CS 378, Symbolic Programming

Gordon S. Novak Jr.

Department of Computer Sciences
University of Texas at Austin
novak@cs.utexas.edu
http://www.cs.utexas.edu/users/novak

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, or *denote*, other things: their *referents*:

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

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

- solve equations
- translate a program to another language
- specialize a generic program
- 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 `sin(x)`, it will just return a value, but will have no side effects.

Functional programming does everything by *composition* of functions:

```
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.[1] 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

[1] 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)          ; 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))
ford
```

List

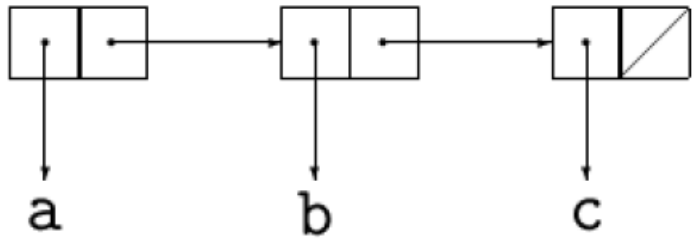
A *list* is made of two-element data structures, called *conses* after the function `cons` that makes them.

- a *link* , called `rest`, that points to the next element; or `()` or `nil`, an *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:

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




```
(list (+ 2 3) (* 2 3))  ->  (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)
```

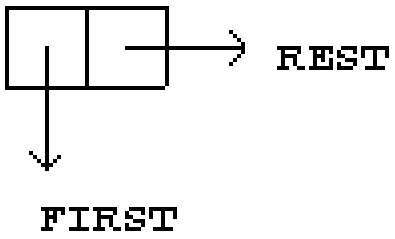
An easy way to think of `(cons x y)` is that it makes a new cons box whose `first` is x and whose `rest` is y, or that `cons` adds a new item to the front of an existing list.

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]




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

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

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

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

(defn third [x] (first (rest (rest x))))
(third '(a b c))                    ->  c

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

[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*; *cf.* the C form *test* `?` *thenval* `:` *elseval*

```
(if true 2 3)        ->  2

(if false 2 3)       ->  3

(if '() 2 3)         ->  2

(if (+ 2 3) 2 3)     ->  2
```

Tests on `cons`

`(cons?` *item* `)` tests whether *item* is a `cons`, i.e. non-empty part of a list or tree structure. `cons?` is not a built-in Clojure function; do `(load-file "cs378.clj")`

`(empty?` *item* `)` tests whether *item* is empty, i.e. either `nil` or `'()` .

These special tests are needed because Clojure is inconsistent in the ways it represents list structure: `nil` and `'()` are different.

Clojure error messages can be confusing since they come from the Java Virtual Machine rather than Clojure.

```
user=> (first 3.14)

Execution error ...
Don't know how to create ISeq from:
    java.lang.Double
```

Translated into English, this means: you tried to do `first` or `rest` on something that is not a `cons`; it was (in this case) a `Double`. So, test `cons?` before doing `first` or `rest`.

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.

```
(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*.

Tracing Function Calls

There is a trace package that allows tracing function calls:

```
(load-file "cs378/trace.clj")
user=> (trace (* 2 3))
TRACE: 6
6

(deftrace factorial [n]
  (if (= n 0) 1 (* n (factorial (- n 1)))))
user=> (factorial 4)
TRACE t256: (factorial 4)
TRACE t257: | (factorial 3)
TRACE t258: | | (factorial 2)
TRACE t259: | | | (factorial 1)
TRACE t260: | | | | (factorial 0)
TRACE t260: | | | | => 1
TRACE t259: | | | => 1
TRACE t258: | | => 2
TRACE t257: | => 6
TRACE t256: => 24
24
```

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:

```
(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
```

Note that we are using `empty?` to test for an empty list; this is because Clojure could represent the end of a list in different ways.

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.

Filter a List

A *filter* keeps items in a list that pass some test, while removing other items.

```
(defn myfilter [fn lst]
  (if (empty? lst)        ;  base case
      '()
      (if (fn (first lst))   ; does it pass?
          (cons (first lst)  ; yes: keep it
                (myfilter fn (rest lst)))
          ; else: ignore it but keep going
          (myfilter fn (rest lst)) ) ) )
```

```
user=> (myfilter number? '(2 a d 4 3 d e 8))

(2 4 3 8)
```

`filter` is a built-in function of Clojure.

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]

```
(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 answer variable
```

[3]Note that we define 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 that are done, *i.e.* `cons` is like a push onto a stack.

```
(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; it takes $O(n_x)$ time and $O(n_x)$ stack space. Time and stack space are independent of y, since y is reused but not processed.

Append: Big O Hazard

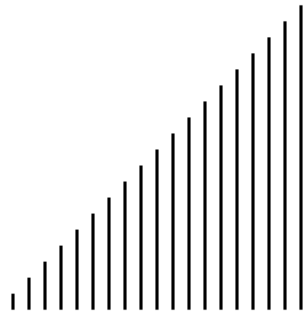
Suppose we want to make a list `(0 1 ... n-1)`.

```
user=> (listofn 5)
(0 1 2 3 4)

(defn listofnb [n i answer]          ; version 1
  (if (< i n)
      (listofnb n
                (+ i 1)
                (append answer (list i)) )
      answer))

(defn listofnb [n i answer]          ; version 2
  (if (< i n)
      (listofnb n
                (+ i 1)
                (cons i answer))  ; backwards
      (reverse answer)))  ; fix backwards

(defn listofn [n] (listofnb n 0 '() ) )
```

What is the Big O of each version?

Beware the Bermuda Triangle

Some gym teachers punish misbehaving students by making them run 50 meters. However, the way in which they must do it is to run 1 meter, then come back, then run 2 meters, then come back, ...



How many total meters does a student run?

$$\sum_{i=1}^{n} i = n * (n+1)/2 = O(n^2)$$

```
for ( i = 0; i < n; i++ )       ; i up to n times
  for ( j = 0; j <= i; j++ )    ; O(i)
    sum += a[i][j];
```

Rule: If a computation of $O(i)$ is inside a loop where i is *growing* up to n, the total computation is $O(n^2)$.

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.

```
(member 'b '(a b c))    ->    (b c)

(member 'f '(a b c))    ->    nil

(defn member [item lst]
  (if (empty? lst)
      nil
      (if (= item (first lst))
          lst
          (member item (rest lst)) ) ) )
```

[4] Clojure has sets as a built-in type, which is more efficient for large sets than what is shown here.

