view is to base intelligence on large amounts of shallow knowledge rather than on sophisticated inference.

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:[32]

  ```
  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.
  ```

[32] 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 a lot 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 legally to form 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'll never 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:**[33] 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.

[33] The singular is *anaphor*.

Outline of Natural Language Section

- Introduction
- Morphology (word forms) and Lexicon (dictionary). A *morphological analyzer* converts words to root forms and affixes.
- Grammars and Parsing (syntax). A *grammar* is a set of rules describing legal ways to combine words and phrases. A *parser* analyzes a sentence to determine its structure according to the Lexicon and Grammar. A *generator* converts a meaning structure to a statement in the language, using the Lexicon and Grammar.
- Semantics
  - Case theory
  - Semantic Markers, Selection Restrictions, Case Frames
- Representation of meaning: semantic networks, predicate calculus
- Anaphora, reference, discourse
- Applications; semantic grammar; augmented transition network

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.[34]

```
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.

[34]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.[35]

```
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

[35] 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*.[36]

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:[37]

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!

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

[37] 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.[38]

A *Hidden Markov Model* (HMM) tagger chooses the tag for each word that maximizes: [39]

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

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%.

[38] from Jabberwocky, by Lewis Carroll.
[39] 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.[40]

[40] 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?):

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
  el caballo blanco

- 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.

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

Phrase Structure Grammar

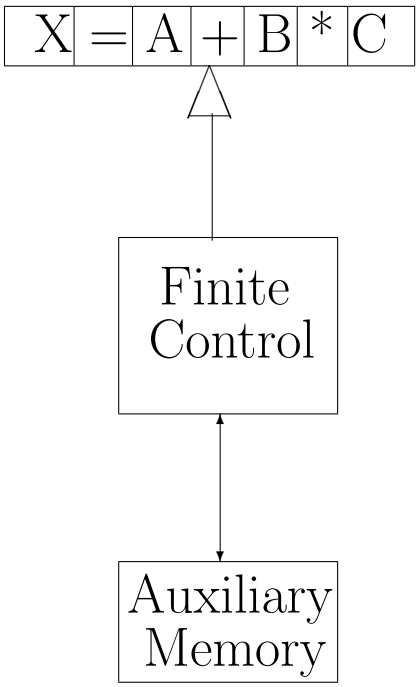
A grammar describes the structure of the sentences of a language in terms of components, or phrases. The mathematical description of phrase structure grammars is due to Chomsky.[41]

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.

[41] 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?"

Chomsky Hierarchy of Languages

Regular Languages

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

- 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.

Context Sensitive Languages

Productions: $\alpha \to \beta$
$\alpha \in V^* N V^*$
$\beta \in V^+$
$|\alpha| \leq |\beta|$

The strings around the N on the left-hand side of the production are the *context*, so a production works only in a particular context and is therefore context sensitive.

- Context sensitivity seems applicable for some aspects of natural language, e.g., subject-verb agreement.
  John likes Mary.
  $*$ John like Mary.
- No effective parsing algorithm is known.
- Parsing is NP-complete, i.e., would take exponential time.
- Context sensitive languages are less often used in practice than context free languages.

What Kind of Language is English?

- English is Context Free.[42]
- English is not Context Free.[43]
- 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?*

[42] 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.

[43] 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.[44]

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.

[44] 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:

```
     <NP>
     /  \
 <art> <noun>
   |     |
  The   man   ate   fish
```

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

Chart Parser

A *chart parser* is a type of bottom-up parser that produces all parses in a triangular array called the *chart*; each chart cell contains a set of nonterminals. The bottom level of the array contains all possible parts of speech for each input word. Successive levels contain reductions that span the items in levels below: cell $a_{i,k}$ contains nonterminal N iff there is a parse of N beginning at word i and spanning k words.

5	S				
4					
3	NP		VP		
2	NP			NP	
1	Art	Adj, N	N, V	Art	N
0	The	old	man	the	boats
	1	2	3	4	5

The chart parser eliminates the redundant work that would be required to reparse the same phrase for different higher-level grammar rules.

The Cocke-Kasami-Younger (CKY) parser is a chart parser that guarantees to parse any context-free language in at most $O(n^3)$ time.

Augmented Transition Networks

An ATN [45] 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.

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

Augmented Transition Networks

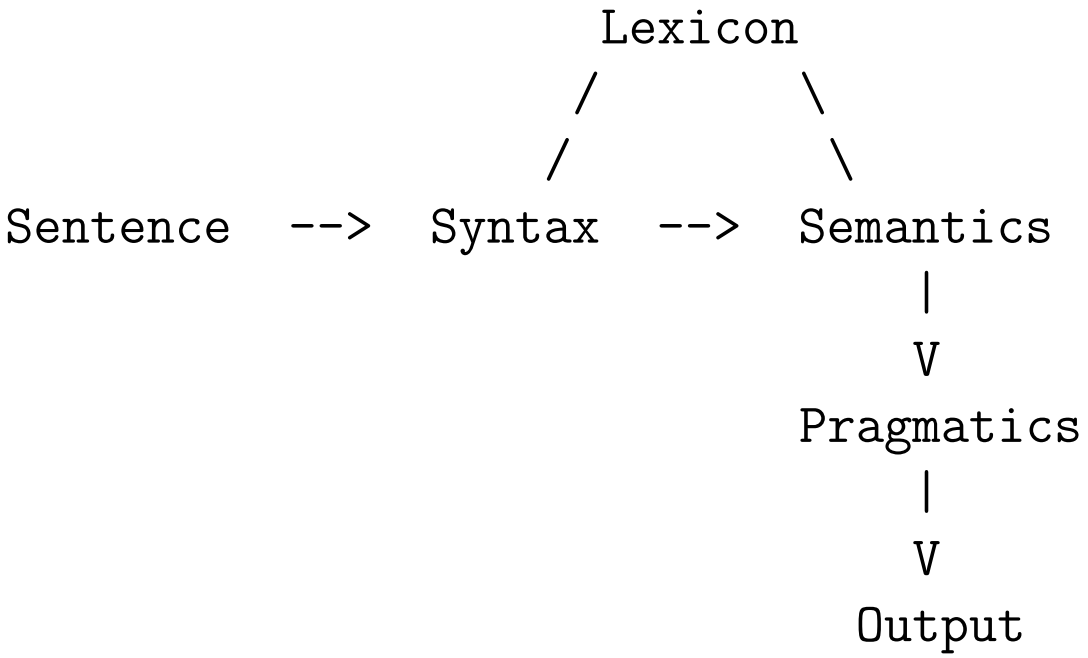
A grammar can be written in network form. Branches are labeled with parts of speech or phrase names. Actions, such as constructing a database query, can be taken as arcs are traversed.

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.

Semantics

Several tasks fall under the heading of semantics:[46]

- Selecting correct word sense meanings.
- Removing ambiguity: choosing interpretations that "make sense" when many interpretations are syntactically possible.

  ```
  John saw my dog driving to work this
  morning.
  ```

- Resolving pronoun references.

  ```
  Bill wanted John's bike.
  He stole it.
  ```

- Resolving other references.

  ```
  ...  a ladder ...
  A man is 10 ft from the top.

  A bridge is supported at each end.
  ```

[46] *Semantics* is the study of meaning. For our purposes, *semantics* means the process of determining the meaning of an input sentence.

Semantics ...

- Producing a representation of the *meaning* of the input language. This representation may be quite different from the input.

  ```
  John sold his rabbit to Bill.
  Did Bill buy a pet?
  ```

- Filling in missing information.

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

  The battery *is part of* my car. The battery's being dead *caused* my car not to start. The car's not starting *caused* me to be late for work because I usually use the car to drive to work and because if a car will not start, it cannot be driven.

Ambiguity

There is much more potential ambiguity in natural language sentences than one might think.

Lexical Ambiguity: A word can have multiple parts of speech and multiple meanings for each part of speech.

> *You can verb anything.* – William Safire.

The number of combinations of meanings is the *product* of the number of meanings for each word.

Syntactic Ambiguity: Phrases might be attached to different parts of the sentence. Especially troublesome are prepositional phrase attachment and conjunctions.

