Course Topics

• Introduction
• Clojure
• List and tree representation
• Tree recursion and tree search
• Evaluation and interpretation
• Pattern matching, substitution, rewrite rules
• Symbolic mathematics: algebra, calculus, unit conversion
• Program generation from trees
• Predicate calculus, rule representation
• Backchaining and Prolog; program composition
• Rule-based expert systems
• Semantic grammar and Natural language interfaces
• Functional programs, MapReduce
• Symbolic programming in ordinary languages
Symbolic Programming

*Symbols* are words that refer to other things: their *referents*:

- Variable names in programming languages
- Variables in equations
- English words

Symbolic Programming manipulates structures of symbols, typically represented as trees:

- translate a program to another language
- specialize a generic program
- solve equations
- convert units of measurement
- translate English into a database query
- compose program components
Functional Programming

A functional program is one with no side effects:

• changing a global variable
• updating a database
• printing

If we call \( \text{sin}(x) \), it will just return a value, but will have no side effects.

Functional programming does everything by composition of functions:

\[
\text{guacamole: season(mash(slice(peel(wash(avocado)))))}
\]

Functions are composed so that the output of one function is the input of the next function.

Functional programming works well with distributed cloud computing: the function can be replicated on many servers and executed in parallel on massive amounts of data.
Clojure

Clojure (clojure.org) is a Lisp dialect that is compiled into Java Virtual Machine (JVM) code. Since it runs on the JVM, Clojure connects to the Java ecosystem and has been used for serious web applications by large companies. There is also a ClojureScript that rests on JavaScript.

Our Clojure system just handled its first Walmart black Friday and came out without a scratch.
- Anthony Marcar, Senior Architect, WalmartLabs

\footnote{The name Clojure is a pun on closure, a runtime object that combines a function and its environment (variable values), with a j for Java.}
Why Symbolic and Functional Programming?

Symbolic and functional programming provide a great deal of power to do interesting and large applications easily.

Problem: Given known variable values, find appropriate physics equations and solve them to find a desired variable value.

- Ordinary programming: 5 person-years by two PhD students; 25,000 lines of C++.
- Symbolic Programming: 200 lines, a few days.

Not only does Walmart run its huge e-commerce operation using Clojure, they built it with only 8 developers.

blog.cognitect.com/blog/2015/6/30/walmart-runs-clojure-at-scale
Class Projects

This course will involve interesting projects that will give students a taste of what can be done with symbolic programming:

- Introduction to Clojure, Tree Recursion
- Symbolic Math: Evaluation, Algebra, Calculus, Unit Conversion
- Pattern Matching, Substitution, Rewrite Rules
- Program Generation from Trees
- Backchaining and Prolog: finding a composition of functions that achieves a desired program
- Rule-based Expert Systems
- Semantic Grammar and Natural Language Interfaces
- Functional Programs over massive data: MapReduce and Clojure
- Symbolic programming in ordinary languages
Clojure

We will learn only a fraction of Clojure, mainly the small amount we will need to handle lists and trees (composed of nested lists).

We will use the List/Tree as our primary data structure and as a notation that allows trees to be printed and to be input as data.

Like Lisp, Clojure is homoiconic, which means that Clojure code and Clojure data are the same thing. This means that a program can write (and then execute) code at runtime.

A list is written inside parentheses (the external notation). A function call is written with the function name as the first thing in the list, followed by arguments:

\[(+ \ i \ 2) \quad ; \ i + 2\]
Quotation

Since we will be working with symbols, we must be able to distinguish between use of a symbol as a name that denotes a value (its binding) and a reference to the symbol itself.

A single-quote symbol ’ is used to denote the symbol itself; this is a read macro that expands to a pseudo-function quote.

> (def car ’ford) ; > denotes input to Clojure

car
ford

> ’car
car

> (quote car)
car

> (eval (quote car))
car

> ford
List

A list is made of two-element data structures, called conses after the function cons that makes them.

- a link, called rest, or pointer to the next element; to () or empty list if there is no next element.
- some contents, called first, as needed for the application. The contents could be a number, a symbol, or a pointer to another linked list.

A list is written inside parentheses:

\[
\text{(list 'a 'b 'c)} \quad \rightarrow \quad (a b c)
\]

\[
\text{(list (+ 2 3) (* 2 3))} \quad \rightarrow \quad (5 6)
\]

We will use the parentheses notation for lists, and nested lists for trees.
Constructing a List

A list can be built up by linking new elements onto the front. The function `(cons item list)` makes a new list element or cons cell pointing to *item* and adds it to the front of *list*:

```
(cons 'a nil)   ->  (a)
(cons 'a '())   ->  (a)
(cons 'a (list)) ->  (a)
(cons 'a '(b c)) ->  (a b c)
```

An easy way to think of `(cons x y)` is that it makes a new cons box whose first is *x* and whose rest is *y*.

Note that there is *structure sharing* between the result of *cons* and the list that is the second argument of *cons*.

```
(def x '(a b c))
(def y (cons 'd x))
>x
(a b c)
>y
(d a b c) ; x and y share (a b c)
```
Access to Parts of a List

The two fields of a cons cell are called first, the first item in the list, and rest, the rest of the list after the first element.\(^2\)

\[
\begin{align*}
(\text{first } '(a b c)) & \rightarrow a \\
(\text{rest } '(a b c)) & \rightarrow (b c) \\
(\text{first } (\text{rest } '(a b c))) & \rightarrow b \\
(\text{second } '(a b c)) & \rightarrow b \\
(\text{first } (\text{rest } (\text{rest } '(a b c)))) & \rightarrow c \\
(\text{defn third} [x] (\text{first } (\text{rest } (\text{rest } x)))) & \rightarrow c \\
(\text{third } '(a b c)) & \rightarrow c \\
(\text{rest } (\text{rest } (\text{rest } '(a b c)))) & \rightarrow ()
\end{align*}
\]

\(^2\)In the original Lisp, first was called car, and rest was called cdr.
List Access Functions

There are easy rules for the access functions using the parenthesized representation of lists:

**first** returns the first thing in a list. A thing is:

- a basic item such as a number or symbol
- a balanced pair of parentheses and everything inside the parens, no matter how large.

(first '(a b c)) -> a

(first '((a b) c)) -> (a b)

**rest** returns the rest of a list after the first thing. Simply move the left parenthesis to the right past the first thing.

(rest '(a b c)) -> (b c)

(rest '((a b) c)) -> (c)

(rest (rest '((a b) c))) -> ()
IF statement

The if statement in Clojure has the forms:

(if test thencode)

(if test thencode elsecode)

The test is interpreted as follows:

- false or nil are considered to be false.
- anything else is considered to be true, including ( ).

The if statement returns the value of either the thencode or elsecode.

(if true 2 3) -> 2
(if false 2 3) -> 3
(if '() 2 3) -> 2
(if (+ 2 3) 2 3) -> 2
Recursion

A *recursive* program calls itself as a subroutine. Recursion allows one to write programs that are powerful, yet simple and elegant. Often, a large problem can be handled by a small program which:

1. Tests for a *base case* and computes the value for this case directly.
2. Otherwise,
   (a) calls itself recursively to do *smaller* parts of the job,
   (b) computes the answer in terms of the answers to the smaller parts.

```clojure
(defn factorial [n]
  (if (<= n 0)
    1
    (* n (factorial (- n 1))) )
)
```

**Rule:** Make sure that each recursive call involves an argument that is *strictly smaller* than the original; otherwise, the program can get into an *infinite loop*.

A good method is to use a counter or data whose size decreases with each call, and to stop at 0; this is an example of a *well-founded ordering*.
Designing Recursive Functions

Some guidelines for designing recursive functions:

1. Write a clear definition of what your function should do, including inputs, outputs, assumptions. Write this definition as a comment above the function code.

2. Identify one or more base cases: simple inputs for which the answer is obvious and can be determined immediately.

3. Identify the recursive case: an input other than the base case. How can the answer be expressed in terms of the present input and the answer provided by this function (assuming that it works as desired) for a smaller input?

There are two common ways of making the input smaller:

- Remove a piece of the input, e.g. remove the first element from a linked list.
- Cut the input in half, e.g. follow a branch of a tree.
Design Pattern for Recursive Functions

A design pattern is an abstracted way of writing programs of a certain kind. By learning design patterns, you can write programs faster and with fewer errors.

A design pattern for recursive functions is:

```clojure
(defn myfun [arg]
  (if (basecase? arg)
    (baseanswer arg)
    (combine arg (myfun (smaller arg))))
```

In this pattern,

- `(basecase? arg)` is a test to determine whether `arg` is a base case for which the answer is known at once.
- `(baseanswer arg)` is the known answer for the base case.
- `(combine arg (myfun (smaller arg)))` computes the answer in terms of the current argument `arg` and the result of calling the function recursively on `(smaller arg)`, a reduced version of the argument.

**Exercise:** Show how the factorial function corresponds to this design pattern.
Recursive Processing of List

Recursive processing of a list is based on a base case (often an empty list), which usually has a simple answer, and a recursive case, whose answer is based on a recursive call to the same function on the rest of the list.

As an example, we can recursively find the number of elements in a list. (The Clojure function that does this is called count.)

```
(defn length [lst]
  (if (empty? lst) ; test for base case
    0 ; answer for base case
    (+ 1
      (length (rest lst)))) ) ) ; recursive call
```
Recursive List Design Pattern

(defn fn [lst]
  (if (empty? lst) ; test for base case
        baseanswer ; answer for base case
    (some-combination-of
      (something-about (first lst))
      (fn (rest lst))) ) ) ; recursive call

The recursive version is often short and elegant, but it has a potential pitfall: it requires $O(n)$ stack space on the function call stack. Many languages do not provide enough stack space for 1000 calls, but a linked list with 1000 elements is not unusual.
Tail Recursive Processing of List

A function is *tail recursive* if it either

- returns an answer directly, e.g `return 0;`
- the answer is *exactly* the result of a recursive call, `return myself(something);`

Tail recursion often involves the use of an extra function with extra variables as parameters: think of picking apples and putting them into a bucket as you go; the bucket is an extra variable. The main function just initializes the extra variables, while the helper function does the work.\(^3\)

```clojure
(defn lengthb [lst answer]
  (if (empty? lst) ; test for base case
    answer ; answer for base case
    (lengthb (rest lst) ; recursive call
      (+ answer 1))) ) ) ; update answer

(defn length [lst]
  (lengthb lst 0)) ; init extra variable
```

\(^3\)Note that we defined the helper function first: otherwise, the Clojure compiler will generate an error when the main function calls it before it is defined.
Tail Recursive List Design Pattern

(defn fnb [lst answer]
    (if (empty? lst)  ; test for base case
        answer  ; answer for base case
        (fnb (rest lst)
            (some-combination-of
                answer
                (something-about (first lst))))))

(defn fn [lst] (fnb lst answerinit))

A smart compiler can detect a tail-recursive function and compile it so that it is iterative and uses $O(1)$ stack space; this is called tail-call optimization. Unfortunately, Java and thus Clojure are not this smart.
Constructive Tail Recursive Reverse

reverse makes a new linked list whose elements are in the reverse order of the original list; the original is unchanged.

(reverse '(a b c)) -> (c b a)

This function takes advantage of the fact that cons creates a list in the reverse order of the conses.