Intersection

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

```
(intersection '(a b c) '(a c e))   ->  (c a)
```

```
(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 (INTERSECTTRB (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.

```
(union '(a b c) '(a c e))   ->  (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.

```
(set-difference '(a b c) '(a c e))   ->  (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.

Let

The `let` construct in Clojure allows local variables to be defined and initialized.

`(let [` var_1 $init_1$ ... `]` *code* `)`

```
(let [ d (* 2.0 r)
       c (* Math/PI d) ]   ... )
```

Each variable is initialized with the value of the corresponding initialization code; these initialization steps are executed in order, so the result of one can be used in later init code.[6]

`let` is useful to save the result of a (possibly expensive) function call so that it can be used multiple times without having to be recomputed.

There can be multiple forms inside the `let`; the value of the `let` is the value of the last form.

```
(defn lookupstudent [eid]
  (let [idname (assocl eid alist)]
    (if idname
        (second idname)
        "John Doe") ) )
```

[6]This is called `let*` is some Lisp dialects.

Map

The `map` function applies a given function to each element of a list, producing a new list of the results:

`(map` *function* *list* `)`

```
(map symbol? '(2 medium 7 large))
    ->  (false true false true)

(map (fn [x] (* 2 x)) '(2 3 7))
    ->  (4 6 14)
```

Note that this example used an *anonymous function*:

`(fn [` *args* `]` *code* `)`

This is useful when the function is small and not worth giving it a name as a separate function.

We could define `length` (not very efficiently) by mapping a list to a list of `1`'s, which we add using `reduce` and `+`:

```
(defn length [x]
  (reduce + (map (fn [z] 1) x)))

 (length '(2 medium 3 large))
    ->  (1 1 1 1)    ->  4
```

Some and Every

The `some` function (∃ or *there exists* in math notation) applies a given function to each element of a list, returning the first result that is not `nil` or `false`:

`(some` *function list* `)`

```
(some (fn [x] (and (> x 3) x))
      '(1 2 17 4 0) )
    ->  17
```

Note that in this case we both tested for the desired item `(> x 3)` and then returned the item itself, since we want that answer rather than `true` as returned by the test.

The `every?` function (∀ or *for all* in math notation) applies a given function to each element of a list, returning `true` if the function is not `nil` or `false` for every list element.

`(every?` *function list* `)`

```
(every? (fn [x] (> x 3)) '(17 4 5))
    ->  true
```

println, do, when, str

The `println` function prints its arguments:

```
(let [n 3] (println "n = " n))
    ->  n =  3
```

The `do` form wraps multiple forms and executes them in order, similar to { ... } in C-like languages. Use `do` when printing, since printing is a side-effecting operation. The value of `do` is the value of the last form.

`when` is like a combination of `if` and a `do` in the true branch (with no false branch):

```
(when (> 5 3)
  (println "Whew!")
  (println "I'd be worried otherwise."))
```

`str` makes a string from the values of its arguments.

```
user=>(str 3 " and " '(a b))
"3 and (a b)"
```

doseq and sort

`doseq` is a convenient looping function to do the same thing for every element of a list:

```
user=> (doseq [item '(a b c)]
         (println "item = " item))
item =  a
item =  b
item =  c
nil           ; value of doseq
```

There is a convenient `sort` function that makes a sorted version of a list (nondestructive):

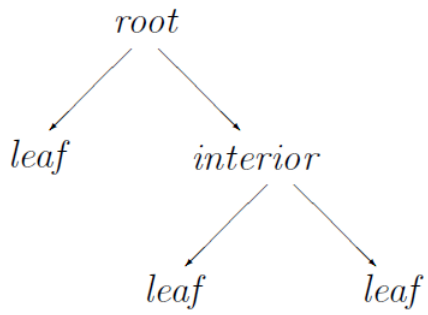
```
user=> (let [ lst    '(z a p)
              sorted (sort lst) ]
         (list lst sorted) )

((z a p) (a p z))
```

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*.

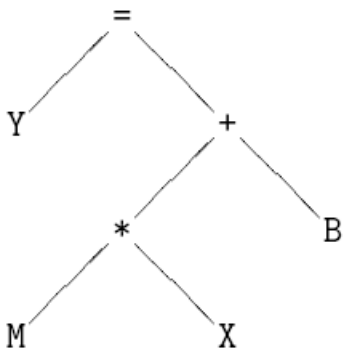


Trees occur in many places in computer systems and in nature.

Arithmetic Expressions as Trees

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

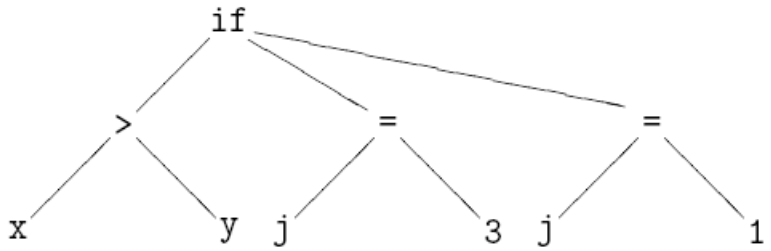
```
y = m * x + b          (= y (+ (* m x) b))
```



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.

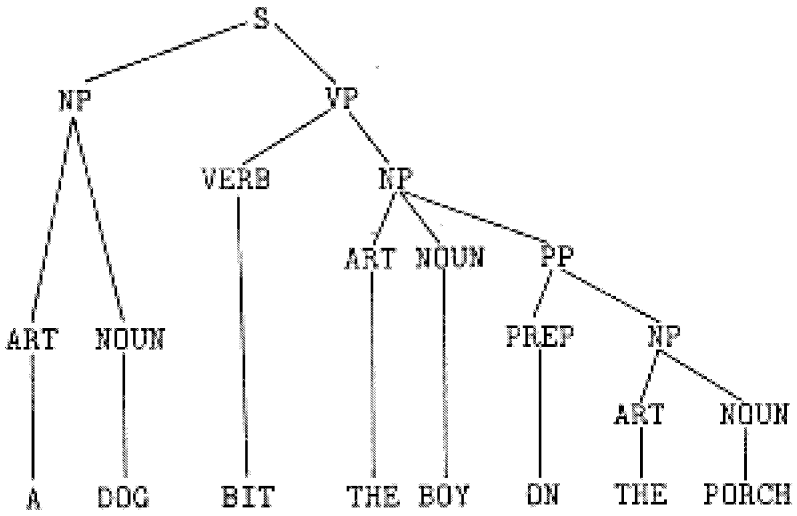
```
if ( x > y )
   j = 3;
 else
   j = 1;
```



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.

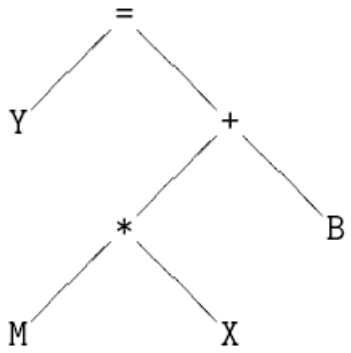


The speaker wants to communicate a structure, but must make it linear in order to say it. The listener needs to re-create the structure intended by the speaker. Parts of the *parse tree* can then be related to object symbols in memory.

Representations of Trees

Many different representations of trees are possible:

- Binary tree: contents and left and right links.
  - arithmetic expressions



  - `cons` as a binary tree node, with `first` and `rest` treated equally as binary 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.

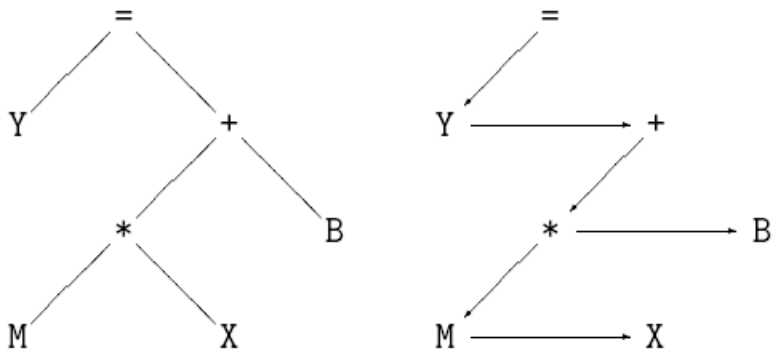
First-Child / Next-Sibling Tree

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 * x + b
```