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

```
Lowering the level of the lake allows
city officials to kill weeds and
residents to repair their docks.
```

Reducing Ambiguity

A classical AI technique is:

> *Use global consistency to constrain interpretation of locally ambiguous features.*

Suppose we have a sentence such as:

> `John hit the ball with a bat.`

How can we determine that `ball` is a spherical object (not a dance), that `hit` means to strike (not to arrive at), and that `bat` is a baseball bat (not a small flying animal)?

```
John struck the spherical object with
a stick.
John arrived at the dance accompanied
by a small flying animal.
John arrived at the dance that
featured a small flying animal.
John struck the dance with a stick.
...
```

Case Theory

Fillmore[47] proposed a theory of *deep case structures*, in which elements of a sentence are related to the verb by *deep case*[48] relationships.

Unlike some languages, English does not modify most words for different cases. Cases used in English are:

Formal Name:	Description:	Example:	Example Use:
Nominative	subject	he	He hit the ball.
Objective	direct object	him	John hit him.
Dative	indirect object	him	I gave him a book.
Genitive	possessive	his	He lost his keys.

Although English does not make the cases obvious, Fillmore argued that the cases are still present in English:

Mother baked for three hours.
The pie baked for three hours.
* Mother and the pie baked for three hours.

Why is the third sentence anomalous?[49] Although both "Mother" and "the pie" are syntactic subjects, Fillmore argued that they are in different deep cases and thus cannot be conjoined.

[47] Charles Fillmore, *The case for case*, 1968.

[48] *case*: an inflectional form of a noun, pronoun, or adjective indicating its grammatical relation to other words; such a relation whether indicated by inflection or not. – *Webster's 9th New Collegiate Dictionary*

[49] Linguists use the * marker to indicate a bad sentence.

Case Relations

Agent	instigator of the event
Counter-agent	resistance against action
Object/Theme/Patient	entity that moves or changes
Result	entity that comes into existence
Instrument	physical cause of event
Source	place from which something moves
Goal	place to which something moves
Experiencer	entity which receives effects
Locus	place where event occurs
Modality	tense etc. of verb

Case Relation Example

John broke the window with the hammer.
John broke the window.
The hammer broke the window.
The window broke.

These could all refer to (parts of) the same deep case structure:

```
Break1:
   Tok:         Break
   Modality:    Tense past, Voice active, ...
   Agent:       John
   Object:      Window
   Instrument:  Hammer
```

Such a case structure can also be represented as a semantic network:

Case Frames

A *case frame* is a lexical definition for a word sense that tells how other phrases are related to it. A case frame might include:

- **Selection Restrictions:** restrictions on the possible slot fillers for the frame.
- **Semantic Structure:** how to build an output structure to represent the meaning.
- **Related Phrases:** how certain phrases, e.g. prepositional phrases, may fit into the meaning.

```
HIT-1    <subj>  +animate
         <obj>   +concrete
         [with <inst>  +concrete,-animate]

-->    STRIKE    AGENT       <subj>
                 THEME       <obj>
                 INSTRUMENT  <inst>
```

Similar patterns work well for prepositions:

```
<person> WITH <person>
<location> OF <object>
```

Disambiguation Using Case Frames

The *selection restrictions* of the case frame can be matched against the *semantic markers* of different sense meanings of words to choose combinations that make sense.

```
John hit the ball with a bat.
```

The selection restrictions of `HIT-1` require the marker `+concrete` for the object; this is true of the "spherical object" meaning of `ball`, but not of the "dance" meaning. The correct sense of `bat` can be found in a similar way.

Preference Semantics

Yorick Wilks has suggested the use of *preferences* rather than absolute restrictions for the selection of sense meanings. The preferences are numbers that express the "goodness of fit" of representations.

> `Would you hit a woman with a child?`
> `No, I'd hit her with a brick.`[50]

The humor derives from the violation of a selection restriction – the use of `child` as instrument of the hitting. Preference semantics would indicate that such an interpretation is not preferred, but is not impossible.

[50] "The eternal question and immortal answer of burlesque." – e.e. cummings

Pronoun Reference

A pronoun often refers to the first preceding noun phrase that agrees with it in number and gender – but not always.

Charniak: no simple-minded syntactic algorithm will work for pronoun reference.

> `The city council denied the demonstrators a permit because they feared violence.`
>
> `The city council denied the demonstrators a permit because they advocated violence.`[51]

Deep semantics is required to resolve such cases.

[51] Winograd, T., *Understanding Natural Language*, Academic Press, 1972.

Deep Semantics Influences Parsing

The man *at one end* weighs 150 lb.
The man [who is] at one end [of the lever] ...

The man supports the lever *at one end.*
* ... the lever [that is] at one end [of the lever]

Both examples fit the pattern:

```
<object> AT <location>
```

However, in the second case, attaching *at one end* to the noun phrase would be wrong. Additional semantics is needed.

Rule: an object may not be modified by a location on itself.

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

Scripts and Reference

Schank's scripts[52] can be seen as memory triggers that bring into focus the objects that may be referenced in later sentences:

```
John went to a restaurant.
The waiter ...
```

In order to understand the definite reference *the waiter*, the reader must be prepared to see such a reference. Conceptually, a script implements something like: $\forall x Restaurant(x) \rightarrow \exists w Waiter(w) \wedge \exists z Food(z) \wedge ...$ These newly created objects can then be related to the things mentioned in later sentences.

[52]Schank, R. C. and Abelson, R. P., *Scripts, Plans, Goals, and Understanding*, Hillsdale, NJ: Lawrence Erlbaum, 1977.

Internal Representation

Some way of internally representing the *meaning* of natural language statements must be chosen.

Natural language itself *cannot* be the internal representation:

```
John sold a boat to Bill.
What did Bill buy?
Did John receive money?
```

At minimum, a representation must be able to represent:

- Objects
- Properties of objects
- Relationships among objects.

Semantic networks and predicate calculus are two popular representations.

How Not to do Representation

One pitfall is so common that it has a name: *Pretend It's English*. If English words or phrases are used in a representation, it looks good when printed – but only to a human.

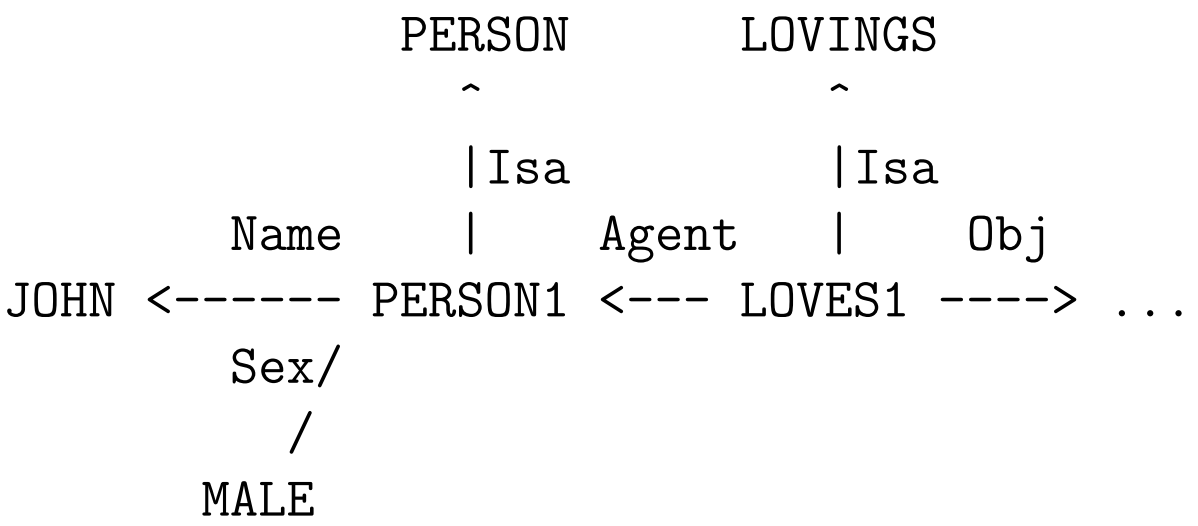
```
(lintel must-be-supported-by post1)

point-mass(man)
```

Is `must-be-supported-by` related to `support`? Only if it is specially programmed; no relation between the two is built-in. Is `man` a man? No, it is just an arbitrary name for a predicate calculus constant.

Tokens

In general, it is a poor idea to use words themselves as the internal representation. If we represent `John loves Mary` as `LOVES(JOHN,MARY)` there can be only one John and one Mary.

```
LOVES(PERSON1,PERSON2)
NAME(PERSON1,JOHN)
ISA(PERSON1,PERSON)
SEX(PERSON1,MALE) . . .
```



Time Sequence

Natural language often involves an implicit time sequence.

```
It was a dark and stormy night.
A shot rang out.
John opened the door and saw the body.
```

The language understander needs to infer:

- Time sequence: John heard the shot, *then* opened the door, *then* saw the body.
- What is true at different times: it is still a dark and stormy night, but the shot can no longer be heard.
- Causal sequence: John opened the door *because* he heard the shot. (A backward inference: we know John opened the door, so we presume he heard the shot.)