(defn trrevb [lst answer]
  (if (empty? lst)
    answer
    (trrevb (rest lst)
      (cons (first lst) answer))
  ))

(defn trrev [lst] (trrevb lst '()))
Tail Recursive Reverse Execution

>(trrev '(a b c d))
1> (TRREVB (A B C D) NIL)
2> (TRREVB (B C D) (A))
3> (TRREVB (C D) (B A))
4> (TRREVB (D) (C B A))
5> (TRREVB NIL (D C B A))
<5 (TRREVB (D C B A))
<4 (TRREVB (D C B A))
<3 (TRREVB (D C B A))
<2 (TRREVB (D C B A))
<1 (TRREVB (D C B A))
(D C B A)
Append

`append` concatenates two lists to form a single list. The first argument is copied; the second argument is reused (shared).

```
(append '(a b c) '(d e))  ->  (a b c d e)
```

```
(defn append [x y]
  (if (empty? x)
    y
    (cons (first x)
          (append (rest x) y)) )
)
```

This version of `append` append is simple and elegant, but it takes $O(n_x)$ stack space.
Set as Linked List

A linked list can be used as a representation of a set.\(^4\) member (written \(\in\)) tests whether a given item is an element of the list. member returns the remainder of the list beginning with the desired element, although usually member is used as a predicate to test whether the element is present or not.

\[
\begin{align*}
\text{(member 'dick '(tom dick harry))} & \quad \rightarrow \quad \text{(dick harry)} \\
\text{(member 'fred '(tom dick harry))} & \quad \rightarrow \quad \text{nil}
\end{align*}
\]

\[
\text{(defn member [item lst]}
\begin{align*}
\quad \text{(if (empty? lst)}} & \quad \text{nil} \\
\quad \text{(if (= item (first lst))}} & \quad \text{lst} \\
\quad \text{(member item (rest lst))}} & \quad ))
\end{align*}
\]

\(^4\)Clojure has sets as a built-in type, which is more efficient than what is shown here.
Intersection

The intersection (written $\cap$) of two sets is the set of elements that are members of both sets.

\[
\text{(intersection '}(a \ b \ c) \ '}(a \ c \ e)) \rightarrow (c \ a)
\]

\[
\text{(defn intersection [x y]
  (if (empty? x)
    ()
    (if (member (first x) y)
      (cons (first x)
        (intersection (rest x) y))
      (intersection (rest x) y))
  ))
\]

If the sizes of the input lists are $m$ and $n$, the time required is $O(m \cdot n)$. That is not very good; this version of intersection will only be acceptable for small lists.
Tail-Recursive Intersection

(defn intersecttrb [x y answer]
  (if (empty? x)
    answer
    (intersecttrb (rest x) y
    (if (member (first x) y)
      (cons (first x) answer)
      answer)))))

(defn intersecttr [x y] (intersecttrb x y '()))

>(intersecttr '(a b c) '(a c e))

1> (INTERSECTTR (A B C) (A C E))
  2> (INTERSECTTRB (A B C) (A C E) NIL)
  3> (INTERSECTTRB (B C) (A C E) (A))
  4> (INTERSECTTRB (C) (A C E) (A))
  5> (INTERSECTTRB NIL (A C E) (C A))
  <5 (INTERSECTTRB (C A))
  <4 (INTERSECTTRB (C A))
  <3 (INTERSECTTR (C A))
  <2 (INTERSECTTRB (C A))
  <1 (INTERSECTTR (C A))
  (C A)
Union and Set Difference

The *union* (written $\cup$) of two sets is the set of elements that are members of either set.

\[(\text{union } '(a \ b \ c) \ '(a \ c \ e)) \rightarrow (b \ a \ c \ e)\]

The *set difference* (written $-$) of two sets is the set of elements that are members of the first set but not the second set.

\[(\text{set-difference } '(a \ b \ c) \ '(a \ c \ e)) \rightarrow (b)\]

Note that set difference is asymmetric: unique members of the second set, such as e above, do not appear in the output.
Association List

An association list or alist is a simple lookup table or map: a linked list containing a key value and some information associated with the key.\(^5\)

```
(assocl 'two '((one 1) (two 2) (three 3)))
-> (two 2)
```

```
(defn assocl [key lst]
  (if (empty? lst)
    nil
    (if (= (first (first lst)) key)
      (first lst)
      (assocl key (rest lst)) )) )
```

\(^5\)The function assoc, traditionally the name of this function in Lisp, is used for a Clojure lookup on the built-in map datatype. The Clojure version would be much more efficient for a large map.
Trees

A tree is a kind of graph, composed of nodes and links, such that:

• A link is a directed pointer from one node to another.
• There is one node, called the root, that has no incoming links.
• Each node, other than the root, has exactly one incoming link from its parent.
• Every node is reachable from the root.

A node can have any number of children. A node with no children is called a leaf; a node with children is an interior node.

Trees occur in many places in computer systems and in nature.
Arithmetic Expressions as Trees

Arithmetic expressions can be represented as trees, with operands as leaf nodes and operators as interior nodes.

\[ y = m \times x + b \quad (= y \; (+ \; (* \; m \; x \; b))) \]
Computer Programs as Trees

When a compiler parses a program, it often creates a tree. When we indent the source code, we are emphasizing the tree structure.

```plaintext
if ( x > y )
    j = 3;
else
    j = 1;
```

This is called an *abstract syntax tree* or *AST*. 
English Sentences as Trees

Parsing is 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.

The speaker wants to communicate a structure, but must make it linear in order to say it. The listener needs to re-create the structure intended by the speaker. Parts of the parse tree can then be related to object symbols in memory.
Representations of Trees

Many different representations of trees are possible:

- Binary tree: contents and left and right links.
- First-child/next-sibling: contents, first-child, next sibling.
- Linked list: the first element contains the contents, the rest are a linked list of children.
- Implicit: a node may contain only the contents; the children can be generated from the contents or from the location of the parent.
First-Child / Next-Sibling Tree

A node may have a variable number of children. Dedicating many links would be wasteful if the average number of children is much smaller than the maximum, and it would still limit the possible number of children.

Luckily, we can use the same structure as the binary tree, with just two links, and have unlimited children with no wasted space.

\[ y = m \times x + b \]

Tree: Representation:

The layout of nodes in this kind of tree is the same as for a traditional tree; the right-hand links “fall down” and become horizontal.

Down arrows represent the first child, while side arrows represent the next sibling.
Linked List Tree

We can think of a linked list as a tree. The first node of the list contains the contents of the node, and the rest of the list is a list of the children. The children, in turn, can be lists; a non-list is a leaf node. This is similar to first-child / next-sibling.

\[ (= Y (+ (* M X) B)) \]

\[
\begin{array}{c}
Y \\
+ \\
M \\
\star \\
= \\
\end{array}
\quad
\begin{array}{c}
op \\
lhs \\
rhs \\
\star \\
B \\
X \\
\end{array}
\]

(defn op [e] (first e)) ; operator
(defn lhs [e] (second e)) ; left-hand side
(defn rhs [e] (first (rest (rest e)))); right
Binary Tree Recursion

While recursion is not always the best method for linked lists, it usually is the best method for trees. We don’t have a problem with stack depth because depth is only $O(\log(n))$ if the tree is balanced.

Suppose that we want to add up all the numbers, or collect a set of symbols, in a Lisp tree.

```lisp
(defn addnums [tree]
  (if (and (list? tree) ; interior node
           (not (empty? tree)))
    (+ (addnums (first tree))
       (addnums (rest tree)) )
    (if (number? tree) ; leaf node
        tree
        0) ) )

(defn symbolset [tree]
  (if (and (list? tree) ; interior node
           (not (empty? tree)))
    (union (symbolset (first tree))
           (symbolset (rest tree)) )
    (if (symbol? tree)
        (list tree)
        '() ) )
```
Design Pattern: Binary Tree Recursion

This pattern is like the one for lists, except that it calls itself twice for interior nodes. This is essentially the same as the divide-and-conquer design pattern.

(defn myfun [tree]
  (if (interior? tree)
    (combine (myfun (first tree)) ; left
              (myfun (rest tree))) ; right
    (baseanswer tree) )); leaf node

(defn addnums [tree]
  (if (and (list? tree) ; interior node
            (not (empty? tree)))
    (+ (addnums (first tree))
      (addnums (rest tree)))
    (if (number? tree) ; leaf node
        tree
        0)) )
Flattening Binary Tree

An ordered binary tree can be flattened into an ordered list by a backwards inorder traversal. We do the inorder backwards so that pushing onto a stack (using \texttt{cons}) can be used to accumulate the result. This accumulation of a result in an extra variable is similar to tail recursion.

\begin{verbatim}
(defn flattenbtb [tree result]
  (if (and (list? tree) ; interior node
            (not (empty? tree)))
    (flattenbtb (lhs tree) ; 3. L child
                (cons (op tree) ; 2. parent
                       (flattenbtb
                        (rhs tree) ; 1. R child
                       result)))
    (if (not (list? tree))
      (cons tree result)
      result))
)

(defn flattenbt [tree] (flattenbtb tree '()))
\end{verbatim}
Examples: Flattening Tree

> (flattenbt '(cat (bat ape bee) (elf dog fox)))

(ape bat bee cat dog elf fox)

> (flattenbt '(= y (+ (* m x) b)))

(y = m * x + b)
Tracing Flattening Binary Tree

> (flattenbt '(cat (bat ape
        bee)
        (elf dog
        fox)))

1> (FLATTENBT (CAT (BAT APE BEE) (ELF DOG FOX)))
  2> (FLATTENBTB (CAT (BAT APE BEE) (ELF DOG FOX)) NIL)
   3> (FLATTENBTB (ELF DOG FOX) NIL)
      4> (FLATTENBTB FOX NIL)
         <4 (FLATTENBTB (FOX))
      4> (FLATTENBTB DOG (ELF FOX))
         <4 (FLATTENBTB (DOG ELF FOX))
         <3 (FLATTENBTB (DOG ELF FOX))
   3> (FLATTENBTB (BAT APE BEE) (CAT DOG ELF FOX))
      4> (FLATTENBTB BEE (CAT DOG ELF FOX))
         <4 (FLATTENBTB (BEE CAT DOG ELF FOX))
      4> (FLATTENBTB APE (BAT BEE CAT DOG ELF FOX))
         <4 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
         <3 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
         <2 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
         <1 (FLATTENBT (APE BAT BEE CAT DOG ELF FOX))
(APE BAT BEE CAT DOG ELF FOX)
Postorder

The Lisp function eval evaluates a symbolic expression. We can write a version of eval using postorder traversal of an expression tree with numeric leaf values. Postorder follows the usual rule for evaluating function calls, i.e., arguments are evaluated before the function is called.

```
(defn myeval [x]
  (if (number? x)
    x
    (apply (eval (op x)) ; execute the op
     (list (myeval (lhs x))
     (myeval (rhs x)))) ) )
```

```
> (myeval '(* (+ 3 4) 5))
1> (MYEVAL (* (+ 3 4) 5))
  2> (MYEVAL (+ 3 4))
    3> (MYEVAL 3)
    <3 (MYEVAL 3)
    3> (MYEVAL 4)
    <3 (MYEVAL 4)
    <2 (MYEVAL 7)
    2> (MYEVAL 5)
    <2 (MYEVAL 5)
    <1 (MYEVAL 35)
35
```
Depth-first search (DFS) is so named because the recursion goes deep in the tree before it goes across. The diagram shows the order in which nodes are examined.

Sometimes, the search will quit and return an answer when a node is either a terminal failure node or a goal node. In other cases, the entire tree will be traversed.

A stack composed of stack frames is used to contain variables of the current node and all of its ancestors, back to the top of the tree. This stack will grow and contract as the tree is traversed, with a maximum stack size equal to tree depth.

Depth-first search is often preferred because of its low $O(\log(n))$ storage requirement. We will never run out of storage in any reasonable amount of computer time.