Tree: **Representation:**



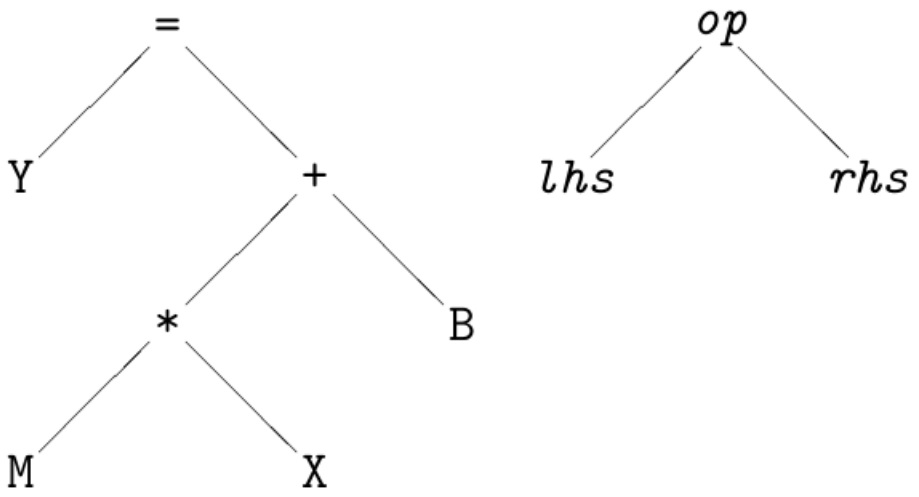
The layout of nodes in this kind of tree is the same as for a traditional tree; the right-hand links "fall down" and become horizontal.

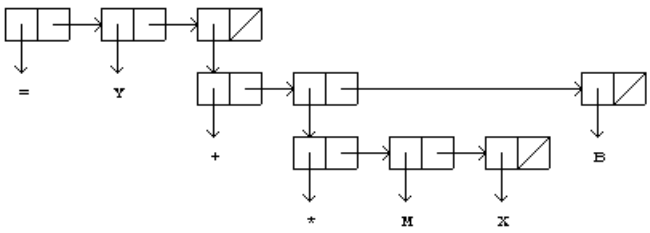
Down arrows represent the `first` child, while side arrows represent the `next` sibling.

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))      (op lhs rhs)
```






```
(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.

```
(defn addnums [tree]
  (if (cons? tree)           ; interior node
      (+ (addnums (first tree))
         (addnums (rest tree)) )
      (if (number? tree)          ; leaf node
          tree
          0) ) )   ; safe: identity value

(defn symbolset [tree]
  (if (cons? tree)           ; interior node
      (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 itsefl *twice* for interior nodes. This is essentially the same as the divide-and-conquer design pattern.

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

```
(defn addnums [tree]
  (if (cons? tree)           ; interior node
      (+ (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 `cons`) can be used to accumulate the result. This accumulation of a result in an extra variable is similar to tail recursion.

```
(defn flattenbtb [tree result]
  (if (cons? tree)           ; interior node
      (flattenbtb (lhs tree)          ; 3. L child
                  (cons (op tree)     ; 2. parent
                        (flattenbtb
                          (rhs tree) ; 1. R child
                          result)))
      (if (not (null? tree))
          (cons tree result)
          result) ) )

(defn flattenbt [tree] (flattenbtb tree '()) )
```

Examples: Flattening Tree

```
>(flattenbt '(cat (bat ape
                       bee)
                  (eel dog
                       fox)))

(ape bat bee cat dog eel fox)

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

(y = m * x + b)
```