Rhetorical Relations

Rhetorical relations (Simmons, Alterman) are a kind of higher-level grammar to describe larger units of text. Expository text, such as might be found in an encyclopedia, often has a high-level structure such as:

1. Give a related concept.
2. Give differentia of the concept being described.
3. Give an example.

Conclusions on Semantics

- Natural language statements contain only the bare outline of the full message.
- Understanding natural language requires filling in a *lot* of information.
- Some kind of frame-like representation is required to guide the many assumptions necessary.

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 is 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.
- 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 biggest 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

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.

ATN Written in Lisp

Using a few simple functions and conventions, it is easy to write ATN-like programs directly in Lisp. Each grammar rule becomes a Lisp function.

```
(DEFUN <phrase-name> ()
  (LET (<local variables>)

    (SAVEPTR)   ; always the first action

    (IF (AND (CAT <category>)
          ; to test parts of speech
             (NEXT)  ; move to next word
             ... )
        (PROGN (SUCCESS)  ; it worked
               <semantics> )
        (FAIL)) ))  ; parsing failed
```

Results may be either returned or stored in global registers or database as a side effect.

Sentence Pointer Handling

```
;   *sent*     = remainder of sentence
;   *word*     = current word
;   *savesent* = saved positions for backup

; initialize for a new sentence
(defun init-sent (sent) (setq *sent* sent)
                        (setword))

;   set *word* for current position
(defun setword () (setq *word* (car *sent*)))

;   move to next word
(defun next    () (pop *sent*) (setword) t)

;   save the current position
(defun saveptr () (push *sent* *savesent*))

;   pop the stack on success
(defun success () (pop *savesent*) t)

;   restore position on failure, return nil
(defun fail    ()
   (setq *sent* (pop *savesent*))
   (setword)  nil)
```

Word Category Testing

```
;   Test if current word is in category
(defun cat (category) (get word category))

;   Define a lexicon.
;   Arg is list of (category ( items ))
;   Each item is word or (word value)
(defun deflexicon (lst)
  (dolist (l lst)
    (let ((category (car l)))
      (setf (part-of-speech category) t)
      (dolist (word (cadr l))
        (if (symbolp word)
            (setf (get word category) t)
            (setf (get (car word)
                       category)
                  (cadr word))) ) ) ) )
```

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.

Lexicon Example

```
;   Now we define a lexicon and grammar
;   for this application.

(deflexicon
 '((part     ((pliers pliers)(tires tire)
              (tire tire)(blimps blimp)
              (widgets widget)))
   (supplier ((sears sears)(acme acme)))
   (customer ((ut ut)(hp hp)(mcc mcc)))
   (city     ((austin austin)
              (dallas dallas)
              (el-paso el-paso)))
   (month    ((january 1)(jan 1)
               (february 2)(feb 2)...))))
```

Note translations to internal form, e.g.,
`tires --> tire`, `February --> 2` .

It is easy to include alternate forms such as abbreviations, slang, and special terms.

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

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

Our example database program takes queries of the form:

```
(QUERYDB <database> <condition> <action>)
```

The condition and action are Lisp code: 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

```
; *retrieve* = things to get from database
; *restrict* = restrictions on the query

;   Main function: ASK
(defun ask (question)
  (let (*retrieve* *restrict*)
    (parse question)
    (querydb 'orderdb
      (cons 'and *restrict*)
      (cons 'list (nreverse *retrieve*)))))

;   Quote something
(defun kwote (x) (if (constantp x) x
                     (list 'quote x)))

;   make a database access call
(defun dbaccess (field)
  (list 'getdb (kwote field)))

;   add a restriction to the query
(defun addrestrict (r) (push r *restrict*))
```

Parser

```
(defun parse (sent)
 (let ()
   (init-sent sent)
   (if (eq *word* 'what) (next))
   (cond ((eq *word* 'customers)
          (next)
          (addretrieve
            (dbaccess 'customer))
          (loc)
          (if (eq *word* 'ordered)
                (next))
          (part) (supplier) (datespec))
         ((eq *word* 'who)
          (next)
          (addretrieve
            (dbaccess 'customer))
          (loc)
          (if (eq *word* 'ordered)
              (next))
          (part) (supplier) (datespec)))
;  (prin1 *retrieve*) (terpri) ; for trace
;  (prin1 *restrict*) (terpri) ; for trace
   (if *sent* (error `words left over'))))
```

Phrase Parsing

```
;   parse a location
(defun loc ()
 (let (locname)
  (saveptr)
  (if (and (eq *word* 'in) (next)
           (setq locname (cat 'city))
           (next))
      (progn
        (addrestrict
          (list 'equal
                (dbaccess 'customer-city)
                (kwote locname)))
        (success))
      (fail) ) ))
```

Example:

```
Input:   ... in Austin ...

Output:  (EQUAL
           (GETDB (QUOTE CUSTOMER-CITY))
           (QUOTE AUSTIN))
```

Grammar Compiler

A simple grammar compiler (`gramcom.lsp`) can compile grammar rules into parsing programs:

```
(defgrammar
(S -> (what (CUST) (LOC)? ordered (PARTS)?
            (SUPP)? (DATE)?)
      (combine $2 $3 $5 $6 $7))
(S -> (who (LOC)? ordered (PARTS)? (SUPP)? (DATE)?)
      (combine (retrieve 'customer) $2 $4 $5 $6))
(CUST -> (customer)  (retrieve 'customer))
(CUST -> (customers) (retrieve 'customer))
(LOC  -> (in (city)) (restrict 'customer-city $2))
(DATE -> (in (month))
                     (restrictb '= 'date-month $2))
(DATE -> (before (month))
                     (restrictb '< 'date-month $2))
(DATE -> (after (month))
                     (restrictb '> 'date-month $2))
(PARTS -> ((part))   (restrict 'part $1))
(SUPP -> (from (supplier)) (restrict 'supplier $2))
)
```

An Example Query

```
(ask '(what customers in austin
            ordered widgets in april))

QUERYDB entry:
  Arg 1: ORDERDB
  Arg 2: (AND (EQUAL
               (GETDB (QUOTE DATE-MONTH))
               4)
              (EQUAL (GETDB (QUOTE PART))
                     (QUOTE WIDGET))
              (EQUAL
               (GETDB
                 (QUOTE CUSTOMER-CITY))
               (QUOTE AUSTIN)))
  Arg 3: (LIST (GETDB (QUOTE CUSTOMER)))

QUERYDB return value = ((MCC) (MCC))
```

Example Queries

```
(ask '(who ordered from sears))
  ((UT) (HP) (UT) (MCC))
(ask '(who ordered blimps))
  ((UT))
(ask '(what customers in austin
            ordered before april))
  ((UT))
(ask '(what customers in austin
            ordered after march))
  ((MCC) (MCC) (UT) (HP) (MCC))
(ask '(who ordered from acme
         after april))
  ((HP))
(ask '(who ordered tires from sears))
  ((HP) (UT))
(ask '(who ordered tires))
  ((HP) (UT) (MCC) (UT))
(ask '(what customers in austin
            ordered widgets in april))
  ((MCC) (MCC))
```