A recursive program typically uses a depth-first order.
Pattern Matching Overview

We have emphasized the use of design patterns in writing programs. We would like to use patterns automatically to generate, improve, or transform programs, equations, and other tree-like data structures.

We will use rewrite rules, each consisting of an input pattern and an output pattern.

- **Input Pattern**: \((- (- ?x ?y))\)
- **Output Pattern**: \((- ?y ?x)\)
- **Rewrite Rule**: \(( (- (- ?x ?y)) (- ?y ?x))\)
- **Example Input**: \((- (- (\sin \theta) z))\)
- **Bindings**: \((?y z) (?x (\sin \theta)))\)
- **Output**: \((- z (\sin \theta))\)
Copy Tree and Substitute

It is easy to write a function to copy a binary tree:

(defn copy-tree [form]
  (if (and (seq? form) (not (empty? form)))
    (cons (copy-tree (first form))
      (copy-tree (rest form)))
    form)
)

Why make an exact copy of a tree that we already have? Well, if we modify \texttt{copy-tree} slightly, we can make a copy with a substitution:

; substitute new for old in form
(defn subst [new old form]
  (if (and (seq? form) (not (empty? form)))
    (cons (subst new old (first form))
      (subst new old (rest form)))
    (if (= form old)
      new
      form) ) )
Substitution Examples

>(subst 'axolotl 'banana '(banana pudding))
(axolotl pudding)

>(subst 10 'r '(* pi (* r r)))
(* pi (* 10 10))

>(def pi 3.1415926535897933)

>(eval (subst 10 'r '(* pi (* r r))))
314.15926535897933
Binding Lists

A *binding* is an association between a name and a value.

In Clojure, we can represent a binding as a *list*: 

```
(list name value), e.g. (?X 3). We will use names that begin with ? to denote variables.
```

A set of bindings is represented as a list, called an *association list*, or *alist* for short. A new binding can be added by:

```
(cons (list name value) binding-list)
```

A name can be looked up using *assocl*:

```
(assocl name binding-list)
```

```
(assocl '?y '((?x 3) (?y 4) (?z 5)))
   = (?y 4)
```

The value of the binding can be gotten using *second*:

```
(second (assocl '?y '((?x 3) (?y 4) (?z 5))))
   = 4
```
Multiple Substitutions

The function `(sublis alist form)` makes multiple substitutions simultaneously:

```
>(sublis '((rose peach) (smell taste))
  '(a rose by any other name
    would smell as sweet))
```

(a peach by any other name would taste as sweet)

; substitute in form with bindings in alist

```
(defn sublis [alist form]
  (if (and (seq? form) (not (empty? form)))
    (cons (sublis alist (first form))
      (sublis alist (rest form)))
    (let [binding (assocl form alist)]
      (if binding
        (second binding)
        form))
  )
)```
Instantiating Design Patterns

*sublis* can be used to instantiate design patterns. For example, we can instantiate a tree-recursive accumulator pattern to make various functions:

```
(def pattern
  '(defn ?fun [tree]
      (if (and (seq? tree) (not (empty? tree)))
        (?combine (?fun (first tree))
          (?fun (rest tree)))
        (if (?test tree) ?trueval ?falseval)))))
```

```
>(sublis '(('?fun    nnums  )
            (?combine +     )
            (?test    number?)
            (?trueval 1     )
            (?falseval 0    )) pattern)
```

```
(defn nnums [tree]
  (if (and (seq? tree) (not (empty? tree)))
    (+ (nnums (first tree))
      (nnums (rest tree)))
    (if (number? tree) 1 0)))
```

```
>(nnums '(+ 3 (* i 5)))
2
```
Tree Equality

It often is necessary to test whether two trees are equal, even though they are in different memory locations. We will say two trees are equal if:

- the structures of the trees are the same
- the leaf nodes are equal

\[
(\text{defn equal } \ [x \ y] \\
(\text{if } (\text{and } (\text{list? } x) (\text{not } (\text{empty? } x))) \\
(\text{and } (\text{list? } y) (\text{not } (\text{empty? } y)) \\
(\text{equal } (\text{first } x) (\text{first } y)) \\
(\text{equal } (\text{rest } x) (\text{rest } y))) \\
(= x y) ) )
\]

>\((\text{equal '}(+ a (* b c)) '(+ a (* b c)))\)

true

Some say that two trees are equal if they print the same. Note that this function treats a cons as a binary first-rest tree rather than as a lhs-rhs tree.
Tracing Equal

\texttt{\textgreater{} (equal '(+ a (* b c)) '(+ a (* b c)))}

\begin{enumerate}
\item \texttt{(equal (+ A (* B C)) (+ A (* B C)))}
\item \texttt{(equal + +)}
\item \texttt{(equal A A)}
\item \texttt{(equal (+ B C) (+ B C))}
\item \texttt{(equal * *)}
\item \texttt{(equal B B)}
\item \texttt{(equal C C)}
\item \texttt{(equal NIL NIL)}
\end{enumerate}

This is our old friend, depth-first search, on two trees simultaneously.
Design Pattern: Nested Tree Recursion

Binary tree recursion can be written in a nested form, analogous to tail recursion. We carry the answer along, adding to it as we go. This form is useful if it is easier to combine an item with an answer than to combine two answers. Compare to p. 38.

(defn myfunb [tree answer]
  (if (interior? tree)
    (myfunb (right tree)
      (myfunb (left tree) answer))
    (combine (baseanswer tree) answer)))

(defn myfun [tree] (myfunb tree init) )

; count numbers in a tree
(defn nnumsb [tree answer]
  (if (and (seq? tree) (not (empty? tree)))
    (nnumsb (rest tree)
      (nnumsb (first tree) answer))
    (if (number? tree)
      (+ 1 answer)
      answer) ) )

(defn nnums [tree] (nnumsb tree 0))
Tracing Nested Tree Recursion

\[ \text{nnums '(+ (* x 3) (/ z 7))} \]  
1> (NNUMSB (+ (* X 3) (/ Z 7)) 0)  
2> (NNUMSB + 0)  
<2 (NNUMSB 0)  
2> (NNUMSB ((* X 3) (/ Z 7)) 0)  
3> (NNUMSB (* X 3) 0)  
4> (NNUMSB * 0)  
<4 (NNUMSB 0)  
4> (NNUMSB (X 3) 0)  
5> (NNUMSB X 0)  
<5 (NNUMSB 0)  
5> (NNUMSB (3) 0)  
6> (NNUMSB 3 0)  
<6 (NNUMSB 1)  
6> (NNUMSB NIL 1)  
<6 (NNUMSB 1)  
<5 (NNUMSB 1)  
<4 (NNUMSB 1)  
<3 (NNUMSB 1)  
3> (NNUMSB (/ Z 7)) 1)  
4> (NNUMSB (/ Z 7) 1)  
5> (NNUMSB / 1)  
<5 (NNUMSB 1)  
5> (NNUMSB (Z 7) 1)  
6> (NNUMSB Z 1)  
<6 (NNUMSB 1)  
6> (NNUMSB (7) 1)  
7> (NNUMSB 7 1)  
<7 (NNUMSB 2)  
7> (NNUMSB NIL 2)  
<7 (NNUMSB 2)  
<6 (NNUMSB 2)  
...
Pattern Matching

Pattern matching is the inverse of substitution: it tests to see whether an input is an instance of a pattern, and if so, how it matches.

>`(match '(go ?expletive yourself)
       '(go bleep yourself))

((?expletive bleep) (t t))

> (match '(defn ?fun [tree]
         (if (and (seq? tree) (not (empty? tree)))
            (?combine (?fun (first tree))
                      (?fun (rest tree)))
            (if (?test tree) ?trueval ?falseval))
         '(defn nnnums [tree]
            (if (and (seq? tree) (not (empty? tree)))
                (+ (nnnums (first tree))
                   (nnnums (rest tree)))
                (if (number? tree) 1 0))))

((?falseval 0) (?trueval 1) (?test number?)
 (?combine +) (?fun nnnums) (t t))
Specifications of Match

- Inputs: a pattern, `pat`, and an input, `inp`.
- Constants in the pattern must match the input exactly.
- Structure that is present in the pattern must also be present in the input.
- Variables in the pattern are symbols (strings) that begin with `?`.
- A variable can match anything, but it must do so consistently.
- The result of `match` is a list of bindings: `null` indicates failure, not `null` indicates success.
- The dummy binding `(T T)` is used to allow an empty binding list that is not `null`. 
Match Function

(defn equal [x y]
  (if (and (list? x) (not (empty? x)))
    (and (list? y) (not (empty? y))
      (equal (first x) (first y))
      (equal (rest x) (rest y)))
    (= x y) )
)

(defn matchb [pat inp bindings]
  (if (and (seq? bindings) (not (empty? bindings)))
    (if (and (seq? pat) (not (empty? pat)))
      (and (seq? inp) (not (empty? inp))
        (matchb (rest pat)
          (rest inp)
          (matchb (first pat)
            (first inp) bindings)
        )
      )
    )
  )
)

(defn match [pat inp] (matchb pat inp '((t t))))
Transformation by Patterns

Matching and substitution can be combined to transform an input from a pattern-pair: a list of an input pattern and an output pattern.

(defn transform [pattern-pair input]
    (let [bindings (match (first pattern-pair) input)]
        (if bindings
            (sublis bindings (second pattern-pair))
        ))

>(transform '((I aint got no ?x)
        (I do not have any ?x))
 'I aint got no bananas))

(I do not have any bananas)
Many kinds of problems can be solved by search, which involves finding a goal from a starting state by applying operators that lead to a new state.

Depth-first search (DFS) follows an implicit tree of size $O(b^{depth})$, where $b$ is the branching factor. (number of children of a node). Given a state, we test whether it is a goal or a terminal failure node; if not, we generate successor states and try searching from each of them. Many of these searches may fail, and we will backtrack and try a different branch.
Solving Equations

Simple equations can be solved by search, using rules of algebra as operators to transform equations into equivalent forms until an equation for the desired variable is produced.

We will think of the same data structure in several ways:

- **Equation**: \( y = m \cdot x + b \)
- **List structure**: \((= Y (+ (* M X) B))\)
  or \(( \text{op } \text{lhs } \text{rhs} )\) recursively
- **Tree**: 

  \[
  \begin{align*}
  &\quad \text{op} \\
  \text{lhs} &\quad \text{rhs} \\
  \quad \text{rhs} &\quad \text{lhs} \\
  &\quad \text{op} \\
  \text{lhs} &\quad \text{rhs} \\
  \quad \text{rhs} &\quad \text{lhs} \\
  &\quad \text{op} \\
  \text{lhs} &\quad \text{rhs} \\
  \quad \text{rhs} &\quad \text{lhs} \\
  &\quad \text{op} \\
  \text{lhs} &\quad \text{rhs} \\
  \end{align*}
  \]

- **Node of a Tree**: an equation is a node in a search tree whose nodes are equivalent equations.
- **Executable code**: `eval` can evaluate an expression using a set of variable bindings.
Solving an Equation by Search

We can perform algebraic operations by manipulating the list structure representation of an expression tree (taking apart the original tree and constructing a new tree). To solve an equation $e$ for a desired variable $v$:

- **Base cases:**
  - If the $\text{lhs}$ of $e$ is $v$, return $e$.
  - If the $\text{rhs}$ of $e$ is $v$, rewrite $e$ to switch the $\text{lhs}$ and $\text{rhs}$ of $e$, and return that.
  - If only an undesired variable or constant is on the right, ($\text{rhs}$ is not a cons), fail by returning $\text{nil}$.

- **Recursive case:** Rewrite $e$ using an algebraic law, and try to solve that equation. Return the first result that is not $\text{nil}$.

Often, there are two possible ways to rewrite an equation; it is necessary to try both. Thus, the process will be a binary tree search.

We are rewriting an equation in every possible legal way; most of these will not be what we want, but one may work. If we find one that works, we return it.
Examples: Base Cases

>(solve '(= x 3) 'x) ; lhs is desired var
(= x 3)

>(solve '(= 3 x) 'x) ; rhs is desired var
(= x 3)

>(solve '(= 3 y) 'x) ; rhs not cons: fail.
nil
Recursive Cases: Operators

The recursive case has a rhs that is an operation:

\[(= \alpha (\ op \ \beta \ \gamma ))\]

We are hoping that the desired variable will be somewhere in \(\beta\) or \(\gamma\); to get to it, we must apply some kind of inverse operation to both sides of the equation to get rid of \(op\) and isolate \(\beta\) or \(\gamma\).

In general, there may be two inverse operations to try.

We can produce the result of the inverse operation by constructing a new equation from the given one, e.g., given:

\[(= \alpha (\ + \ \beta \ \gamma ))\]

we can construct two new possibilities:

\[(= (-\ \alpha \ \beta \ ) \ \gamma )\] (subtract \(\beta\) from both sides)

\[(= (-\ \alpha \ \gamma \ ) \ \beta )\] (subtract \(\gamma\) from both sides)

After making a new equation, we simply call \texttt{solve} to try to solve \textit{that} equation. We return the first solution that is not \texttt{nil}. 

Recursive Tree Search

In effect, the search process will rewrite the original equation in every possible legal way. Most of these will not be what we want, and will fail, but one of them will be solved for the desired variable.