Tracing Flattening Binary Tree

```
>(flattenbt '(cat (bat ape
                       bee)
                  (eel dog
                       fox)))

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

Search

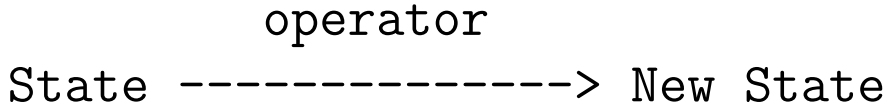
Search programs find a solution for a problem by trying different *sequences* of *actions* (*operators*) until a solution is found.

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

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

State Space Search

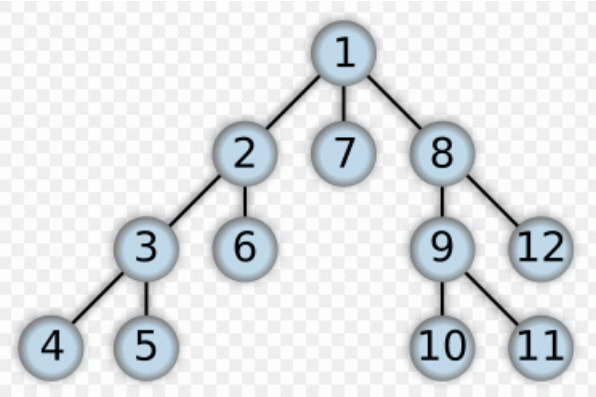
A *state space* represents a problem in terms of *states* and *operators* that change states.



A state space consists of:

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

Depth-First Order: Recursive Default



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.

Tree Recursion

Writing a tree-recursive program requires answering several questions:

- What are the branches?
  - `first` and `rest`
  - `lhs` and `rhs`
- How should results from branches be combined?
  - additive
    - ordered, e.g. consing onto ordered list
    - unordered, e.g addition or union
  - structural
  - search: first one that works
- Is an ordering of the tree search required by the domain?
  - preorder: parent before children
  - inorder: one child, parent, other child
  - postorder: children first, then parent, e.g. must evaluate children before performing operation on them

DFS: Code to Build Answer

Depth-first Search tries operators one at a time; after applying one operator to produce a new state, it calls itself recursively to see if the goal can be reached from that state.

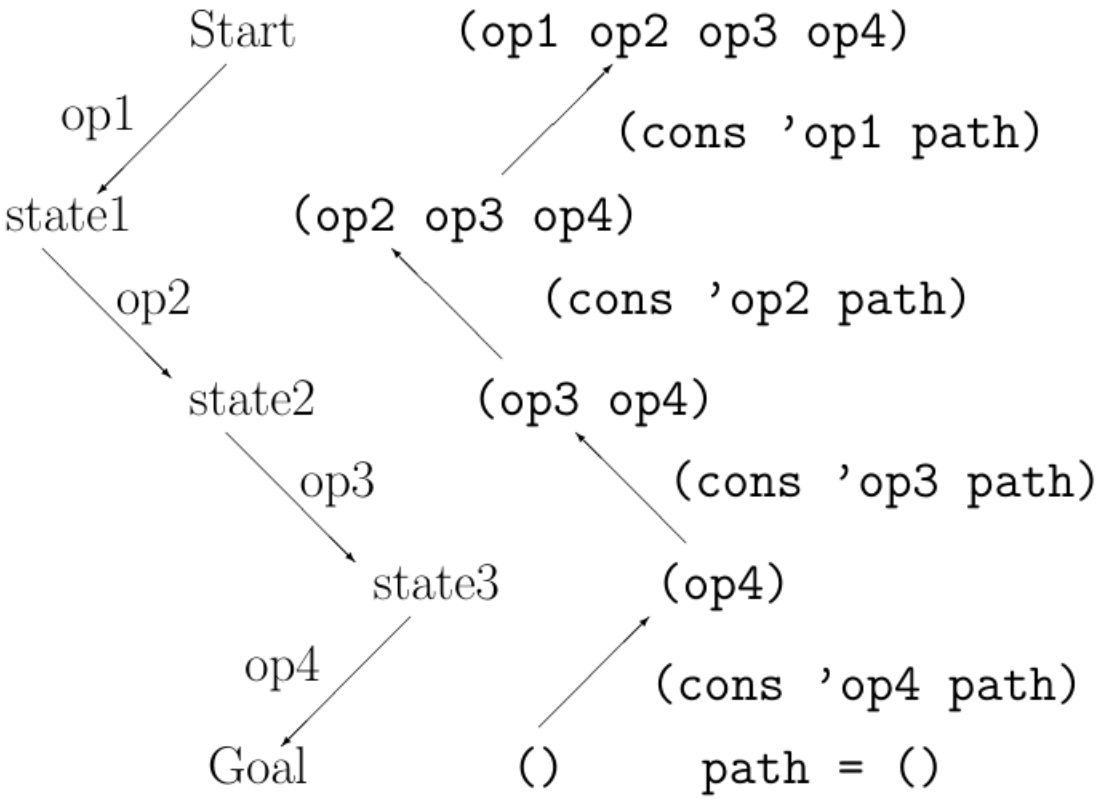
```
      apply op                search
state ---------> newstate --- ... ---> Goal
  <- (cons op path) <- path = ( ops ) <- ()
```

If `path` is a list of ops to get from `newstate` to a goal, then `(cons op path)` will get from `state` to a goal.

```
(defn search [state]
  (if (goal? state)
      '()   ; no ops needed to get to goal
      (if (failure? state)
          nil
          (let [newstate (op1 state)]
            (let [path (search newstate)]
              (if path
                  (cons 'op1 path)
                  ... try another op
                  )))) ))
```

Answer for State Space Search

In state space search, we want to find a *sequence of operators* that will lead from a start state to a goal state.



Each call to the search function returns either `nil` (failure) or a list of operators that lead to a goal state. At each state, if applying an operator leads to a goal, we `cons` that operator onto the list of operators returned from below by the search.

Design Pattern: Depth-first Search

```
(defn search [state]    ; returns path to
goal
  (if (goal? state)
      '()    ; no ops needed to get to goal
      (if (failure? state)
        nil
        (some
          (fn [op]
            (let
              [path (search
                      (applyop op state))]
              (and path (cons op path)) ))
          (operators state)) )))
```

Complications:

- It may not be possible to apply `op`.
- Applying `op` might violate a constraint.
- We could get into a loop applying `op` and its inverse, or going in a circle back to the same state.

Comments on Search Algorithm

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

Tail-Recursive Search

The search algorithm can also be written in a tail-recursive style.

- The path (sequence of operators) to the goal is passed down as a variable, initially `'()`.
- Each time an operator `op` is applied to create a new state, `op` is `cons`ed onto the path for the recursive call. Since `cons` acts as a stack and makes a list in backwards order, the path is a backwards list of operators taken from the start.
- When the goal is found, a `reverse` of the path gives a solution (sequence of operators to get from the start to the goal).

Robot Mouse in Maze

Depth-first search of an implicit tree can simulate a robot mouse in a maze. The goal is to return a sequence of steps to guide the mouse to the cheese.