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 2005?
A: 0
```

  Does this mean:

  1. Nobody failed.
  2. CS 381K was not offered in Summer 2005.
  3. There is no such course as CS 381K.

- **Gaps:** Is it possible to state the desired question?

Menu-based Natural Language

Texas Instruments (Harry Tennant, Craig Thompson) has developed a menu-based natural language access system.

Advantages:

- Fast: one mouse click selects several words.
- Spelling errors are impossible.
- Syntax errors are impossible.
- Semantic errors are impossible.
- The user sees all queries that are possible: self-training, permits finding a way to do a desired query.

Statistical NLP

> Every time I fire a linguist, the performance of the recognizer goes up. – Fred Jelinek

Traditional NLP involves writing grammars, parsers, and semantic programs to understand language.

An alternative is to use statistical approaches, based on statistics from massive amounts of text, without attempting to understand the natural language. This approach has been surprisingly successful in several area:

- Part-of-speech tagging
- Machine translation
- Speech understanding

The basic approach is Bayesian statistics: given the preceding language, what is the most probable interpretation of the current input?

Speech Understanding

Advantages:

- Fast compared to typing.
- No bulky equipment (keyboard).
- Hands not required (vehicle driver).
- Voice reply is easy to implement.

Problems:

- Ambiguity: “five” vs. “nine”.
- Pauses between words may be required.
- Noisy environments.

Hidden Markov models (HMM) are often used for speech understanding.

Machine Translation Example

This is example is produced by Babelfish:
`http://babelfish.altavista.com/`

English: (quote from Paul Graham)

> Back when I was writing books about Lisp, I used to wish everyone understood it. But when we started Viaweb I found that changed: I wanted everyone to understand Lisp except our competitors.

To German:

> Ziehen Sie wenn ich war Schreiben Bücher über LISP zurück, ich pflegte, jeder zu wünschen verstand es. Aber, als wir Viaweb begannen, fand ich, dass geändert: Ich wünschte jeder LISP ausgenommen unsere Konkurrenten verstehen.

And back to English:

> Pull if I were letter books concerning LISP back, I tended, everyone to wish understood it. But, when we began Viaweb, I found that changed: I wished excluded our competitors to each LISP understand.

Sentence Understanding in ISAAC

One end of a pole 10 ft long is supported by a man.

1. The initial noun phrase "one end" is parsed, producing this structure:

```
tok1
  tok    end
  lframe np
  nbr    (ns)
  det2   one
```

2. An attempt is made to parse the prepositional phrase as a modifier of the noun phrase.

3. The prepositional phrase parser causes the noun phrase "a pole" to be parsed:

```
tok2
  tok    pole
  lframe np
  det    indef
  nbr    (ns)
```

4. The sentence is examined for modifiers of the noun phrase. The phrase "10 ft long" is found and causes a modifier to be added to `tok2`:

```
  mods   ((length 10 ft))
```

5. The preposition semantics routine for `of` is called with `tok1` and `tok2` as arguments. The discrimination net in `ofsem` classifies this use of `of` as being of the form: `<location> of <object>`.

6. `ofsem` calls `idrfnt` to identify the referent of "a pole". This causes `tok2` to be assigned the semantic frame `physent` and causes an object `pole3` to be created and added to the internal model. The modifier `(length 10 ft)` is transferred to the new object at this time.

```
tok2
  tok    pole
  lframe np
  det    indef
  nbr    (ns)
  mods   ((length 10 ft))
  sframe physent
  rfnt   (pole3)
```

7. `ofsem` next assigns the semantic frame `locpart` to `tok1` and fills the `semobj` (semantic object) slot of this frame with the *referent* of the phrase that was the object of the preposition. `ofsem` then returns True, indicating that the modification was semantically acceptable.

```
tok1
  tok    end
  lframe np
  nbr    (ns)
  det2   one
  sframe locpart
  semobj (pole3)
```

Note that since the meaning of `tok2` has been determined and used, `tok2` is no longer part of the parse structure.

8. The verb phrase parser is called (with `tok1` as an argument) to parse the verb phrase. Since the verb phrase is passive, `tok1` is attached to it as the `obj` case argument.

```
tok4
  tok    support
  lframe np
  mainvb supported
  aux    (is)
  trans  t
  pasv   t
  obj    tok1
```

9. `vpmod` is called to parse modifiers of the verb phrase. It attempts to parse the prepositional phrase, which causes the noun phrase "a man" to be parsed.

```
tok5
  tok    person
  lframe np
  word   man
  mods   ((restrict (sex male))
          (restrict (age adult)))
  nbr    (ns)
  sframe physent
  rfnt   (person6)
```

10. The preposition semantics routine for `by` is called with the verb phrase and noun phrase tokens as arguments. `bysem` calls `idrfnt` to identify the referent of `tok5`; this causes `tok5` to be assigned the semantic frame `physent` and causes a new object `person6` to be created and added to the physical model.

    `bysem` then makes `tok5` the filler for the `subj` case argument of the verb phrase token `tok4`.

11. Since the whole sentence has now been parsed, the verb semantics for the verb `support` are now called. The structure seen by `supportsem` is as follows:

```
tok1                tok4              tok5
 tok    end          tok  support     sframe physent
 sframe locpart      subj tok5        rfnt   (person6
 semobj (pole3)      obj  tok1
```

    This is of the form `<physent> support <locpart>`.

12. `supportsem` calls `locnp` to identify the location referent of the phrase `tok1`. This causes a location object `loc7` to be added to the internal model.

```
tok1
  tok    end
  sframe locpart
  semobj (pole3)
  rfnt   (loc7)
```

13. `idatt` is called to identify an attachment between `pole3` at location `loc7` and `person6` (location unknown). This causes the creation of an attachment relation in the internal model. In addition, the `support` relation between `person6` and `pole3` is recorded.

This completes processing of the sentence. The token structures produced during the parsing process will not be used any further.

One end of a pole 10 ft long is supported by a man.

The internal model produced in response to this sentence is as follows:

```
pole3
  tok        pole
  entity     physent
  locs       (loc7)
  attach     (attach8)
  supportby  (person6)
  length     (10 ft)

person6
  tok        person
  word       man
  entity     physent
  restrict   ((sex male)
              (age adult))
  attach     (attach8)
  support    (pole3)

loc7
  entity     location
  frame      location
  object     pole3
  locname    end
  select     (one)

attach8
  frame      attach
  typeatt    pinjoint
  locs       ((pole3 loc7)
              (person6 nil))
```

Preposition Semantics

Sense-meaning determination for `of`:

quantifier `of` *objects*	each of the objects
measure `of` *value*	a length of 10 ft
object `of` *value attribute*	a pole of uniform cross-section
location `of` *object*	one end of the lever
attribute `of` *object*	the weight of the lever
group `of` *objects*	pair of legs
part `of` *object*	hinges of a door

Execution of semantics:

1. Functional `sframe` form
2. Special semantic routines

Expert Systems[53]

Expert systems attempt to capture the knowledge of a human expert and make it available through a computer system.[54]

Expert systems are expected to achieve significant actual performance in a specialized area that normally requires a human expert for successful performance, e.g, medicine, geology, investment counseling.

Expert systems have been some of the most successful applications of A.I. Since these programs must perform in the real world, they encounter important issues for A.I.:

- Lack of sufficient input information
- Probabilistic reasoning

[53]These slides jointly authored with Bruce Porter.

[54]Duda, R. O. and Shortliffe, E. H., "Expert Systems Research", Science, vol. 220, no. 4594,15 April 1983, pp. 261-268.

Power-Based Strategy

Some have hoped that powerful theorem-proving methods on fast computers would allow useful reasoning from a set of axioms. Several problems have kept this power-based strategy from succeeding.

- Combinatoric explosion: blind search using even a small axiom set takes excessive time.
- Knowledge representation: few real-world relationships are universally true.
- Lack of inputs: many problems lack some inputs, but require fast action anyway.

Knowledge-Based Strategy

> "In the Knowledge Lies the Power"

The knowledge-based strategy is to include within the program a great deal of knowledge to cover particular cases.

The surprising finding:

> A thousand rules can provide significant performance within a limited domain.

Expert Reasoning

Expert reasoning typically has special characteristics:

- Use of specialized representations appropriate to the domain and specialized problem-solving methods based on those representations.
- Translation of observables into specialized terminology and representations (e.g., "person has turned blue" into "patient is cyanotic").
- Use of empirical rules of thumb (e.g., "to blow out a tree stump, use one stick of dynamite per 4 inches of stump diameter"[55]).
- Use of empirical correlations (e.g., certain bacteria have been observed to be likely to cause infection in burn patients).
- Use of "incidental" facts to discriminate cases (e.g., "a snake that swims with its head out of the water is a water moccasin"). Such discrimination depends on the sparseness of the domain (only certain snakes are possible).

[55]Parker, T., Rules of Thumb, Boston, MA: Houghton Mifflin Publishers, 1983.

Expert Knowledge

Expert knowledge is highly idiosyncratic:

- Build stair steps 7 inches high and 10 inches wide.
- Two times height plus width should equal 25 inches.
- Width times height should equal 72 inches.[56]

- Different rules may be generated for the same phenomena.
- The rules may have no fundamental validity and may give bad answers outside a limited domain of applicability.
- The rules generally work within the limits of applicability, but the expert often doesn't know what the limits are.

[56]Parker, T., Rules of Thumb, Boston, MA: Houghton Mifflin Publishers, 1983.

Choosing a Domain

A domain chosen for an expert system (especially a first one) should have the following characteristics:

- Task takes from a few minutes to a few hours for human experts.
- Specialized task (avoid commonsense reasoning).
- Expertise in the area exists and can be identified.
- An expert who is willing to commit significant amounts of time over a long period is available.
- Opportunity for large payoff.

Problem Characteristics

- Complexity: significant expertise required.
- Lack of algorithmic solution to the problem.
- Data may be unavailable or uncertain.
- "Judgment" may be used in reaching conclusion.
- Many different kinds of *knowledge sources* involved in performing task.

Rule-Based Systems

One of the most popular methods for representing knowledge is in the form of Production Rules. These are in the form of:

`if` *conditions* `then` *conclusion*

Example: MYCIN[57]