```lisp
>(solve '(= y (+ x b)) 'x)

1> (SOLVE (= Y (+ X B)) X)
  2> (SOLVE (= (- Y X) B) X)
     <2 (SOLVE NIL)
  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))
```
**Big O and Termination**

We want to make sure that we cannot get into a loop by transforming an equation endlessly.

**Well-founded Ordering:** If a program has an input that is finite and gets smaller in each recursion, and the program stops when the input reaches a lower boundary, then the program is guaranteed to terminate.

Our program assumes that initially the $\text{lhs}$ is only a single variable. Each recursive step makes the $\text{rhs}$ smaller.

We don’t have to worry about Big O for this problem because the number of operations is limited by the size of the expression tree, which is always small.
Solving a Physics Problem

With the collection of programs that we now have, solving a simple physics problem becomes easy. We assume that we have values for some variables and want the value of another variable:

What is the radius of a circle with area = 10

- Make a list (set) of the variables in the problem (desired variable and variables whose values are given).
  (radius area)
- Find an equation that involves those variables.
  (= area (* 3.14159 (expt radius 2)))
- Solve the equation for the desired variable.
  (= radius (sqrt (/ area 3.14159)))
- evaluate the rhs of the equation for the given values.
  1.78412
Solving Sets of Equations

Given:

• a set of equations

fall:

((= gravity '(q 9.80665 (/ m (* s s)))))
(= horizontal-velocity '(q 0 (/ m s))) ; default
(= height (* 1/2 (* gravity (expt time 2)))))
(= velocity (* gravity time)) ; vertical
(= kinetic-energy
   (* 1/2 (* mass (expt total-velocity 2)))))
(= horizontal-distance (* horizontal-velocity time))

(= total-velocity
   (sqrt (+ (expt velocity 2)
   (expt horizontal-velocity 2))))

• a set of variables with known values:
  ((TIME 4))

• a variable whose value is desired: HEIGHT
Solving a Set of Equations by Search

• Try to find an equation where all variables are known except one.

\[ (= \text{F} (* \text{M} \text{A})) \]

• Solve the equation for that variable.

\[ (= \text{A} (/ \text{F} \text{M})) \]

• Substitute the known values into the right-hand side of the solved equation (function \text{sublis}).

\[ (= \text{A} (/ 8 2)) \]

• evaluate the resulting expression to give the value of the new variable. Add that variable to the binding list.

\[ (= \text{A} 4) \]

• Keep trying until you get the value of the variable you want (or quit if you stop making any progress).
Solving Physics Story Problems

By combining the techniques we have discussed with a simple English parser, a remarkably small Lisp program can solve physics problems stated in English:

```
>(phys '(what is the area of a circle with radius = 2))
12.566370614359172

>(phys '(what is the circumference of a circle with area = 12))
12.279920495357862

>(phys '(what is the power of a lift with mass = 5 and height = 10 and time = 4))
122.583125
```
Solving Equations with Patterns

Solving equations by writing code to construct new equations is somewhat difficult. However, doing it with patterns is easy.

All we need is a list of transformations, from given equation to new equation:

( ( (= ?x (+ ?y ?z)) (= (- ?x ?y) ?z) )
  ( (= ?x (+ ?y ?z)) (= (- ?x ?z) ?y) )
  ... )

To solve an equation, the base cases will be the same as before. If the rhs is a list, simply try every pattern, in a loop, until either one pattern works (success) or the end of the list of patterns is reached (failure).
Generating Code from Equations

The first programming language, FORTRAN, is an abbreviation for FORMula TRANslation. There is a long history of similarity between equations and programs.

Suppose that we have a set of physics equations, and we want a program to calculate a desired variable from given variable values.

Each time an equation is solved for a new variable,

- add the new variable to a list of variables
- \texttt{cons} the solved equation onto a list of code

If the variable that is solved for was the desired variable, \texttt{cons} a return statment onto the code.

At the end, reverse the code (\texttt{trrev}).
Transformation Patterns

Optimization:

(defpatterns 'opt
  '( (+ ?x 0) ?x)
    (* ?x 0) 0)
    (* ?x 1) ?x)
    (setq ?x (+ ?x 1)) (incf ?x)) ))

Language translation:

(defpatterns 'lisptojava
  '( (aref ?x ?y) ("" ?x "[" ?y "]")
    (incf ?x) ("++" ?x))
    (setq ?x ?y) ("" ?x " = " ?y))
    (+ ?x ?y) ("(" ?x " + " ?y ")))
    (= ?x ?y) ("(" ?x " == " ?y ")))
    (and ?x ?y) ("(" ?x " && " ?y ")))
    (if ?c ?s1 ?s2) ("if (" ?c ")" #\Tab
    #\Return ?s1
    #\Return ?s2)))
Program Transformation using Lisp

>code
(IF (AND (= J 7) (/= K 3))
  (PROGN (SETQ X (+ (AREF A I) 3))
    (SETQ I (+ I 1))))

>(cpr (trans (trans code 'opt)
  'lisptojava))

if (((j == 7) && (k != 3)))
{
  x = (a[i] + 3);
  ++i;
}
Programs and Trees

• Fundamentally, programs are trees, sometimes called *abstract syntax trees* or *AST*.

• *Parsing* converts programs in the form of character strings (source code) into trees.

• It is easy to convert trees back into source code form (*unparsing*).

• Parsing - Transformation - Unparsing allows us to transform programs.
Symbolic Differentiation

•

•

•
Algebraic Simplification

•

•

•
Max and Min of a Function

•

•

•
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\(^6\)
- *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.\(^7\)


\(^7\)Diagram by John Sowa, from “The Challenge of Knowledge Soup,” 2005.
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).  

**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*.

---

8Miller, 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.
Logic

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 forms of logic most commonly used are *propositional calculus* and *first-order predicate calculus* (*FOPC*).
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   | $\land$          | $\&\&$ | and           |
  | Disjunction   | $\lor$          | $||$ | or            |
  | Implication   | $\rightarrow$    | $\supset$ | implies, if-then |
  |               | $\iff$            | $\leftrightarrow$ | iff (if and only if) |

- Constants: *True* or filled-in box $\Box$ True
  
  $\Box$ ("box") False
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: $P \lor \neg P$. Otherwise, it is *invalid*: $P$.
- 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*: $P \land \neg P$.

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:
  \[ F \rightarrow G = \neg F \lor G \]
  \[ F \leftrightarrow G = (F \rightarrow G) \land (G \rightarrow F) \]
  \[ A \land B \land C \rightarrow D = \neg A \lor \neg B \lor \neg C \lor D \]

• De Morgan’s Laws:
  \[ \neg(F \lor G) = \neg F \land \neg G \]
  \[ \neg(F \land G) = \neg F \lor \neg G \]

• Distributive:
  \[ F \lor (G \land H) = (F \lor G) \land (F \lor H) \]
  \[ F \land (G \lor H) = (F \land G) \lor (F \land H) \]

Inference Rules

• Modus Ponens: \[ \frac{P, P \rightarrow Q}{Q} \]
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*.

- **Satisfiability**: Find an assignment of truth values to variables that will make a propositional calculus formula true. There are efficient *SAT solvers* that can solve systems with millions of propositional variables.

- **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:

\[ A \]
\[ B \]
\[ D \]
\[ A \land B \rightarrow C \quad \text{or} \quad C \leftarrow A \land B \]
\[ C \land D \rightarrow E \quad \text{or} \quad E \leftarrow C \land D \]

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.

---

9Backward chaining only works for Horn clauses, which have at most one positive literal.
Backchaining\(^{10}\)

Backchaining is easily implemented as a recursive tree search.

The function \((\text{backchain } \text{goal rules facts})\) tries to prove a goal given a set of rules and facts. \text{goal} is a symbol (atom or propositional variable). \text{facts} is a list of atoms that are known to be true. \text{rules} is a list of rules of the form \((\text{conclusion prem}_1...\text{prem}_n)\); each rule states that the conclusion is true if all of the premises are true. For example, the rule \(A \land B \rightarrow C\) would be written \(C \leftarrow A \land B\) or \((c \ a \ b)\), similar to Prolog, \(c :\ a, b\).

\(^{10}\)file backch.clj
**Backchaining Code**

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

```
(defn alltrue [premises rules facts]) ; signature

(defn backchain [goal rules facts]
  (or (member goal facts)
      (some (fn [rule]
              (and (= (first rule) goal)
                   (alltrue (rest rule) rules facts))
              rules)) )

(defn alltrue [premises rules facts]
  (or (empty? premises)
       (and (backchain (first premises) rules facts)
            (alltrue (rest premises) rules facts)) )

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

---

11 file `backch.clj`
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 $F = F_1 \land F_2 \land \ldots \land F_n$ where each $F_i$ is a disjunction of literals. Example: $(\neg P \lor Q) \land (P) \land (\neg Q)$

CNF is used for resolution and for SAT solvers. There is also a *disjunctive normal form*. 
Satisfiability Checking

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

********** edit this **********

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 implementation 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. 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 \textit{bound}; otherwise, it is \textit{free}. 
Order of Quantifiers

The order in which quantifiers appear is very important; it must be maintained.

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:
   \[
   \forall x [\text{Man}(x) \rightarrow \exists y [\text{Woman}(y) \land \text{Loves}(x, y)]]
   \]
   and Skolemized:
   \[
   \text{Man}(x) \rightarrow [\text{Woman}(\text{lover}(x)) \land \text{Loves}(x, \text{lover}(x))]
   \]

2. There is some woman (perhaps Marilyn Monroe) who is loved by every man. This would be written:
   \[
   \exists y \forall x [\text{Man}(x) \rightarrow [\text{Woman}(y) \land \text{Loves}(x, y)]]
   \]
   and Skolemized:
   \[
   \text{Man}(x) \rightarrow [\text{Woman}(a) \land \text{Loves}(x, a)]
   \]
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.
Unification

If a variable is universally quantified, we are justified in substituting any term for that variable.