```
(defn mouse [maze x y prev]
(if (or (= (nth (nth maze y) x) '*) ; hit wall
        (member (list x y) prev)) ; been there
    nil                           ; fail
    (if (= (nth (nth maze y) x) 'c) ; cheese
        '()                       ; success
        (let [path (mouse maze (- x 1) y ; go west
                           (cons (list x y) prev))]
          (if path (cons 'w path)
              (let [path (mouse maze x  (- y 1)
                           (cons (list x y) prev))]
                (if path (cons 'n path)
                    (let [path (mouse maze (+ x 1) y
                           (cons (list x y) prev))]
                     (if path (cons 'e path)
                         (let [path (mouse maze x (+
                           (cons (list x y) prev))]
                           (if path (cons 's path)
                               nil))))))))))     ;
```

Robot Mouse Program

- The maze is a 2-D array. `*` represents a wall. `0` represents an open space. `c` represents cheese.
- The mouse starts in an open position.
- The mouse has 4 possible moves at each point: `w`, `n`, `e` or `s`.
- We have to keep track of where the mouse has been, or it might wander infinitely in circles. The list `prev` is a stack of previous states. If the mouse re-visits a position on `prev`, we want to fail.
- We need to return an answer:
  - `nil` for failure
  - a list of moves that will lead from the current position to the goal; for the goal itself, we return `()`. As we unwind the recursion, we will push the operator that led to the goal onto the answer list at each step.

Robot Mouse Example

```
(def maze
;  0 1 2 3 4 5 6 7 8 9
'((* * * * * * * * * *)    ; 0
  (* 0 0 * * * * * * *)    ; 1
  (* 0 * * * * * * * *)    ; 2
  (* 0 * * * * * * * *)    ; 3
  (* 0 0 0 0 0 0 * * *)    ; 4
  (* * * * 0 * 0 * * *)    ; 5
  (* * * * 0 * 0 * c *)    ; 6
  (* * * * 0 * 0 * 0 *)    ; 7
  (* * * * 0 * 0 0 0 *)    ; 8
  (* * * * 0 * * * * *)))  ; 9

user=> (mouse maze 4 9 '())
(n n n n n e e s s s s e e n n)
```

Tracing the Robot Mouse

```
>(mouse maze 4 9 '())
  1> (MOUSE #2A((* * * * * * * * * *)
                (* 0 0 * * * * * * *)
                (* 0 * * * * * * * *)
                (* 0 * * * * * * * *)
                (* 0 0 0 0 0 0 * * *)
                (* * * * 0 * 0 * * *)
                (* * * * 0 * 0 * C *)
                (* * * * 0 * 0 * 0 *)
                (* * * * 0 * 0 0 0 *)
                (* * * * 0 * * * * *)) 4 9 NIL)
    2> (MOUSE 3 9 ((4 9)))          ; west
    <2 (MOUSE NIL)                  ; hit the wall
    2> (MOUSE 4 8 ((4 9)))          ; north
      3> (MOUSE 3 8 ((4 8) (4 9)))    ; west
      <3 (MOUSE NIL)                  ; hit the wall
      3> (MOUSE 4 7 ((4 8) (4 9)))    ; north
        4> (MOUSE 4 6 ((4 7) (4 8) (4 9)))    ; north
          5> (MOUSE 4 5 ((4 6) (4 7) (4 8) (4 9)))    ; north
            6> (MOUSE 4 4 ((4 5) (4 6) (4 7) (4 8) (4 9))) ; north
              7> (MOUSE 3 4 ((4 4) (4 5) (4 6) (4 7) (4 8) ; west
                8> (MOUSE 2 4 ((3 4) (4 4) (4 5) (4 6)     ; west
                  9> (MOUSE 1 4 ((2 4) (3 4) (4 4) (4 5)   ; west
                    10> (MOUSE 0 4 ((1 4) (2 4) (3 4) (4 4) ; west
                    <10 (MOUSE NIL)         ; hit the wall
                    10> (MOUSE 1 3 ((1 4) (2 4) (3 4) (4 4) ; north
                    11> (MOUSE 1 2 ((1 3) (1 4) (2 4) (3 4)
                    12> (MOUSE 1 1 ((1 2) (1 3) (1 4) (2 4)
                    13> (MOUSE 1 0 ((1 1) (1 2) (1 3) (1 4) ; north
                    <13 (MOUSE NIL)         ; hit the wall
                    13> (MOUSE 2 1 ((1 1) (1 2) (1 3) (1 4) ; east
                    14> (MOUSE 1 1 ((2 1) (1 1) (1 2) (1 3) ; west
                    <14 (MOUSE NIL)   ; ! loop
```

Tracing the Robot Mouse ...