```
Rule 27
If 1) the gram stain of the organism is gram
      negative, and
   2) the morphology of the organism is rod, and
   3) the aerobicity of the organism is
      anaerobic,
Then: There is suggestive evidence (. 6) that
      the identity of the organism is
      Bacteroides.
```

[57]Shortliffe, Edward H., Computer Based Medical Consultations: MYCIN, American Elsevier, 1976. Buchanan, Bruce G. and Shortliffe, Edward H., Rule-Based Expert Systems, Addison-Wesley, 1984.

Advantages of Rules

- Knowledge comes in meaningful chunks.
- New knowledge can be added incrementally.
- Rules can make conclusions based on different kinds of data, depending on what is available.
- Rule conclusions provide "islands" that give multiplicative power.
- Rules can be used to provide explanations, control problem-solving process, check new rules for errors.

EMYCIN

EMYCIN was the first widely used expert system tool.

- Good for learning expert systems
- Limited in applicability to "finite classification" problems:
  - Diagnosis
  - Identification
- Good explanation capability
- Certainty factors

Several derivative versions exist.

Rule-Based Expert Systems[58]

MYCIN diagnoses infectious blood diseases using a backward-chained (exhaustive) control strategy.

The algorithm, ignoring certainty factors, is basically backchaining:

Given:

1. list of diseases, Goal-list
2. initial symptoms, DB
3. Rules

For each g $\in$ Goal-list do
If prove(g, DB, Rules) then Print ("Diagnosis:", g)

Function prove (goal, DB, Rules)
If goal $\in$ DB then return True
elseif $\exists r \in$ Rules such that r_{RHS} contains goal
then return provelist(LHS, DB, Rules)[59]
else Ask user about goal and return answer

[58]Shortliffe, E. Computer-based medical consultations: MYCIN. New York: Elsevier, 1976.
[59]provelist calls prove with each condition of LHS

Production Systems (OPS-5 family)

Production systems can be traced back to the *pandemonium* model of Selfridge, in which many *demons* observe a body of data; each responds when it sees the particular pattern in the data that it is looking for. The model is inherently highly parallel; it is intended to model the way the brain achieves intelligent and rapid responses to new information using relatively slow components.

A production system has two memories:

1. A *production memory*, or long-term memory, containing the production rules.
2. A *working memory*, or short-term memory, containing data about the current problem.

Recognize-Act Cycle

The production system operates on a forward chaining *recognize-act* cycle:

Recognize:
All productions are matched against all data to find the set of all productions that are satisfied by some subset of the data. (The same production might be satisfied by multiple sets of data.) The set of satisfied productions with data bindings is sometimes called the *conflict set*.

Act:
The satisfied productions (or, in implementations on a serial machine, one of them) are *fired*, which means their actions are executed. The actions generally make changes in the working memory. Production systems thus normally operate in a forward-chaining fashion.

Production System (OPS-5)

Recognize - Act Cycle:

1. **Match:** Evaluate LHS of all productions to see which are satisfied.
2. **Conflict Resolution:** Select one production. (If none, halt.)
3. **Act:** Perform RHS actions of selected production.

OPS-5 Data

OPS-5 has two separate memories:

1. Production memory (long-term memory)
2. Working memory (short-term memory for data of the current problem)

Data is stored in pre-declared record form, including the type of the data element.

```
(MATERIAL ^NAME  H2SO4
          ^COLOR COLORLESS
          ^CLASS ACID)
```

The names preceded by ^ are attribute names; internally, these are translated to numeric offsets within the record.

OPS-5 Rules

The following is a made-up rule in the style of rules used in R1/XCON:

```
(P
 Rule803
 (GOAL ^STATUS ACTIVE
       ^NAME   ASSIGN-POWER-SUPPLY
       ^CABINET <CAB>)
 (AMPS ^CABINET <CAB> ^VALUE {<A> > 5})
 (SPACE ^CABINET <CAB> ^VALUE {<S> > 7})
--->
 (ASSERT POWER-SUPPLY ^CABINET <CAB>
                      ^TYPE PS101)
 (REMOVE 1)
 (MODIFY 3 ^VALUE (COMPUTE <S> - 7))
 (MAKE GOAL ^STATUS ACTIVE ^NAME POWER-CORD)
)
```

Note how much of this rule involves control. This is typical of OPS-5 rules.

Conflict Resolution

If OPS-5 were actually to compute the conflict set (all rules that can be satisfied by all data in all combinations), the computational load would be enormous. Instead, OPS-5 computes the "derivative" of the conflict set, i.e., those rules newly enabled as a result of the changes made by the last rule, using the RETE algorithm[60]

Conflict Resolution Algorithm:

1. A rule instantiation can only fire once (prevents loops).
2. Rules whose left-hand side involves the data most recently added to working memory are favored (gives search a "depth-first" bias).
3. Rules with a more specific left-hand side (i.e., more clauses on the left-hand side) are favored.

[60]C. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem", *Artificial Intelligence*, vol. 19, no. 1 (sept. 1982), pp. 17-38. D. Miranker has an alternative algorithm called TREAT.

Evaluation of OPS-5

OPS-5 and variants are widely used expert system tools.

Advantage:

1. The tool does not greatly restrict what one can do.

Disadvantages:

1. Low-level code: the "assembly language" of expert system tools.
2. Kinky code: Although OPS-5 supposedly offers flexible control, rules tend to restrict this control by assertion of "control variables".

Free OPS-like program in C: **CLIPS**
`http://www.ghg.net/clips/CLIPS.html`