If we want to do backchaining with predicate calculus, we need to find a set of substitutions of terms for variables that will make the conclusion of a formula match what we are trying to prove. The process that does this is called unification.

If we have a formula $C \leftarrow A \land B$, we can think of this as being analogous to a subroutine: subroutine $C$ consists of calls to subroutines $A$ and $B$.

We can think of unification as analogous to binding the formal arguments of a subroutine to the actual arguments with which it is called. Unification is more general than subroutine call: whereas subroutine call is top-down, unification can send arguments in both directions.
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 ((varterm) ...).

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)))
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 $\square$ is derived by resolution, then the original set of clauses is unsatisfiable.
- **complete**: If a set of clauses is unsatisfiable, resolution will eventually derive $\square$. 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.
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:

\[
\text{Person}(x) \rightarrow \text{Male}(x) \lor \text{Female}(x)
\]

\[
\neg \text{Person}(x) \lor \text{Male}(x) \lor \text{Female}(x)
\]

which has two positive literals. Such clauses do not work with 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. If the conclusion is true for all possible assumptions, it is true.
Backchaining Theorem Prover

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

\(\text{>(goal '}(\text{father} \ X \ (\text{HARMONIA})))\)
1> \((\text{GOAL} \ (\text{FATHER} \ X \ (\text{HARMONIA})))\)
((X ARES))

\(\text{>(goal '}(\text{parent} \ Z \ (\text{HARMONIA})))\)
1> \((\text{GOAL} \ (\text{PARENT} \ Z \ (\text{HARMONIA})))\)
2> \((\text{GOAL} \ (\text{FATHER} \ Z \ (\text{HARMONIA})))\)
<2 \((\text{GOAL} \ ((Z \ ARES)))\)
((Z ARES))

\(\text{>(goal '}(\text{grandparent} \ X \ (\text{HARMONIA})))\)
1> \((\text{GOAL} \ (\text{GRANDPARENT} \ X \ (\text{HARMONIA})))\)
2> \((\text{GOAL} \ (\text{PARENT} \ Z \ (\text{HARMONIA})))\)
3> \((\text{GOAL} \ (\text{FATHER} \ Z \ (\text{HARMONIA})))\)
<3 \((\text{GOAL} \ ((Z \ ARES)))\)
<2 \((\text{GOAL} \ ((Z \ ARES)))\)
2> \((\text{GOAL} \ (\text{PARENT} \ X \ (\text{ARES})))\)
3> \((\text{GOAL} \ (\text{FATHER} \ X \ (\text{ARES})))\)
<3 \((\text{GOAL} \ ((X \ ZEUS)))\)
<2 \((\text{GOAL} \ ((X \ ZEUS)))\)
((X ZEUS))

99
(setf (get 'father 'ground) '(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))
Knowledge Rep. in Predicate Calculus

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

\[
\begin{align*}
(DOG & \ DOG1) \\
(NAME & \ DOG1 \ FIDO) \\
(HOUND & \ DOG1) \\
(LOVES & \ JOHN \ MARY)
\end{align*}
\]

Facts can be retrieved in response to *patterns:*

\[
\begin{align*}
(LOVES & \ JOHN \ MARY) & \text{Does John love Mary?} \\
(LOVES & \ JOHN \ ?X) & \text{Whom does John love?} \\
(LOVES & \ ?X \ MARY) & \text{Who loves Mary?} \\
(LOVES & \ ?X \ ?Y) & \text{All pairs of lovers.}
\end{align*}
\]

**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:

\[ \forall x P_1(x) \land \ldots \land P_n(x) \rightarrow C(x) \]

However, the interpretation of rules may or may not correspond to a formal logical interpretation.
**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 $P_1...P_n$.

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

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:
   $$\forall x \forall y \forall z GREATER(x, y) \land GREATER(y, z) \rightarrow GREATER(x, z)$$

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:

\[ \forall x \forall y WIFE(x, y) \rightarrow LOVES(x, y) \]

\[ WIFE(John, Mary) \]
\[ WIFE(Bill, Jane) \]
\[ \ldots \]

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).
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 \textit{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”.

105
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.
Expert Systems\textsuperscript{12}

*Expert systems* attempt to capture the knowledge of a human expert and make it available through a computer system.\textsuperscript{13}

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

\textsuperscript{12}These slides jointly authored with Bruce Porter.
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”\(^\text{14}\)).

- 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).

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.\(^\text{15}\)

- 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.

\(^{15}\text{Parker, T., Rules of Thumb, Boston, MA: Houghton Mifflin Publishers, 1983.}\)
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:

\[
\text{if} \ conditions \ \text{then} \ conclusion
\]

Example: MYCIN\textsuperscript{16}

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 (0.6) that the identity of the organism is Bacteroides.

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\textsuperscript{17}

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 \( g \in \text{Goal-list} \) do
If \( \text{prove}(g, \text{DB}, \text{Rules}) \) then Print (“Diagnosis:”, \( g \))

Function \( \text{prove} \) (goal, DB, Rules)
If \( \text{goal} \in \text{DB} \) then return True
elseif \( \exists r \in \text{Rules} \) such that \( r_{RHS} \) contains goal
then return \( \text{provelist}(LHS, \text{DB}, \text{Rules}) \textsuperscript{18} \)
else Ask user about goal and return answer

\textsuperscript{18}provelist calls prove with each condition of LHS
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: \( P(\text{rain}|\text{cloudy}) \)

If \( A \) and \( B \) are independent, then \( P(A|B) = P(A) \) and \( P(AB) = P(A) \times P(B) \).

Expert systems usually deal with events that are not independent, e.g. a disease and its symptoms are not independent.

Bayes’ Theorem

\[ P(AB) = P(A|B) \times P(B) = P(B|A) \times P(A) \]

therefore \( P(A|B) = P(B|A) \times P(A) / P(B) \)
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.

\[ P(A|B) = P(B|A) \times P(A) / P(B) \]

Suppose: \( A = \) Patient has measles, \( B = \) has a rash
Then: \( P(\text{measles/rash}) = \)
\[ P(\text{rash/measles}) \times P(\text{measles}) / P(\text{rash}) \]

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.\(^{19}\)

<table>
<thead>
<tr>
<th></th>
<th>Toothache</th>
<th>¬ Toothache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cavity</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>¬ Cavity</td>
<td>0.01</td>
<td>0.89</td>
</tr>
</tbody>
</table>

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.

\(^{19}\)Example from Russell & Norvig.
Chain Rule

We can compute probabilities using a chain rule as follows:

\[ P(A \land B \land C) = P(A|B \land C) \times P(B|C) \times P(C) \]

If some conditions \( C_1 \land \ldots \land C_n \) are independent of other conditions \( U \), we will have:

\[ P(A|C_1 \land \ldots \land C_n \land U) = P(A|C_1 \land \ldots \land C_n) \]

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.\textsuperscript{20}

\textsuperscript{20}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”.

\[
\text{MB} \quad 0 - 1 \quad \text{Measure of Belief} \\
\text{MD} \quad 0 - 1 \quad \text{Measure of Disbelief} \\
\text{CF} = \text{MB} - \text{MD} \quad -1 \text{ to } 1 \quad \text{Certainty Factor} \\
\quad -1 \quad \text{Definitely False} \\
\quad 0 \quad \text{No information, or cancellation} \\
\quad +1 \quad \text{Definitely True}
\]

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 $\text{AND}$ function:

\[
(\text{AND} (\text{SAME CNTXT GRAM GRAMNEG})
\text{AND} (\text{SAME CNTXT MORPH COCCUS}))
\]

$\text{AND}$ operates as follows:

1. If any clause is false ($\text{nil}$) or has $CF \leq .2$, $\text{AND}$ returns false ($\text{nil}$). 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 $CF > .2$, $\text{AND}$ returns the minimum of the clause $CF$ values.

There is also a function $\text{OR}$ that returns the maximum of its argument $CF$ values.
Rule Certainty Factors

Premise:

\[
\text{($AND (SAME CNTXT SITE BLOOD)}
\text{ (SAME CNTXT GRAM GRAMNEG)}
\text{ (SAME CNTXT MORPH ROD)}
\text{ (SAME CNTXT BURNED))}
\]

Action:

\[
\text{(CONCLUDE CNTXT}
\text{ IDENTITY PSEUDOMONAS-AERUGINOSA}
\text{ TALLY 400)}
\]

The result of the rule as a whole is calculated as follows:

1. \text{$AND$} sets the global variable \text{TALLY} to the minimum CF of its components.
2. The Rule CF is \text{TALLY} times the CF specified in the \text{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:

$$cf_{combine}(CF_p, CF_n) = \begin{cases} 
  CF_p + CF_n \times (1 - CF_p) & CF_p > 0, CF_n > 0 \\
  (CF_p + CF_n) & \text{signs differ} \\
  /(1 - \min(|CF_p|, |CF_n|)) & \end{cases}$$

$$-cf_{combine}(-CF_p, -CF_n) \quad CF_p < 0, CF_n < 0$$

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
- $\texttt{AND}$ takes minimum CF in premise
- conclude $CF = CF_{\text{premise}} \times CF_{\text{rule}}$
- 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 $0.5 + 0.6 \times (1 - 0.5) = 0.8$. This could also be computed as $0.6 + 0.5 \times (1 - 0.6) = 0.8$. 
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\textsuperscript{21}

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.

Increasing Certainty

EMYCIN systems sometimes include rules of the form:
\[ A \land B \rightarrow A \]

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 (0.8) 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 $CF > .2$. 
Explanation

A rule-based expert system such as EMYCIN makes it easy to provide explanations that make sense.

A *why* question can be answered by evaluating all rules that match the conclusion being questioned; those whose premise evaluates to true ($> 0.2$) can be reported.

Likewise, *why not* questions can be answered by examining rules that match the conclusion and reporting the first premise clause that evaluates to false ($< 0.2$).
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.
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

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: +

---

Final Decision Tree with Classifications

One-level Decision Tree

Two-level Decision Tree
Algorithm for Rule Induction

Instances: a set of training instances
Features: feature vector \((f_1, f_2, \ldots, f_n)\) input from teacher
Domains: set of domains for Features \(\{d_1, d_2, \ldots, d_m\}\)
Classes: set of classes \(\{c_1, c_2, \ldots, c_k\}\)

(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:

```
   f
  /   \
 d_1   d_2
  \    
   \  
    d_m

root labeled \(f\)
arcs labeled with values from \(d\)
each child is a subtree created by recursive call:
```
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:

\[ out_{i+1} = \alpha \times in_i + (1 - \alpha) \times out_i, \text{ where } \alpha << 1. \]

This filter reduces “noise” in the input, passing through the long-term trend, but with a time delay. There is memory of past data, but with exponential forgetting of old data. A filter like this was used to adjust weights of heuristic feature detectors in Samuel’s checker-player program. Multiple parameter values can be learned simultaneously.

\[ \text{23The filter lets low frequencies pass through, while blocking high frequencies.} \]
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.
Example of Orthogonal Knowledge Sources

Look for orthogonal knowledge sources in your domain. A Jigsaw Puzzle is a good example,

Generate-and-Test Solution: Select a piece; try to fit it to every other piece until you succeed. Repeat until done. $O(n^2)$ comparisons required.

Orthogonal Knowledge Sources:

1. Pieces with one straight edge must go on the edge.
2. Pieces with two straight edges are corners.
3. Color must be continuous across piece boundaries.
4. A piece of a given color is likely to go in a picture area of that color. (E.g., a light blue piece is likely to be part of the sky area.)