```
                14> (MOUSE 3 1 ((2 1) (1 1) (1 2) (1 3)
                <14 (MOUSE NIL)
                14> (MOUSE 2 2 ((2 1) (1 1) (1 2) (1 3)
                <14 (MOUSE NIL)
                <13 (MOUSE NIL) ...
            <7 (MOUSE NIL)                  ; fail back to (4 4)
            7> (MOUSE 5 4 ((4 4) (4 5) (4 6) (4 7)        ; east
              8> (MOUSE 6 4 ((5 4) (4 4) (4 5) (4 6)
                9> (MOUSE 6 5 ((6 4) (5 4) (4 4) (4 5)    ; south
                  10> (MOUSE 6 6 ((6 5) (6 4) (5 4)
                  11> (MOUSE 6 7 ((6 6) (6 5) (6 4)
                  12> (MOUSE 6 8 ((6 7) (6 6) (6 5)       ; south
                  13> (MOUSE 7 8 ((6 8) (6 7) (6 6)       ; east
                  14> (MOUSE 8 8 ((7 8) (6 8) (6 7)       ; east
                  15> (MOUSE 8 7 ((8 8) (7 8) (6 8)       ; north
                  16> (MOUSE 8 6 ((8 7) (8 8) (7 8)       ; north
                  <16 (MOUSE (CHEESE))        ; found the cheese!
                  <15 (MOUSE (N CHEESE))      ; last move was N
                  <14 (MOUSE (N N CHEESE))    ; push on operators
                  <13 (MOUSE (E N N CHEESE))    ; as we backtrack
                  <12 (MOUSE (E E N N CHEESE))
                  <11 (MOUSE (S E E N N CHEESE))
                  <10 (MOUSE (S S E E N N CHEESE))
                <9 (MOUSE (S S S E E N N CHEESE))
              <8 (MOUSE (S S S S E E N N CHEESE))
            <7 (MOUSE (E S S S S E E N N CHEESE))
          <6 (MOUSE (E E S S S S E E N N CHEESE))
        <5 (MOUSE (N E E S S S S E E N N CHEESE))
      <4 (MOUSE (N N E E S S S S E E N N CHEESE))
    <3 (MOUSE (N N N E E S S S S E E N N CHEESE))
  <2 (MOUSE (N N N N E E S S S S E E N N CHEESE))
<1 (MOUSE (N N N N N E E S S S S E E N N CHEESE))
(N N N N N E E S S S S E E N N CHEESE)
```

Depth-First Search

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

Advantages:

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

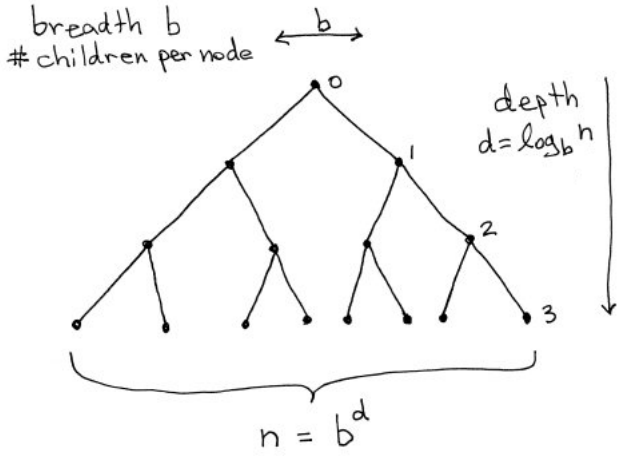
Disadvantages:

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

Big O for Trees

If a tree is uniform and balanced, we can describe it in terms of several parameters:

- **b**, the *breadth* or *branching factor*, is the number of branches per interior node. For a binary tree, $b = 2$.
- **d** is the *depth*, the height of the tree. $d = log_b(n)$
- **n** is the number of leaf nodes. $n = b^d$



Note that *most of the nodes are on the bottom row of the tree.* If $b = 2$, half the nodes are on the bottom; if b is higher, an even greater proportion will be on the bottom.

In general, a tree algorithm will have Big O:

- if one branch of the tree is followed and the others are abandoned, $O(log(n))$: proportional to depth.
- if all branches of the tree are processed, $O(n * log(n))$

Bounded Depth-First Search

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

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

Problems:

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

Iterative Deepening

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

Advantages:

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

Disadvantage:

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

Cost of Iterative Deepening

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

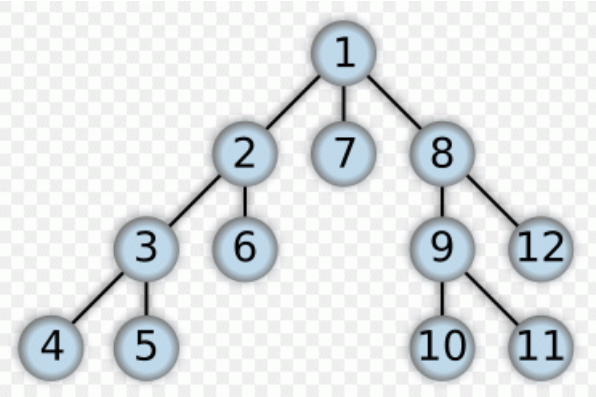
Korf calculates the work done by iterative deepening as $b^d * (1 - 1/b)^{-2}$, where the multiplier approaches 1 as b increases.[7]

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

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

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

Depth-First Search



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.

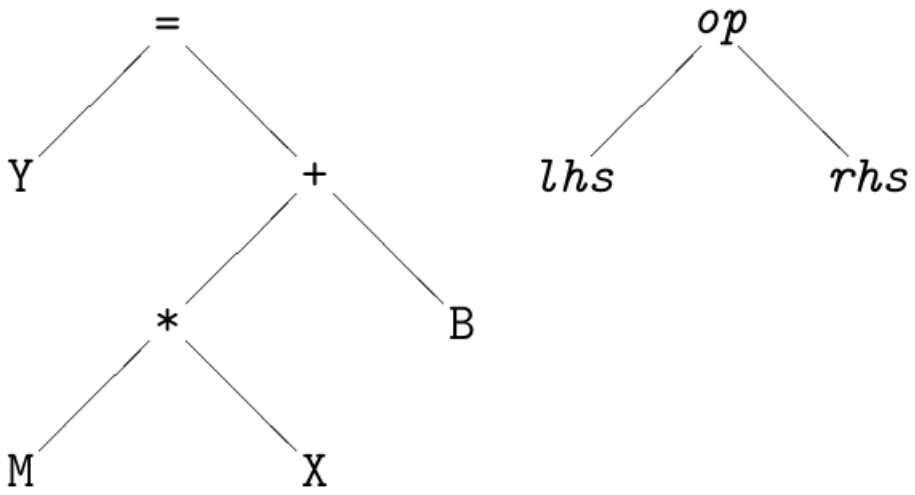
Depth-first search (*DFS*) follows an implicit tree of size $O(b^{depth})$, where b is the *branching factor*. (number of children of a node). Given a state, we test whether it is a goal or a terminal failure node; if not, we generate successor states and try searching from each of them. Many of these 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 `(` *op lhs rhs* `)` recursively
- **Tree:**



- **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 `lhs` of `e` is `v`, return `e`.
  - If the `rhs` of `e` is `v`, rewrite `e` to switch the `lhs` and `rhs` of `e`, and return that.
  - If only an undesired variable or constant is on the right, (`rhs` is not a cons), fail by returning `nil`.
- **Recursive case:** Rewrite `e` using an algebraic law, and try to solve that equation. Return the first result that is not `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:
`(=` α `(` *op* β γ `))`

We are hoping that the desired variable will be somewhere in β or γ; 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 β or γ.

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:
`(=` α `(+` β γ `))`

we can construct two new possibilities:

`(= (-` α β `)` γ `)` (subtract β from both sides)

`(= (-` α γ `)` β `)` (subtract γ from both sides)

After making a new equation, we simply call `solve` to try to solve *that* equation. We return the first solution that is not `nil`.

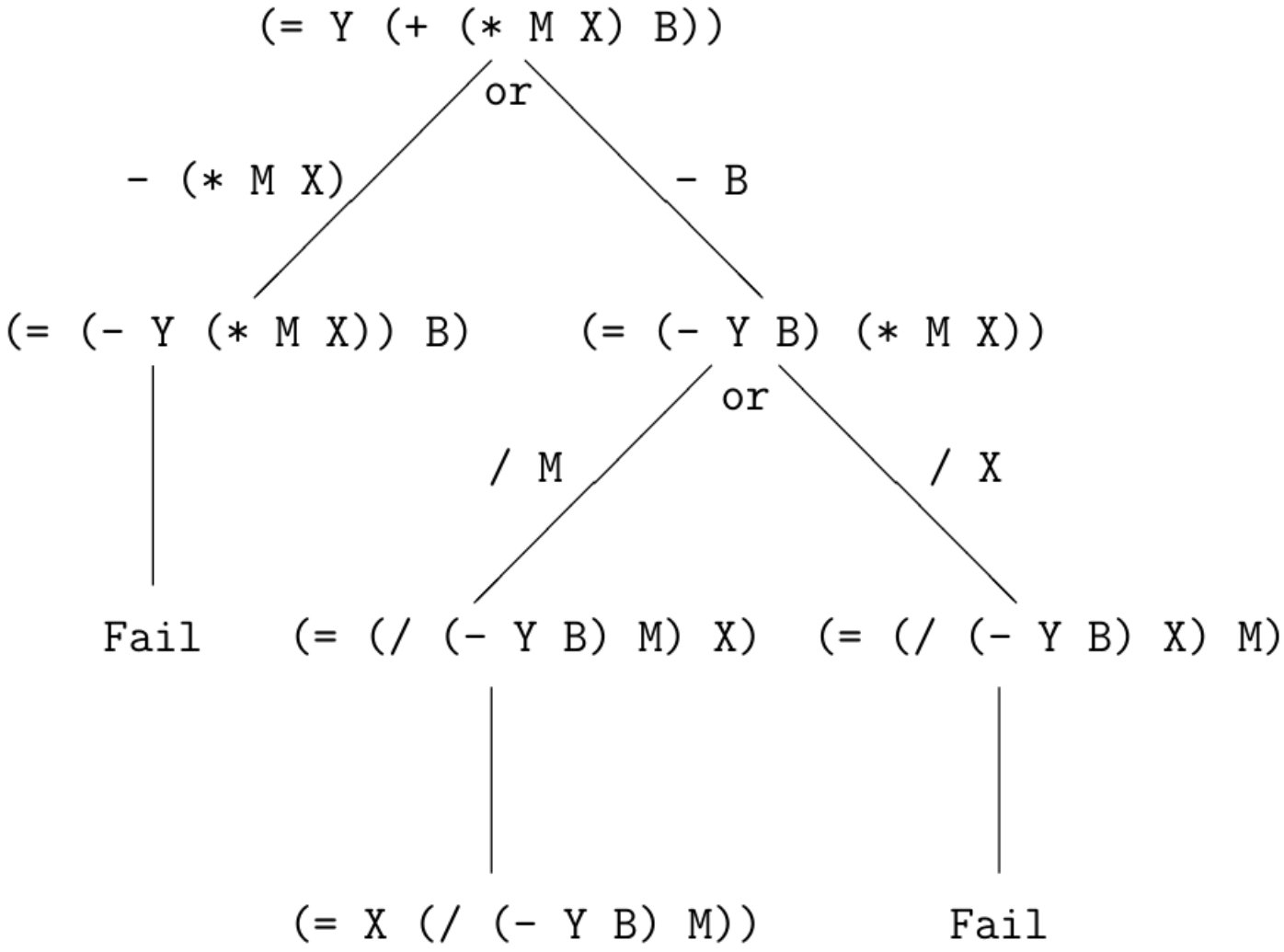
Recursive Tree Search

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.