```
(defrule reactor_pressure_demon
  "Reactors blow if pressure too high"
  (object (is-a reactor)
          (name ?n)
          (pressure ?p&:(>= ?p 610)))
  =>
  (send ?n blown))
```

Reasoning Under Uncertainty

Human expertise is based on effective application of learned biases. These biases must be tempered with an understanding of strengths and weaknesses (range of applicability) of each bias.

In expert systems, a model of inexact reasoning is needed to capture the judgmental, "art of good guessing" quality of science.

In this section we discuss several approaches to reasoning under uncertainty.

- Bayesian model of conditional probability
- EMYCIN's method, an approximation of Bayesian
- Bayesian nets, a more compact representation used for multiple variables.

Bayes' Theorem

Many of the methods used for dealing with uncertainty in expert systems are based on Bayes' Theorem.

Notation:

$P(A)$	Probability of event A
$P(AB)$	Probability of events A and B occurring together
$P(A\|B)$	Conditional probability of event A given that event B has occurred

If A and B are *independent*, then $P(A|B) = P(A)$.

Expert systems usually deal with events that are *not* independent, e.g. a disease and its symptoms are not independent.

Bayes' Theorem

$P(AB) = P(A|B) * P(B) = P(B|A) * P(A)$

therefore $P(A|B) = P(B|A) * P(A)\ /\ P(B)$

Uses of Bayes' Theorem

In doing an expert task, such as medical diagnosis, the goal is to determine identifications (diseases) given observations (symptoms). Bayes' Theorem provides such a relationship.

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

Suppose: $A =$ Patient has measles, $B =$ has a rash
Then: $P(measles/rash) =$

$$P(rash/measles) * P(measles)/P(rash)$$

The desired diagnostic relationship on the left can be calculated based on the known statistical quantities on the right.

Joint Probability Distribution

Given a set of random variables $X_1 ... X_n$, an *atomic event* is an assignment of a particular value to each X_i.

The *joint probability distribution* is a table that assigns a probability to each atomic event. Any question of conditional probability can be answered from the joint.[61]

	Toothache	¬ Toothache
Cavity	0.04	0.06
¬ Cavity	0.01	0.89

Problems:

- The size of the table is combinatoric: the product of the number of possibilities for each random variable.
- The time to answer a question from the table will also be combinatoric.
- Lack of evidence: we may not have statistics for some table entries, even though those entries are not impossible.

[61] Example from Russell & Norvig.

Chain Rule

We can compute probabilities using a chain rule as follows:

$$P(A \wedge B \wedge C) = P(A|B \wedge C) * P(B|C) * P(C)$$

If some conditions $C_1 \wedge ... \wedge C_n$ are independent of other conditions U, we will have:

$$P(A|C_1 \wedge ... \wedge C_n \wedge U) = P(A|C_1 \wedge ... \wedge C_n)$$

This allows a conditional probability to be computed more easily from smaller tables using the chain rule.

Bayesian Networks

Bayesian networks, also called *belief networks* or *Bayesian belief networks*, express relationships among variables by directed acyclic graphs with probability tables stored at the nodes.[62]

[62]Example from Russell & Norvig.

Computing with Bayesian Networks

If a Bayesian network is well structured as a poly-tree (at most one path between any two nodes), then probabilities can be computed relatively efficiently.

One kind of algorithm, due to Judea Pearl, uses a message-passing style in which nodes of the network compute probabilities and send them to nodes they are connected to.

Several software packages exist for computing with belief networks.

A Heretical View

My own view is that CF combination algorithms are not a major issue.

Question: How accurate does the computation in the middle need to be, given that the input data are only accurate to (say) $\pm 10\%$?

It's hard to argue that extreme accuracy in the computation is required.

Remember:

- Use CF's as a last resort, when a good guess is the best you can do.
- Never trust a CF to have more than one digit of accuracy.

EMYCIN's Certainty Factors

EMYCIN's methods of doing Certainty Factor calculations represent a good set of engineering choices. They have been criticized, but represent a useful technique worthy of study.

Several kinds of CF's are involved:

- Data CF
- CF from antecedent of a rule
- CF due to rule as a whole
- Combination of CF's from multiple rules.

There is a further question of what a CF is supposed to mean.

Certainty Factor Meaning

Traditional probability values are on a scale of 0-1. Shortliffe argues this does not support "ruling out" reasoning of the kind done in medicine.

EMYCIN CF's are on a scale of -1 to +1. A CF combines both a "positive probability" and a "negative probability".

MB	0 - 1	Measure of Belief
MD	0 - 1	Measure of Disbelief
CF = MB - MD	-1 to 1	Certainty Factor
	-1	Definitely False
	0	No information, or cancellation
	+1	Definitely True

When a data parameter is True/False, "False" is represented as "True" with a CF of -1.

EMYCIN Data CF's

Each piece of data has a CF associated with it; even if the parameter is single-valued, there may be multiple possibilities:

COLOR = ((RED .6) (BLUE .3))

Data is referenced using predicates that differ on:

- the CF values that cause the predicate to be "true"
- the CF value returned by the predicate.

The two most commonly used predicates are:

- `SAME`: "True" if data $CF > .2$; returns data CF.
- `KNOWN`: "True" if data $CF > .2$; returns 1.0

EMYCIN Antecedent CF

The *antecedent* ("if part") of a rule is usually a conjunction of conditions, using the EMYCIN `$AND` function:

```
($AND (SAME CNTXT GRAM GRAMNEG)
      (SAME CNTXT MORPH COCCUS))
```

`$AND` operates as follows:

1. If any clause is false (`nil`) or has $CF \leq .2$, `$AND` returns false (`nil`). Thus, .2 is used as a cutoff threshold. Any data believed less strongly than .2 is considered to be false.
2. If every clause has $CF > .2$, `$AND` returns the minimum of the clause CF values.

There is also a function `$OR` that returns the maximum of its argument CF values.

Rule Certainty Factors

Premise:

```
($AND (SAME CNTXT SITE BLOOD)
      (SAME CNTXT GRAM GRAMNEG)
      (SAME CNTXT MORPH ROD)
      (SAME CNTXT BURNED))
```

Action:

```
 (CONCLUDE CNTXT
           IDENTITY PSEUDOMONAS-AERUGINOSA
           TALLY 400)
```

The result of the rule as a whole is calculated as follows:

1. `$AND` sets the global variable `TALLY` to the minimum CF of its components.
2. The Rule CF is `TALLY` times the CF specified in the `CONCLUDE` line, divided by 1000.

This forms the input to the CF Combination algorithm.

Certainty Factor Combination

When two sets of evidence imply the same conclusion, there is a need to compute the total certainty factor based on the certainties of the sets of evidence.

A CF combination method should be:

- Commutative: $A \cdot B = B \cdot A$
- Associative: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

This will make the resulting CF independent of the order in which pieces of evidence are considered.

Certainty Factor Combination

If a datum's previous certainty factor is CF_p and a new rule computes a certainty factor CF_n, the combined certainty factor is given by:

$$cfcombine(CF_p, CF_n) =$$

$$\begin{cases} CF_p + CF_n * (1 - CF_p) & CF_p > 0,\ CF_n > 0 \\ (CF_p + CF_n) \\ \quad /(1 - min(|CF_p|, |CF_n|)) & \text{signs differ} \\ -cfcombine(-CF_p, -CF_n) & CF_p < 0,\ CF_n < 0 \end{cases}$$

This algorithm has a desirable feature: it is associative and commutative; therefore the result is independent of the order in which rules are considered.

A CF of + 1 or -1 is dominant and sets the combined CF to that value.

Summary of CF Computations

- $CF > .2$ threshold
- `$AND` takes minimum CF in premise
- `conclude` CF $= CF_{premise} * CF_{rule}$
- CF combination algorithm

Examples:

```
If:     A (.6)
    and B (.3)
    and C (.4)
Then: conclude D tally 700
```

The resulting rule value for `D` is the minimum premise CF (.3) times the rule CF (.7), or 0.21 .

Suppose that two separate rules reach the same conclusion with CF's of 0.5 and 0.6 ; the resulting CF is $.5 + .6 * (1 - .5) = .8$. This could also be computed as $.6 + .5 * (1 - .6) = .8$.

Contradictions:

EMYCIN's CF calculations allow contradictory rules to cancel one another. While in logic a contradiction is intolerable, use of rules and exceptions, expressed as contradicting rules, seems to be the way humans often think.

Example: FUO program for diagnosing fever of unknown origin. Patient is a 17 year old female with persistent high fever, headache, lethargy, cardiac symptoms....

Results: Lung cancer (.81), Endocarditis (.7)

Endocarditis was the correct diagnosis. The physician expert remarked that lung cancer was consistent with the symptoms, but that patients that young never get lung cancer.

One way to handle this in EMYCIN is to add a rule that "rules out" certain cancers in young patients; the CF can be made a function of the patient's age.

Duplicate Rules

Duplicated rules in EMYCIN are harmful because each of the rules will fire; this will cause the CF's of the rules to be combined, giving a larger CF than was intended.

In a large expert system, it is easy for duplicate rules to be created by different rule-writers (or even the same one).

In logic, duplicated rules have no effect.

Rule Subsumption

A common programmer error is to leave out one or more clauses in the antecedent of a rule. This causes the rule to be over-broad in its application and may cause it to subsume other rules.

Example: SACON structural analysis consultant[63]

```
If: 1) The analysis error (in percent) that is
       tolerable is less than 5, and
    2) The non-dimensional stress of the
       substructure is greater than .5, and
    3) The number of cycles the loading is to be
       applied is greater than 10000
Then: It is definite (1.0) that fatigue is a
       phenomenon ...
```

A rule like this one, but without antecedent clause (3), was included. The bad rule subsumed this rule and two others (stress $>$ .7, cycles $>$ 1000; stress $>$ .9, cycles $>$ 100) with the same conclusion.

[63]Bennett, J. S. and Engelmore, R. S., "SACON: a Knowledge-based Consultant for Structural Analysis", Proc. IJCAI-79, pp. 47-49, Morgan Kaufmann Publishers.

Increasing Certainty

EMYCIN systems sometimes include rules of the form: $A \wedge B \rightarrow A$

Such a rule is logically redundant, but may serve to increase the CF of the conclusion based on additional evidence.