Each of these knowledge sources is orthogonal to the others (independent); each reduces search by reducing the number of edge pattern comparisons required.
The Tuning Fallacy

The Tuning Fallacy operates as follows:

• A breadboard system is trained on one set of data and achieves 90% accuracy on that data. This is taken as a “proof of concept”.

• However, the trained system only gets 70% accuracy on a new data set.

• If trained on the new data set, the system reaches 90% on that data, but only gets 70% on the original data set: the training set phenomenon.

• The goal becomes to find a setting of the weights that will recognize all the exemplars simultaneously.

The Fallacy: There is no such setting. But one can spend years and millions of dollars looking for one.

The problem is the false alarm rate as the number of exemplars grows.

The ROC Curve\textsuperscript{24} is often used to show true positive rate recall plotted against false positive rate.

\textsuperscript{24}Receiver Operating Characteristic
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:²⁵

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.

²⁵METAL 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, \( \text{length} \propto -\log(\text{frequency}) \).

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

- facsimile transmission \( \rightarrow \) fax
- \( \text{latissimus dorsae} \) \( \rightarrow \) lat
- Mediterranean \( \rightarrow \) Med
- robot \( \rightarrow \) bot
Areas of Natural Language

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

- **Syntax:** The rules by which words can be put together to form legal sentences.

- **Semantics:** Study of the ways statements in the language denote *meanings*.

- **Pragmatics:** Knowledge about the world and the social context of language use.

Q: Do you know the time?
A: Yes.
Computer Language Understanding

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

There are many problems in making such a translation:

- **Ambiguity:** There may be multiple ways of translating a statement.
  
  - **Lexical Ambiguity:** most words have multiple meanings.
    
    The pitcher broke his arm.
    
    The pitcher broke.
  
  - **Grammatical Ambiguity:** Different ways of parsing (assigning structure to) a given sentence.
    
    One morning I shot an elephant in my pajamas. 
    How he got in my pajamas I’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:** 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.

\(^{26}\)The singular is *anaphor.*
Morphology

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

For English, a fairly simple suffix-stripping algorithm plus a small list of irregular forms will suffice.\(^\text{27}\)

\[
\text{running} \quad \rightarrow \quad \text{run} + \text{ing}
\]

\[
\text{went} \quad \rightarrow \quad \text{go} + \text{ed}
\]

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.

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.28

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

---

28 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.\(^{29}\)

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:\(^{30}\)

1. Every enter now severally so, let
2. What means, sir. I confess she? then all sorts, he is trim, captain.
4. They say all lovers swear more performance than they are wont to keep obliged faith unforfeited!

---

\(^{29}\) corpus (Latin for body) is singular, corpora is plural. A corpus is a collection of natural language text, sometimes analyzed and annotated by humans.

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.\textsuperscript{31}

A \textit{Hidden Markov Model} (HMM) tagger chooses the tag for each word that maximizes: \textsuperscript{32}

\[ P(\text{word} \mid \text{tag}) \ast P(\text{tag} \mid \text{previous n tags}) \]

For a bigram tagger, this is approximated as:

\[ t_i = \underset{j}{\text{argmax}} P(w_i \mid t_j)P(t_j \mid t_{i-1}) \]

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%.

\textsuperscript{31} from Jabberwocky, by Lewis Carroll.
\textsuperscript{32} Jurafsky, \textit{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.33

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  I saw the man on the hill with the telescope.

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

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

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

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

  *gloria in excelsis Deo*

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

  *la casa blanca*

  *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:

\[ V^* \] 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

\[ \epsilon \] 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.34

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 $V$.

---

Recognizing Automaton

\[ X = A + B * C \]

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

The recognizer answers *Yes* or *No* to the question “Is the input string a member of the language?”
Regular Languages

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

• Only one nonterminal can appear in any derived string, and it must appear at the right end.

• Equivalent to a deterministic finite automaton (simple program).

• Parser never has to back up or do search.

• Linear parsing time.

• Used for simplest items (identifiers, numbers, word forms).

• Any finite language is regular.

• Any language that can be recognized using finite memory is regular.
Context Free Languages

Productions: $A \rightarrow \alpha$

$A \in N$

$\alpha \in V^*$

• 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: $A + B \ast (C + D)$.

  – A noun phrase can contain a prepositional phrase, which contains a noun phrase:
    
    a girl with a hat on her head.
What Kind of Language is English?

- English is Context Free.\textsuperscript{35}
- English is not Context Free.\textsuperscript{36}
- 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:

\textit{What is a good way to describe English for computer processing?}

\textsuperscript{35}Gazdar, G., “NLs, CFLs, and CF-PSGs”, in Sparck Jones, K. and Wilks, Y., Eds., \textit{Automatic Natural Language Processing}, Ellis Horwood Ltd., West Sussex, England, 1983.

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. 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.

---

37 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>. 
Augmented Transition Networks

An ATN \(^{38}\) 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.

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:

\[
\text{Lexicon} \\
/ \ \\
/ \ \\
/ \ \\
\text{Sentence} \rightarrow \text{Syntax} \rightarrow \text{Semantics} \\
| \\
| \\
| \\
\text{Pragmatics} \\
| \\
| \\
\text{Output}
\]

Unfortunately, such a clean separation doesn’t work well in practice.
Problems with Separability

• Lexicon:
  – New uses of words.
    
    *You can verb anything.* – William Safire
  – Metaphor: The computer is down.

• Syntax:
  – Ambiguity: *hundreds* of syntactically possible interpretations of ordinary sentences.
  – Agreement:
    
    Bill and John love Mary.
  – Elision: omission of parts of a sentence.
    
    He gave John fruit and Mary candy.

• Discourse:
  – The meaning of a sentence depends on context.
Combining Syntax and Semantics

There are several advantages to combining syntactic and semantic processing:

• **Removal of Ambiguity:** It is better to eliminate an incorrect parsing before it is generated, rather than generating all possible interpretations and then removing bad ones.
  – Computer time is saved.
  – Eliminating one bad partial interpretation eliminates many bad total interpretations.

• **Reference:** It is often advantageous to relate the sentence being parsed to the model that is being constructed during the parsing process. “John holds the pole at one end [of the pole].”

• **Psychological Plausibility:** People can deal with partial and even ungrammatical language.

  All your base are belong to us.

  This sentence no verb. – D. Hofstadter
How to Combine Syntax & Semantics

• **Grammar and Parser:** no place to include program operations.
  Note that in natural language processing we often want the parsing that is chosen for ambiguous sentences to depend on semantics.

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

• **Augmented Transition Network:** best of both worlds.
Natural Language as an AI Problem

Natural language understanding is a classical AI Problem:

- **Minimal Input Data:** the natural language statement does not contain the message, but is a minimal specification to allow an intelligent reader to construct the message.

- **Knowledge Based:** the interpretation of the message is based in large part on the knowledge that the reader already has.

- **Reference to Context:** the message implicitly refers to a context, including what has been said previously.

- **Local Ambiguity:** many wrong interpretations are superficially consistent with the input.

- **Global Constraints:** there are many different kinds of constraints on interpretation of the input.

- **Capturing the Infinite:** a language understanding system must capture, in finite form, rules sufficient to understand a potentially infinite set of statements.
Reference

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

A pole supports a weight at one end.

Determiners:

• Indefinite: a
  
  Make a new object.

• Definite: the, one, etc.
  
  Find an existing object;
  else, find something closely related to an existing object;
  else, make a new one.

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

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

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

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

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

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

the *100 lb* boy

the *heavy* end
English

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

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

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

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

(defun op (x) (first x))
(defun lhs (x) (second x))
(defun rhs (x) (third x))

(defun op->english (op)
  (list 'the
    (second (assoc op '((+ sum)
      (- difference)
      (* product)
      (/ quotient)
      (sin sine)
      (cos cosine))))))
  'of))

(defun exp->english (x)
  (if (consp x) ; operator?
    (append
      (op->english (op x))
      (exp->english (lhs x))
      (if (null (cddr x)) ; unary?
        ()
        (cons 'and
          (exp->english (rhs x)) ) )
    )
    (list x)) ) ; leaf: operand

39file expenglish.lsp
Generating English

```
{lisp}
(load "~/projects/cs375/expenglish.lsp")

(exp->english 'x)

(X)

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

(THE SUM OF X AND Y)

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

(THE QUOTIENT OF THE COSINE OF Z AND THE SUM OF X AND THE SINE OF Y)
```
Simple Language Processing: ELIZA

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

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

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

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

Problems:

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

  Patient: I just feel like jumping off the roof.
  Doctor: Tell me more about the roof.
Spectrum of Language Descriptions

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

- **ELIZA:**
  Too restricted. A large application, PARRY – an artificial paranoid – was attempted, but failed to get good enough coverage even with 10,000 patterns.

- **General English Grammar:**
  Too ambiguous. Hundreds of interpretations of ordinary sentences.

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

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

\[
\text{WHAT} \text{ CUSTOMERS IN AUSTIN ORDERED WIDGET BRACKETS IN MARCH}
\]

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 fastest French sub in the Med?"
Example Semantics for a Semantic Grammar

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYP</th>
<th>OWN</th>
<th>LAT</th>
<th>LONG</th>
<th>SPD</th>
<th>LNG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kitty Hawk</td>
<td>CV</td>
<td>US</td>
<td>10°00′N</td>
<td>50°27′E</td>
<td>35</td>
<td>1200</td>
</tr>
<tr>
<td>Eclair</td>
<td>SS</td>
<td>France</td>
<td>20°00′N</td>
<td>05°30′E</td>
<td>15</td>
<td>50</td>
</tr>
</tbody>
</table>

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

This query is parsed by the top-level production

\[
<S> \rightarrow \text{What is } <\text{ship-property}> \text{ of } <\text{ship}>? 
\]

which is conveniently structured in terms of:

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

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.
Parsing English

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

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

Backup means that parsing is a search process, i.e. likely to be NP-complete. However, since English sentences are usually short, this is not a problem in practice.

An Augmented Transition Network (ATN) framework facilitates parsing of English.
A global variable *sent* points to a list of words that is the remaining input sentence:
(GOOD CHINESE RESTAURANT IN LOS ALTOS)

A global variable *word* points to the current word:
GOOD

(cat category) tests whether a word is in the specified category. It can also translate the word, e.g. (cat 'month) might return 3 if *word* is MARCH.

(next) moves to the next word in the input

(saveptr) saves the current sentence position on a stack.

(success) pops a saved position off the stack and returns T.