```
>(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))
```

Recursive Tree Search Example

Goal: Solve for `X` given `(= Y (+ (* M X) B))`



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 `lhs` is only a single variable. Each recursive step makes the `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)))
  ```

- `eval`uate 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

Suppose that we have an association list of known variables and their values, and a list of equations:

```
values = ((m 2) (f 8))
```

- If the desired variable has a known value, return it.
- Try to find an equation where all variables are known except one: `(= f (* m a))`
- Solve the equation for that variable: `(= a (/ f m))`
- Evaluate the right-hand side of the solved equation using the values of the known variables (function `myevalb`) to give the value of the new variable. Add that variable to the binding list:

  ```
  values = ((a 4) (m 2) (f 8))
  ```

- 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
```

Generating Code from Equations

The first programming language, `FORTRAN`, is an abbreviation for `FOR`mula `TRAN`slation. 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
- `cons` the solved equation onto a list of code

If the variable that is solved for was the desired variable, `cons` a return statment onto the code.

At the end, reverse the code (`reverse`).

Eliminating Unused Equations

The opportunistic algorithm, computing whatever can be computed from the available values, may generate some equations that are valid but not needed for finding the desired value.

Optimizing compilers use two notions, *available* and *busy*.

- A value is *available* at a point p if it has been assigned a value above the point p in a program, e.g. as an argument of a subroutine or as the left-hand-side of an assignment statement.
- A value that is the left-hand-side of an assignment statement is *busy* or *live* if it will be used at a later point in the program.

We can eliminate equations whose lhs is not busy by proceeding backwards through the code:

- Initially, the desired value is busy.
- For each equation, if the lhs of the equation is a member of the busy list, keep the equation, and add its rhs variables to the busy list. Otherwise, the equation can be discarded.

Conservation Laws

- Common sense:

  There is no such thing as a free lunch.

  You can't have your cake and eat it too.

- Physics:

  Mass-energy is neither created nor destroyed.

  For every action, there is an equal and opposite reaction.

- Finance:

  Money is neither created nor destroyed.

  Every transaction requires an equal and opposite transaction.

Double-Entry Bookkeeping

Double-entry bookkeeping dates back some 1000 years, introduced in Korea and by Jewish bankers in Egypt, then in Italian banks during the Renaissance.

The basic idea is simple: money moves from one account to another; an addition to one account must correspond to a subtraction from another account.

For example, suppose you withdraw $100 in cash from your bank account at an ATM:

	Cash in Pocket	Bank Account
Get cash	+$100	-$100

Double-entry bookkeeping provides an *audit trail* that allows the flow of money to be followed.

Representing Financial Contracts

Modern financial contracts are more complex than an immediate subtraction from one account and addition to another account.

Wimpy:

> I'd gladly pay you Tuesday for a hamburger today.

Can we represent this formally?