```
If; 1) The identity of the snake is
       rattlesnake, and
    2) It bites someone, and
    3) He dies
Then: There is strongly suggestive
      evidence (.8) that the identity of
      the snake is rattlesnake.
```

EMYCIN CF vs. Probability Theory

EMYCIN certainty factor calculations differ in significant ways from standard probability theory. These differences have attracted criticism, but often have good practical motivations.

CF Threshold: EMYCIN generally considers anything with CF $< .2$ to be false.

Q: "Can't this prevent several pieces of weak support from adding up to a significant level of support?"

A: Yes, it is possible, but doesn't seem to be a problem in practice.

The benefit of the CF threshold is that it keeps the system from asking a lot of dumb questions:

```
If: A (.01)
and B (.02)
and C (.001)
and D (as yet unknown) ...
```

In such a case, we don't want the system to ask questions about D, which is a most unlikely prospect. Asking dumb questions will quickly discourage potential users of the system.

Sensitivity Analysis

In general, sensitivity analysis attempts to determine the sensitivity of the output of a computation to small changes in the input.

An expert system should be relatively insensitive to small changes in the input or CF's; high sensitivity indicates bad design.

The sensitivity of MYCIN to small changes in CF values has been empirically tested; MYCIN was found to be relatively insensitive to CF values.

In part, this is due to the fact that MYCIN "plays it safe:" it treats for all organisms found with $CF > .2$.

Expert Systems vs. Decision Trees

There is a rule used by Expert Systems experts:

```
If:   There is a known algorithm to solve
      a problem,
Then: Use it.
```

So, if a decision tree will work for your problem, by all means use one.

The trouble is that decision trees work only for a relatively small class of problems, where:

1. All needed data can be obtained with certainty.
2. Data are discrete (Boolean or one of a fixed set of choices).
3. The structure of the problem is known and is fixed.
4. The problem can be "factored" well, preferably many times.
5. There is a single conclusion for each set of data.

Machine Learning [64]

Learning Techniques:

1. Learning from Examples
2. Learning from Advice
3. Learning from Analogy

[64]Cohen, P. R., Feigenbaum, E. A., Handbook of Artificial Intelligence, Volume 3. Morgan Kaufmann Publishers, 1982, p. 327.

Rule Induction

Motivation: acquire expert knowledge from examples of expert's problem solving.

Assumption is that it is easier for expert to demonstrate his expertise than to "tell all he knows".

Input to the induction algorithm is classified examples (which corresponds to I/O of human expert):

$< f_{11}, f_{12}, ..., f_{1n} > \quad classification_1$
$< f_{21}, f_{22}, ..., f_{2m} > \quad classification_2$

Output from the induction algorithm is a decision tree with features labeling interior nodes and classifications labeling leaves.

Sample Decision Tree:

Classifications = {chair, stool, table}
Features = {number of legs, armrests, height}
Domains:
number of legs = {3, 4}
armrests = {yes, no}
height = {tall, short}

New objects can be classified using the decision tree.

Example of Rule Induction[65]

Classifications	=	{ +, -}
Features	=	{height, hair, eyes}
Domains:		
height	=	{tall, short}
hair	=	{dark, red, blond}
eyes	=	{blue, brown}

Training set:
short, blond, blue: +
short, dark, blue: -
tall, dark, brown: -
tall, blond, brown: -
tall, dark, blue: -
short,blond, brown: -
tall, red, blue: +
tall, blond, blue: +

[65]Quinlan, "Learning Efficient Classification Procedures", *Machine Learning*, Morgan Kaufmann Publ., 1983.

Final Decision Tree with Classifications

Algorithm for Rule Induction

Instances:	a set of training instances
Features:	feature vector $(f_1, f_2, ..., f_n)$ input from teacher
Domains:	set of domains for Features $\{d_1, d_2, ..., d_m\}$
Classes:	set of classes $\{c_1, c_2, ..., c_k\}$ (simply $\{+, -\}$ for single concept learning.)

Function formrule (Instances, Features, Domains, Classes)
For some class $\in$ Classes
If all members of Instances fall into class
then return class
else f $\leftarrow$ select-feature (Features, Instances)
d $\leftarrow$ domain from set Domains corresponding to f
return a tree of the form:

Alternatives for `select-feature`

1. Random selection: guaranteed to give a decision tree which is consistent with the training set. No guarantee of optimality.
2. Information theoretic selection: select the feature which maximally partitions the set of instances. Heuristic for finding decision tree with minimum expected classification time.
3. Minimal cost selection: allow for the fact that some features are costly to evaluate. For example, body temperature is easier to determine than lung-capacity. Put least costly features high in the tree.

Limitations of Rule Induction

1. "Flat" classification rules produced with no justification facility
2. Lots of training examples are necessary
3. Training must be noise-free
4. Each training example must be described using all the features
5. Classifications and features are static sets
6. Rules produced do not distinguish between correlation and causality

Digital Low-Pass Filter

A simple mechanism that can be used to "learn" parameter values over time is the digital low-pass filter. A simple digital low-pass filter is defined by:

$$out_{i+1} = \alpha * in_i + (1 - \alpha) * out_i, \text{ where } \alpha << 1.$$

This filter will remove most short-term "noise" in the input, while passing through the long-term trend.

A filter like this was used to adjust weights assigned to heuristic feature detectors in Samuel's checker-player program.

An advantage of such a mechanism is that multiple parameter values can be "learned" simultaneously, despite the fact that the system's performance changes as the parameter values change.

Getting Knowledge From Expert

1. Watch (and videotape) the expert doing examples. Encourage expert to talk aloud about actions, strategy and reasoning behind conclusions. Ask questions to keep expert talking.
2. Focus on a test case and build a system to handle that case as soon as possible.
3. Review initial system with expert; fix as needed.
4. Add rules related to existing rules to expand coverage.
5. Try additional test cases; fix as errors are found.
6. Rewrite and restructure the whole system when needed.
7. The order in which the expert asks questions is an important clue to the strategy being used. Ordering is also an important component of expert knowledge in some domains, especially design.

Interaction with Expert

Test cases often reveal missing pieces of knowledge.

Example: Fever of Unknown Origin

Patient is 17-year-old female; persistent fever, headache, lethargy, cardiac symptoms, ...

Diagnoses: Lung cancer (.81), Endocarditis (7).

Correct diagnosis was endocarditis. Physician expert said the diagnosis of lung cancer was consistent with the symptoms; however, lung cancer would never be expected in a patient this young.

Result: new rules added to rule out certain cancers in young patients. Patient age can be used to determine certainty factor of ruling out lung cancer.

Conceptual Islands

An important thing to look for in gathering knowledge about a domain from an expert is "conceptual islands:" intermediate conclusions that have special meaning in the domain. Often these islands have specialized terminology associated with them.

Example: Compromised Host in MYCIN

A compromised host is a patient who has been weakened and therefore cannot fight off infections as well as a normal person.

Advantages of Conceptual Islands

Conceptual Islands reduce the number of rules required and aid robustness.

Expansion with Conceptual Islands

Islands give multiplicative power. New rules that reach a conceptual island become effective with all the rest of the system's knowledge.

Orthogonal Knowledge Sources

Often the best way to get discriminating power is to find another knowledge source that is "orthogonal to" the existing ones (i.e., discriminating on a basis unrelated to the existing set of data.

Example: Nuclear magnetic resonance data used in conjunction with mass spec data in DENDRAL.

Neuron

- *cell body:* 5 - 100 microns diameter
- *dendrites:* branching structures that receive inputs from other neurons
- *axon:* long process (up to 1 meter) extending from the cell body; carries an output pulse
- *myelin sheath:* insulates the axon
- *axon terminal:* interface to the next neuron's dendrite
- *synapse:* connection between neurons
- *afferent* neurons carry signals toward the brain; *efferent* neurons carry signals away from the brain.

Neuron Firing

- When sufficiently stimulated, neuron fires a discrete pulse.
- Electric pulse travels down the axon (90 m/sec).
- Synaptic bulb releases neurotransmitter molecules.
- Neurotransmitters diffuse across the synaptic cleft.
- Neurotransmitters bind to receptors on dendrite and stimulate or inhibit the target neuron.
- Scavenger chemicals clean up the neurotransmitters.
- Neuron must recharge before it can fire again.
- Strength of signal is determined by *rate* of firing (max 1 KHz).

Function of Brain Regions[66]

[66]Illustrations in this section are from *Scientific American*, Sept. 1979 and Sept. 1992, and from *Science*.

Somatotopy: Sensory and Motor

Somatotopy is the principle that adjacent areas of body or retina are mapped to adjacent areas of brain.

Early Experimental Work

Camillo Golgi: developed Golgi stain that makes individual neurons visible.

Santiago Ramón y Cajal performed first detailed study of brain and nervous system; established that neurons are separate cells.

Function from Brain Injury

A. R. Luriia studied brain-injured Russian soldiers during World War II, correlating their psychological deficits with the location of injury.

Injury to Broca's area and Wernicke's area cause distinct *aphasias* (speech disorders).

Positron Emission Tomography

Positron emission tomography (PET) and functional magnetic resonance imaging (fMRI) provide real-time pictures of brain activity.

- Radio-labeled glucose is injected into arteries that feed the brain.
- Blood flow is modulated by brain activity.
- Pictures of radiation concentration show which brain areas are active

Micro-electrodes

Micro-electrodes inserted into the brain of an experimental animal make it possible to observe the firing of a single neuron.

Experiments can then be done to show what will make the neuron fire.

David Hubel and Torsten Wiesel used this technique to study the visual cortex of the cat.

Hubel and Wiesel: Visual Cortex

- Retinal ganglion: center-surround difference detection
- Line detection: about 6° resolution
- Moving line detection
- (presumably) Moving object detection

Visual Processing Areas

Vision

- Many processing units, highly interconnected
- Retinal cells are denser towards center

Mental Rotation

The very existence of mental images has been controversial.

Shepard and Metzler[67] showed that response time in matching rotated 3-D images is proportional to angle of rotation.

Georgopoulos *et al.*[68] demonstrated a mental rotation in a monkey.

[67]Shepard, R. N., and Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science*, 171, 701-703.

[68]Georgopoulos, A. P., *et al.*, "Mental Rotation of the Neuronal Population Vector", *Science*, vol. 243, no. 4888 (13 Jan. 1989), pp. 234-236.

Somatotopic Processing

Brain processing of information seems to be based on:

- Somatotopic mapping
- Parallel processing
- Massive interconnection
- Successive layers of processing

Machine Vision

Vision can be thought of as "inverse optics" or "inverse graphics:" recovery of the 3-D structure and properties of a scene from a 2-D image.

- Locations of objects in 3-D
- Identification of objects
- Motion

In a sense, this is a mathematically impossible problem because information is lost in making the 2-D image from 3-D reality.

Digitization

Digitization refers to the input of a picture in digital form, typically as an array of intensity values. For color, there will be three intensity values per *pixel*.

- *Lots* of data: 3 megabytes for 1024 x 1024 color picture.
- Introduces "noise" because digitized values are inexact.
- Introduces edge artifacts at boundaries between digitized values.

Low-Pass Filtering

Low-pass filtering is a "smoothing" operation. It reduces noise, blurs edges, and introduces delay in the time domain. Low-pass operations include averaging, summation, integration, and convolution with a Gaussian. Averaging n samples reduces noise by $\sqrt{n}$. An easy low-pass filter in one dimension:

$$Out_i = \alpha \cdot In_i + (1 - \alpha) \cdot Out_{i-1} \qquad where\ \alpha \ll 1$$

High-Pass Filtering

"Contrast, the relative difference between light and dark, is the coinage of the visual system." – Lawrence Cormack

High-pass filtering or edge detection is a "sharpening" operation.

- Sharpens edges
- Increases noise

High-pass operations include differentiation, subtraction, convolution with the second derivative of a Gaussian.

Convolution

The *convolution* of two picture functions g and f, denoted $g * f$, is defined as:

$$g * f(x, y) = \int \int_{-\infty}^{\infty} g(u, v) \cdot f(x - u, y - v) du\ dv$$

For example, the image recorded by a camera is the convolution of the original image with the point spread function of the camera optics.

If the function decays rapidly to zero outside a local area, convolution can be approximated by applying a grid-like *operator* to the image:

1	1	1
0	0	0
-1	-1	-1

Such an operator can rapidly be applied to a whole image by special hardware, either operating on a stored image or on a raster scan.

Feature Detection

Hubel and Wiesel showed that the cells in the retinal ganglion compute a center-surround difference function.

Marr proposed that the right function to use is the second derivative of a 2-D Gaussian, or *sombrero function*.

This function detects places where the image intensity changes.

Hough Transform

Edges (boundaries between areas of different intensity) are important for image understanding. Edges are often approximately straight. However, they may be discontinuous, and there may be noise.

The *Hough transform* finds edges despite these problems.

- For each image feature (edge) point that is above a threshold, map that point to each (r, θ) pair that represents a line through that point, and add 1 to the counter for (r, θ).
- The (r, θ) counters are *histograms*: high values indicate where lines are.
- Identify the exact locations and endpoints of strong lines from the image.

Generalized Hough Transform

The principle of the Hough transform can be generalized to find line patterns other than straight lines, *e.g.*, circles.

- Map the image points into points in parameter space
- Threshold the histograms in parameter space
- Identify desired curves in original image.

This approach requires much more storage and computation as the number of parameters increases.

Segmentation

Segmentation is the process of breaking the image into regions that are more or less homogeneous or represent regions of objects.

- For polygonal scenes, the lines found by the Hough transform can be used as input to Waltz filtering.
- Regions can sometimes be found by identifying particular colors and grouping pixels of those colors into polygons.

Shape from Shading

For some objects, variation in shading is an important clue to surface orientation.

Lambert's Cosine Law

Johann Heinrich Lambert (1728-1777) stated the law:

> Flux per unit solid angle leaving a surface in any direction is proportional to the cosine of the angle between that direction and the normal to the surface.

A surface that obeys Lambert's law is called *lambertian*; "flat" paint produces such a surface. The *albedo* of the surface is the fraction of incoming light that is reflected. Illumination is spread across the surface proportional to the cosine of the angle with the surface normal. A reflective surface is *specular* (mirror-like).

The cosine law can be solved to determine the surface normals from observed intensity, thus recovering object shape from shading. Assumptions must be added to constrain the problem enough to make it mathematically solvable.

Non-planar objects may be represented as generalized cylinders.

Object Identification

Object identification is difficult and still unsolved.

- Template matching
- Picture grammar
- Graph matching
- Invariant properties: number of holes, moment of inertia, ratio of boundary to area, aspect ratio.

Motion

Understanding moving objects is very important.

- Subtracting two pictures at adjacent time points gives motion edges.
- A "velocity image" or array of velocity vectors can be computed.
- Lines with the same velocity can be grouped as an object.

Other Sensory Modes

Other sensors besides TV images can be used.

- Laser stripes: illuminating an object with a plane of laser light and photographing it from two angles gives 3-D shape.
- Sonar detectors can be used.

Adding Semantics to Vision

Knowing what is expected can help identify what is seen, *e.g.,* in an aerial photograph:

- Roads and rivers are expected to be continuous.
- There should be a bridge at the intersection of a river and a road.
- Houses are near a street, parallel with street, have a driveway to street.

Practical Vision

Machine vision is used in industry for several tasks:

- Orientation of chassis for robot assembly.
- Reading results of test programs from screen of computer.
- Inspection of printed circuit boards.

Theory and Practice

We should not assume that practical results will flow out of successful theories rather than vice-versa. In the past, it has at least as often been the case that successful theories have been constructed on the basis of engineering observations.

– Y. Aloimonos and A. Rosenfeld

Driving a Car

Ernst D. Dickmanns, "Vehicles capable of dynamic vision: a new breed of technical beings?", *Artificial Intelligence*, vol. 103, pp. 49-76, 1998.

This system performs autonomous road vehicle guidance in public traffic on freeways at speeds beyond 130 km/h.

- Integrates AI methods with system dynamics and control engineering
- Integration of multiple sensors (vision, gyro, accelerometers)
- Several levels of representation:
  - Low-level: perception and motor controls
  - High-level: symbolic representations of plans, goals, environment
- Feedback from higher levels of perception to lower levels to guide perception, gain efficiency.

Driving: Architecture

Driving: Detailed Architecture

Control of Spacecraft

Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams, "Remote Agent: to boldly go where no AI system has gone before," *Artificial Intelligence*, vol. 103, pp. 49-76, 1998.

This system autonomously controls the Deep Space One spacecraft.

- Constraint-based temporal planning and scheduling
- robust multi-threaded execution
- model-based mode identification and reconfiguration
- predicate calculus representation
- self-diagnosis using state transition models with probabilities
- multiple levels of representation

Remote Agent: Architecture

Engine / Valve Configurations

Understanding Machines from Movies

Tzachi Dar, Leo Joskowicz, and Ehud Rivlin, "Understanding mechanical motion: From images to behaviors," *Artificial Intelligence*, vol. 112, pp. 147-179, 1999.

This system produces descriptions of planar fixed axes mechanical motion from image sequences.

- Most machines are combinations of machine elements that can be catalogued.
- Machine vision gives observed motion in images
- Observed motions suggest basic components
- Simultaneous motions suggest relationships
- Component relations are parsed into a description of the machine

Understanding Machines: Example