(fail) restores a saved position from the stack (restoring *sent* and *word*) and returns NIL.
Parsing Functions

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

(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)))))
Grammar Compiler

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

```
(rulecom '(LOC  -> (in (city))
               (restrict 'customer-city $2)))

(DEFUN LOC62 ()
  (LET ($1 $2)
    (SAVEPTR)
    (IF (AND (AND (EQL (SETQ $1 *WORD*) 'IN)
                  (NEXT))
           (SETQ $2 (CITY)))
     (PROGN (SUCCESS)
            (RESTRICT 'CUSTOMER-CITY $2))
           (FAIL))))
```

41 file gramcom.lsp
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*))
Restaurant Database Grammar

(defgrammar
  (s -> ((command) (a/an)? (qual)? (resttype)?
    (restword) (qualb)? (loc)?)
  (makequery (combine (retrieve 'restaurant)
    (retrieve 'streetno)
    (retrieve 'street)
    (retrieve 'rating)
    $3 $4 $6 $7)))

  (s -> (where can (i/you) (get) (qual)?
    (resttype)? food ? (loc)?)
  (makequery (combine (retrieve 'restaurant)
    (retrieve 'streetno)
    (retrieve 'street)
    (retrieve 'rating)
    $5 $6 $8)))

  (command -> (what is) t)
  (qual -> ((quality))
    (restrictb '>= 'rating $1))
  (qualb -> (rated above (number))
    (restrictb '>= 'rating $3))
  (resttype -> ((kindfood))
    (restrict 'foodtype $1))
  (loc -> (in (city)) (restrict 'city $2))
Restaurant Queries

%lisp
>(load "~/projects/cs375/restaurant.lsp")

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

((2001-FLAVORS-ICE-CREAM-&-YOGUR 2485 TELEGRAPH-AVE)
 (BASKIN-ROBBINS 1471 SHATTUCK-AVE)
 (DOUBLE-RAINBOW 2236 SHATTUCK-AVE)
 (FOSTERS-FREEZE 1199 UNIVERSITY-AVE)
 (MARBLE-TWENTY-ONE-ICE-CREAM 2270 SHATTUCK-AVE)
 (SACRAMENTO-ICE-CREAM-SHOP 2448 SACRAMENTO-ST)
 (THE-LATEST-SCOOP 1017 ASHBY-AVE))

>(askr '(show me chinese restaurants rated above 2.5 in los altos))

((CHINA-VALLEY 355 STATE-ST)
 (GRAND-CHINA-RESTAURANT 5100 EL-CAMINO-REAL)
 (HUNAN-HOMES-RESTAURANT 4880 EL-CAMINO-REAL)
 (LUCKY-CHINESE-RESTAURANT 140 STATE-ST)
 (MANDARIN-CLASSIC 397 MAIN-ST)
 (ROYAL-PALACE 4320 EL-CAMINO-REAL))
Physics Problems

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

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

42file physgram.lsp
Physics Queries

%lisp
>(load "'/projects/cs375/physics.lsp")

>(phys '(what is the area of a circle
       with diameter = 10))

78.539816339744831

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

35.449077018110316

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

588.399
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?
Mapping

A mapping $M : D \rightarrow R$ specifies a correspondence between elements of a domain $D$ and a range $R$.

If each element of $D$ maps to exactly one element of $R$, and that element $R$ is mapped to only by that one element of $D$, the mapping is one-to-one or injective.

If every element of $R$ is mapped to by some element of $D$, the mapping is onto or surjective.

A mapping that is both one-to-one and onto is bijective.
A mapping can be implemented in several ways:

- A function such as \texttt{sqrt} maps from its argument to the target value.

- If the domain is a finite, compact set of integers, we can store the target values in an array and look them up quickly.

- If the domain is a finite set, we can use a lookup table such as an association list or map data structure such as \texttt{TreeMap} or \texttt{HashMap} in Java or the Clojure maps based on them.

- If the domain is a finite set represented as an array or linked list, we can create a corresponding array or list of target values.
Functional Programming

A *functional* program is one in which:

- all operations are performed by functions
- a function does not modify its arguments or have *side-effects* (such as printing, setting the value of a global variable, writing to disk).

A subset of Lisp or Clojure, with no destructive functions, is an example of a functional language.

```
(defn hypotenuse [x y]
  (sqrt (+ (square x)
            (square y))))
```

Values are passed directly between functions, rather than being stored in variables.

Functional programming is easily adapted to parallel programming: a function can be replicated on many machines, the data set can be broken up into *shards*, and the shards can be processed in parallel on different machines.
Associative and Commutative

An operation \( \circ \) is \textit{associative} if \( a \circ (b \circ c) = (a \circ b) \circ c \).

An operation \( \circ \) is \textit{commutative} if \( a \circ b = b \circ a \).

If an operation \( \circ \) is \textit{both} associative and commutative, then the arguments of the operation can be in any order, and the result will be the same. For example, the arguments of integer \(+\) can be in any order.

This gives great freedom to process the arguments of a function independently on multiple processors.

In many cases, parts of the operation (e.g. partial sums) can be done independently as well.
Computation as Simulation

It is useful to view computation as simulation, cf.: *isomorphism of semigroups*.\(^{43}\)

Given two semigroups \(G_1 = [S, \circ]\) and \(G_2 = [T, \ast]\), an invertible function \(\varphi : S \rightarrow T\) is said to be an *isomorphism* between \(G_1\) and \(G_2\) if, for every \(a\) and \(b\) in \(S\), \(\varphi(a \circ b) = \varphi(a) \ast \varphi(b)\)

from which: \(a \circ b = \varphi^{-1}(\varphi(a) \ast \varphi(b))\)

\[
\begin{array}{c}
\text{problem space} \\
\text{model space}
\end{array}
\]

```
(defn string+ [x y]
  (str ; phi inverse
    (+ ; + in model space
      (read-string x) ; phi
      (read-string y)))); phi

>(string+ "2" "3")
"5"
```

Mapping in Lisp

Clojure and Lisp have functions that compute mappings from a linked list. \texttt{map} makes a new list whose elements are obtained by applying a specified function to each element (\texttt{first}) of the input list(s).

\begin{verbatim}
>(defn square [x] (* x x))

>(map square '(1 2 3 17))

(1 4 9 289)

>(map + '(1 2 3 17) '(2 4 6 8))

(3 6 9 25)

>(map > '(1 2 3 17) '(2 4 6 8))

(false false false true)
\end{verbatim}
Mapcan

The Lisp function `mapcan` works much like `mapcar`, but with a different way of gathering results:

- The function called by `mapcan` returns a list of results (perhaps an empty list).
- `mapcan` concatenates the results; empty lists vanish.

(defn mapcanb [fn args result]
  (if (empty? args)
    result
    (let [res (fn (first args))]
      (if (or (nil? res) (empty? res))
        (mapcanb fn (rest args) result)
        (mapcanb fn (rest args)
          (append (reverse res) result))))
)

(defn mapcan [fn args]
  (reverse (mapcanb fn args '()))
)
Use of Mapcan

mapcan makes it easy to construct a filter function.44

(defn filtr [lst predicate]
  (mapcan (fn [item]
    (if (predicate item)
      (list item) ))
    lst) )

>(filter '(a 2 or 3 and 7) number?)
  ()(2)()(3)() (7)
(2 3 7)

>(filter '(a 2 or 3 and 7) symbol?)
  (a)()(or)()(and)()
(A OR AND)

44filter is a Clojure system function, so a different name is used here.
Input Filtering and Mapping

We can use mapcan to both filter input and map input values to intermediate values for the application.

- **filter**: get rid of uninteresting parts of the input.
- **map**: convert an interesting part of the input to a useful intermediate value.

A key point is that we are not trying to compute the final answer, but to set up the inputs for another function to compute the final answer.

Suppose that we want to count the number of z’s in an input list. We could map a z to a 1, which must be (1) for mapcan; anything else will map to () or nil.

```
(defn testforz [item]
  (if (= item 'z) ; if it is a z
    (list 1) ; emit 1 (map)
    ) ) ; else emit nothing

>(mapcan testforz '(z m u l e z r u l e z))
(1) (1) (1)
(1 1 1)
```
Reduce

The function **reduce** applies a specified function to the first two elements of a list, then to the result of the first two and the third element, and so forth.

\[
\begin{align*}
> (\text{reduce } + \ '(1\ 2\ 3\ 17)) \\
& 23 \\
> (\text{reduce } * \ '(1\ 2\ 3\ 17)) \\
& 102
\end{align*}
\]

**reduce** is what we need to process a result from **mapcan**:

\[
\begin{align*}
> (\text{reduce } + (\text{mapcan testforz} \\
& ' (z\ m\ u\ l\ e\ z\ r\ u\ l\ e\ z))) \\
& ; (1) \quad (1) \quad (1) \\
& ; = (\text{reduce } + \ '(1\ 1\ 1)) \\
& 3
\end{align*}
\]
Combining Map and Reduce

A combination of map and reduce can provide a great deal of power in a compact form.

The Euclidean distance between two points in $n$-space is the square root of the sum of squares of the differences between the points in each dimension.

Using map and reduce, we can define Euclidean distance compactly for any number of dimensions:

\[
\text{(defn edist [pointa pointb]
  (sqrt (reduce +
    (map square
      (map - pointa pointb))))))
\]

> (edist '(3) '(1))

2.0

> (edist '(3 3) '(1 1))

2.8284271247461903

> (edist '(3 4 5) '(2 4 8))

3.1622776601683795
MapReduce and Massive Data

At the current state of technology, it has become difficult to make individual computer CPU’s faster; however, it has become cheap to make lots of CPU’s. Networks allow fast communication between large numbers of cheap CPU’s, each of which has substantial main memory and disk.

A significant challenge of modern CS is to perform large computations using networks of cheap computers operating in parallel.

Google specializes in processing massive amounts of data, particularly the billions of web pages now on the Internet. MapReduce makes it easy to write powerful programs over large data; these programs are mapped onto Google’s network of hundreds of thousands of CPU’s for execution.
Distributed Programming is Hard!

- 1000’s of processors require 1000’s of programs.
- Need to keep processors busy.
- Processors must be *synchronized* so they do not interfere with each other.
- Need to avoid bottlenecks (most of the processors waiting for service from one processor).
- Some machines may become:
  - slow
  - dead
  - evil
  and they may change into these states while your application is running.
- If a machine does not have the data it needs, it must get the data via the network.
- Many machines share one (slow) network.
- Parts of the network can fail too.
What MapReduce Does for Us

MapReduce makes it easy to write powerful programs over large data to be run on thousands of machines.

All the application programmer has to do is to write two small programs:

- Map: Input $\rightarrow$ intermediate value
- Reduce: list of intermediate values $\rightarrow$ answer

These two programs are small and easy to write!
MapReduce does all the hard stuff for us.
Map Sort Reduce

MapReduce extends the Lisp map and reduce in one significant respect: the map function produces not just one result, but a set of results, each of which has a key string. Results are grouped by key.

When our function testforz found a z, it would output (1). But now, we will always produce a key as well, e.g. (z (1)). In Java, we would say:

```
    mr.collect_map("z", list("1"));
```
because the intermediate values are always strings.

There is an intermediate Sort process that groups the results for each key. Then reduce is applied to the results for each key, returning the key with the reduced answer for that key.

At the end of the map and sort, we have:

```
("z" (("1") ("1") ("1")) )
```
with the key and a list of results for that key.
Simplified MapReduce

We think of the map function as taking a single input, typically a String, and emitting zero or more outputs, each of which is a (key, (value)) pair. For example, if our program is counting occurrences of the word liberty, the input "Give me liberty" would emit one output, ("liberty", ("1")).

As an example, consider the problem of finding the nutritional content of a cheeseburger. Each component has a variety of features such as calories, protein, etc. MapReduce can add up the features individually.

We will present a simple version of MapReduce in Lisp to introduce how it works.
Mapreduce in Lisp

(defun mapreduce (mapfn reducefn lst)
  (let (db keylist)
    (dolist (item lst)
      (dolist (resitem (funcall mapfn item))
        (or (setq keylist
                     (assoc (first resitem) db
                         :test 'equal))
            (push (setq keylist
                         (list (first resitem)))
                  db))
        (push (second resitem) (rest keylist)))
    (mapcar #'(lambda (keylist)
                  (list (first keylist)
                        (reduce reducefn
                                   (rest keylist))))
            db))

>(mapreduce 'identity '+'(((a 3) (b 2) (c 1))
                         ((b 7) (d 3) (c 5))))

(((D 3) (C 6) (B 9) (A 3)))
Simple MapReduce Example

\( \text{\texttt{\textgreater (mapreduce 'identity '+}} \) \\
\text{\texttt{'(((a 3) (b 2) (c 1)) \)}} \\
\text{\texttt{((b 7) (d 3) (c 5))) t)}} \) \\
\text{\texttt{Mapping: ((A 3) (B 2) (C 1))}} \\
\text{\texttt{Emitted: (A 3)}} \\
\text{\texttt{Emitted: (B 2)}} \\
\text{\texttt{Emitted: (C 1)}} \\
\text{\texttt{Mapping: ((B 7) (D 3) (C 5))}} \\
\text{\texttt{Emitted: (B 7)}} \\
\text{\texttt{Emitted: (D 3)}} \\
\text{\texttt{Emitted: (C 5)}} \\
\text{\texttt{Reducing: D (3) = 3}} \\
\text{\texttt{Reducing: C (5 1) = 6}} \\
\text{\texttt{Reducing: B (7 2) = 9}} \\
\text{\texttt{Reducing: A (3) = 3}} \\
\text{\texttt{((D 3) (C 6) (B 9) (A 3))}} \)
MapReduce Example

(defun nutrition (food)
  (rest (assoc food
        '((hamburger (calories 80) (fat 8) (protein 20))
          (bun (calories 200) (carbs 40) (protein 8) (fiber 4))
          (cheese (calories 100) (fat 15) (sodium 150))
          (lettuce (calories 10) (fiber 2))
          (tomato (calories 20) (fiber 2))
          (mayo (calories 40) (fat 5) (sodium 20)) ) ) )

  >(nutrition 'bun)
  ((CALORIES 200) (CARBS 40) (PROTEIN 8) (FIBER 4))

  >(mapreduce 'nutrition '+ '(hamburger bun cheese lettuce tomato mayo))
  ((SODIUM 170) (FIBER 8) (CARBS 40) (PROTEIN 28) (FAT 28) (CALORIES 450))
Hamburger Example

>(mapreduce 'nutrition '+
   '(hamburger bun cheese lettuce tomato mayo) t)

Mapping: HAMBURGER
    Emitted: (CALORIES 80)
    Emitted: (FAT 8)
    Emitted: (PROTEIN 20)

Mapping: BUN
    Emitted: (CALORIES 200)
    Emitted: (CARBS 40)
    Emitted: (PROTEIN 8)
    Emitted: (FIBER 4)

Mapping: CHEESE
    Emitted: (CALORIES 100)
    Emitted: (FAT 15)
    Emitted: (SODIUM 150)

Mapping: LETTUCE
    Emitted: (CALORIES 10)
    Emitted: (FIBER 2)

Mapping: TOMATO
    Emitted: (CALORIES 20)
    Emitted: (FIBER 2)

Mapping: MAYO
    Emitted: (CALORIES 40)
    Emitted: (FAT 5)
    Emitted: (SODIUM 20)

Reducing: SODIUM (20 150) = 170
Reducing: FIBER (2 2 4) = 8
Reducing: CARBS (40) = 40
Reducing: PROTEIN (8 20) = 28
Reducing: FAT (5 15 8) = 28
Reducing: CALORIES (40 20 10 100 200 80) = 450

((SODIUM 170) (FIBER 8) (CARBS 40) (PROTEIN 28) (FAT 28) (CALORIES 450))
How MapReduce Works

There is a single *Master* computer and many *Worker* computers.

The Master divides the input data into bite-size chunks of 64 MB and assigns the data chunks to workers. If possible, Master chooses a worker that already has the data on its hard drive in the Google File System, or is close to a computer with the data; this minimizes network traffic.

Think of the data chunks as being like a sack of beans: lots of pieces of data, all more or less alike.
Map Worker

A Map Worker runs the Map program on its assigned data. The Map program receives as input \((\text{inputkey}, \text{inputvalue})\) pairs; for example, \text{inputkey} could be the IP address of a web page (as a string) and \text{inputvalue} could be the contents of that web page (all as one string).

The Map worker emits \((\text{outputkey}, \text{list(mapvalue)})\) pairs. \text{outputkey} could be the same as \text{inputkey}, but often is different. For example, to count links to a web page, \text{outputkey} could be the IP address of a page that is linked to by the page being processed.

If there are \(R\) Reduce Workers, the \text{outputkey} is hashed modulo \(R\) to determine which Reduce Worker will get it; hashing randomizes the assignment of keys to Reduce Workers, providing load balancing.

The Map Worker has \(R\) output buffers corresponding to \(R\) files that it is producing as output, one for each Reduce Worker. The \((\text{outputkey}, \text{list(mapvalue)})\) pair is put into the corresponding output buffer.
Buffering

Buffering is a technique used to match a small-but-steady process (e.g. a program that reads or writes one line at a time) to a large-block process (e.g. disk I/O).

Disk I/O has two problematic features:

- A whole disk block (e.g. 4096 bytes) must be read or written at a time.
- Disk access is slow (e.g. 8 milliseconds).

An I/O buffer is an array, the same size as a disk block, that is used to collect data. The application program removes data from the block (or adds data to it) until the block is empty (full), at which time a new block is read from disk (written to disk).

If there are $R$ Reduce tasks, each Map task will have $R$ output buffers, one for each Reduce task. When an output buffer becomes full, it is written to disk. When the Map task is finished, it sends the file names of its $R$ files to the Master.
Load Balancing

Some data values are much more popular than others. For example, there were 13 people on a class roster whose names started with $S$, but only one $K$, and no $Q$ or $X$.

If MapReduce assigned Reduce tasks based on key values, some Reduce tasks might have large inputs and be too slow, while other Reduce tasks might have too little work.

MapReduce performs load balancing by having a large number $R$ of Reduce tasks and using hashing to assign data to Reduce tasks:

$$task = Hash(key) \mod R$$

This assigns many keys to the same Reduce task. The Reduce task reads the files produced by all Map tasks for its hash value (remote read over the network), sorts the combined input by key value, and appends the value lists before calling the application’s Reduce function.
Reduce Worker

A Reduce Worker receives from the Master a set of $M$ file addresses, one for each Map worker. The Reduce worker reads these files; these reads must go across the network, and therefore may take some time and cause network congestion.

The Reduce Worker first sorts its input data by $key$ and groups together all the data values for each $key$. It then runs the Reduce program on each data set.

The result is a list, $(key, list(value))$; these are put into the output buffer of the Reduce worker (these will now be sorted by key). When done, the Reduce worker send the file address of its output file to the Master.

The Master can finally combine all the output files from Reduce workers into sorted order by doing a Merge.
PageRank

The PageRank algorithm used by Google expresses the ranking of a web page in terms of two components:

- a base value, \((1 - d)\), usually 0.15
- \(d \times \sum_{i \in \text{links}} \frac{PR_i}{n_i}\) where \(PR_i\) is the page rank of a page that links to this page, and \(n_i\) is the number of links from that page.

The PageRank values can be approximated by relaxation by using this formula repeatedly within MapReduce. Each page is initially given a PageRank of 1.0; the sum of all values will always equal the number of pages.

- **Map:** Share the love: each page distributes its PageRank equally across the pages it links to.
- **Reduce:** Each page sums the incoming values, multiplies by 0.85, and adds 0.15.
PageRank Example

Iterative PageRank converges fairly quickly for this net:\(^45\)

\[
\begin{array}{ccc}
A & B & C \\
1.0000000 & 1.0000000 & 1.0000000 \\
1.0000000 & 0.5750000 & 1.4250000 \\
1.3612500 & 0.5750000 & 1.0637500 \\
1.0541875 & 0.7285312 & 1.2172813 \\
1.1846891 & 0.5980297 & 1.2172813 \\
1.1846891 & 0.6534929 & 1.1618181 \\
1.1375454 & 0.6534929 & 1.2089618 \\
1.1776175 & 0.6334568 & 1.1889257 \\
1.1605868 & 0.6504874 & 1.1889257 \\
1.1605868 & 0.6432494 & 1.1961637 \\
1.1667392 & 0.6432494 & 1.1900114 \\
1.1615097 & 0.6458642 & 1.1926261 \\
1.1637322 & 0.6436416 & 1.1926261 \\
... \\
1.1633691 & 0.6444319 & 1.1921989 \\
\end{array}
\]

The sum of PageRank values is the total number of pages. The value for each page is the expected number of times a random web surfer, who starts as many times as there are web pages, would land on that page.

\(^{45}\)http://pr.efactory.de/e-pagerank-algorithm.shtml
Running PageRank Example

Starting MapReduce on:

\[((a (1.0 (b c))) (b (1.0 (c))) (c (1.0 (a))))\]

- mapping: key = a val = (1.0 (b c))
- emitting: key = b val = (0.5)
- emitting: key = c val = (0.5)
- emitting: key = a val = ((b c))

- mapping: key = b val = (1.0 (c))
- emitting: key = c val = (1.0)
- emitting: key = b val = ((c)) ...

- reducing: key = a val = (((b c)) (1.0))
- result: key = a val = (1.0 (b c))

- reducing: key = b val = ((0.5) ((c)))
- result: key = b val = (0.575 (c))

- reducing: key = c val = ((0.5) (1.0) ((a)))
- result: key = c val = (1.425 (a))

Starting MapReduce on:

\[((a (1.0 (b c))) (b (0.575 (c))) (c (1.425 (a))))\]

- reducing: key = a val = (((b c)) (1.425))
- reducing: key = b val = ((0.5) ((c)))
- reducing: key = c val = ((0.5) (0.575) ((a)))

Starting MapReduce on:

\[((a (1.36125 (b c))) (b (0.575 (c))) (c (1.06375 (a))))\]

- reducing: key = a val = (((b c)) (1.06375))
- reducing: key = b val = ((0.680625) ((c)))
- reducing: key = c val = ((0.680625) (0.575) ((a)))

... after 10 steps:

Result = ((a (1.16673918 (b c)))

(b (0.64324941 (c)))

(c (1.19001141 (a))))
Advanced Performance

The notions of Big O and single-algorithm performance on a single CPU must be extended in order to understand performance of programs on more complex computer architectures. We need to also account for:

- Disk access time
- Network bandwidth and data communication time
- Coordination of processes on separate machines
- Congestion and bottlenecks as many computers or many users want the same resource.
Performance Techniques in MapReduce

- The Google File System (GFS) stores multiple copies (typically 3) of data files on different computers for redundancy and availability.

- Master assigns workers to process data such that the data is on the worker’s disk, or near the worker within the same rack. This reduces network communication; network bandwidth is scarce.

- Combiner functions can perform partial reductions (adding "1" values) before data are written out to disk, reducing both I/O and network traffic.

- Master can start redundant workers to process the same data as a dead or “slacker” worker. Master will use the result from the worker that finishes first; results from later workers will be ignored.

- Reduce workers can start work as soon as some Map workers have finished their data.
Algorithm Failure

If MapReduce detects that a worker has failed or is slow on a Map task, it will restart redundant Map tasks to process the same data.

If the redundant Map tasks also fail, maybe the problem is that the data caused the algorithm to fail, rather than hardware failure.

MapReduce can restart the Map task without the last unprocessed data. This causes the output to be not quite right, but for some tasks (e.g. average movie rating) it may be acceptable.
Atomic Commit

In CS, the word *atomic*, from Greek words meaning *not cut*, describes an all-or-nothing process: either the process finishes without interruption, or it does not execute at all.

If multiple worker machines are working on the same data, it is necessary to ensure that only one set of result data is actually used.

An *atomic commit* is provided by the operating system (and, ultimately, CPU hardware) that allows exactly one result to be committed or accepted for use. If other workers produce the same result, those results will be discarded.

In MapReduce, atomicity is provided by the file system. When a Map worker finishes, it renames its temporary file to the final name; if a file by that name already exists, the renaming will fail.