```
(and (one hamburger)
     (give (zcb tuesday 5 USD)) )
```

`zcb` is a zero-coupon bond, i.e. a promise to pay $5 at a future time, Tuesday.

This kind of representation is recursive, allowing complex contracts to be represented with a small set of *combinators*.

Finance Combinators[8]

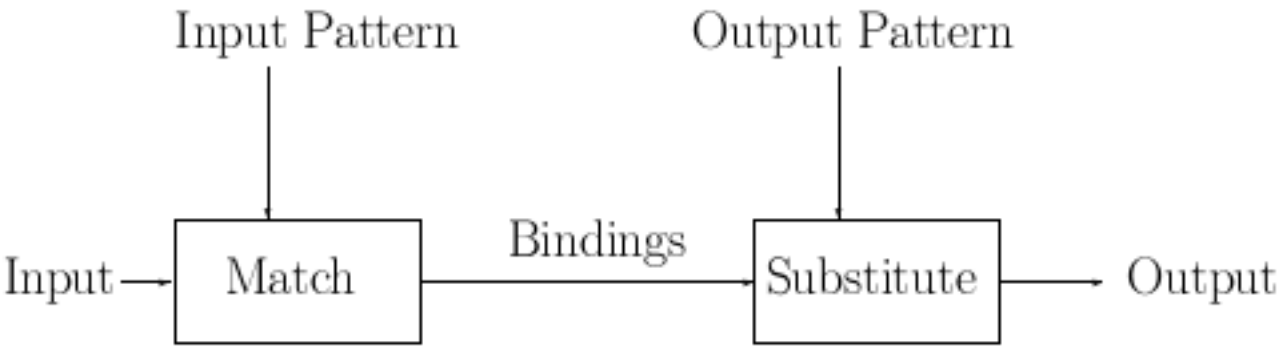
`(zero)`	a contract that has no rights and no obligations
`(one` k`)`	one unit of currency k
c	you immediately acquire contract c
`(give` c`)`	to give a contract c to another party; like negation
`(at` t c`)`	if you acquire contract c before time t, it becomes effective at time t
`(truncate` t c`)`	contract c ceases to exist after time t
`(and` c_1 c_2`)`	both contracts c_1 and c_2
`(or` c_1 c_2`)`	your choice of contracts c_1 and c_2
`(cond` b c_1 c_2`)`	you acquire contract c_1 if the observable b is true, else you acquire c_2
`(scale` o c`)`	multiply contract c by observable o
`(when` b c`)`	you must acquire contract c when b becomes true (but worthless if b can never be true)
`(anytime` b c`)`	you may acquire contract c any time b becomes true
`(until` b c`)`	is like contract c but must be abandoned when b becomes true

[8] S. Peyton Jones and J.M. Eber, "How to write a financial contract", in The Fun of Programming, ed Gibbons and de Moor, Palgrave Macmillan 2003

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 (cons? 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 `copy-tree` slightly, we can make a copy with a substitution:

```
; substitute new for old in form

(defn subst [new old form]
  (if (cons? form)
      (cons (subst new old (first form))
            (subst new old (rest form)))
      (if (= form old)
          new
          form) ) )
```

Substitution Examples

```
;               for      in
>(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))))
314.1592653589793
```

Loop Unrolling

Loop unrolling is the compile-time expansion of a loop into repetitions of the code, with the loop index replaced by its value in each instance.

```
for (i = 0; i < 3; i++)
  disp[i] = c2[i] - c1[i];
```

is expanded into:

```
disp[0] = c2[0] - c1[0];
disp[1] = c2[1] - c1[1];
disp[2] = c2[2] - c1[2];
```

	Loop:	Unrolled:
Instructions:	20	12
Executed:	57	12

The second form runs faster, and it may generate less code. This is a useful optimization when the size of the loop is known to be a small constant at compile time.

Loop Unrolling Code

The code to accomplish loop unrolling is simple:

```
(defn unrollb [code ivar n nmax codelst]
  (if (>= n nmax)
      (cons 'do (reverse codelst))
      (unrollb code ivar (+ n 1) nmax
        (cons (subst n ivar code) codelst) )))

(defn unroll [loopcode]
  (unrollb (third loopcode)            ; code
           (get (second loopcode) 0)   ; i
           0
           (get (second loopcode) 1)   ; nmax
           '()))

(unroll '(dotimes [i 3]
           (println (get arr i))) )

(do (println (get arr 0))
    (println (get arr 1))
    (println (get arr 2)))
```

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:

```
;  replace      by            by        in
>(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 (cons? form)
      (cons (sublis alist (first form))
            (sublis alist (rest form)))
      (let [binding (assocl form alist)]
          (if binding  ; (name value) or nil
              (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 (cons? 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 (cons? 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

```
(defn equal [x y]
  (if (cons? x)
      (and (cons? y)
           (equal (first x) (first y))
           (equal (rest x) (rest y)))
      (= x y) ))

>(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

```
>(equal '(+ a (* b c)) '(+ a (* b c)))

  1> (EQUAL (+ A (* B C)) (+ A (* B C)))
    2> (EQUAL + +)
    <2 (EQUAL T)
    2> (EQUAL (A (* B C)) (A (* B C)))
      3> (EQUAL A A)
      <3 (EQUAL T)
      3> (EQUAL ((* B C)) ((* B C)))
        4> (EQUAL (* B C) (* B C))
          5> (EQUAL * *)
          <5 (EQUAL T)
          5> (EQUAL (B C) (B C))
            6> (EQUAL B B)
            <6 (EQUAL T)
            6> (EQUAL (C) (C))
              7> (EQUAL C C)
              <7 (EQUAL T)
              7> (EQUAL NIL NIL)
              <7 (EQUAL T)
            <6 (EQUAL T)
          <5 (EQUAL T)
        <4 (EQUAL T)
        4> (EQUAL NIL NIL)
        <4 (EQUAL T)
      <3 (EQUAL T)
    <2 (EQUAL T)
  <1 (EQUAL T)
T
```

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. 48.

```
(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 (cons? tree)
      (nnumsb (rest tree)
              (nnumsb (first tree) answer))
      (if (number? tree)
          (+ 1 answer)
          answer) ) )
(defn nnums [tree] (nnumsb tree 0))
```

Tracing Nested Tree Recursion

```
>(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)
    ...
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 (cons? tree)
              (?combine (?fun (first tree))
                        (?fun (rest tree)))
              (if (?test tree) ?trueval ?falseval))

       '(defn nnums [tree]
          (if (cons? tree)
              (+ (nnums (first tree))
                 (nnums (rest tree)))
              (if (number? tree) 1 0))) )

((?falseval 0) (?trueval 1) (?test number?)
 (?combine +) (?fun nnums) (t t))
```

Specifications of Match

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

Match Function

```
(defn equal [x y]
  (if (cons? x)
      (and (cons? y)
           (equal (first x) (first y))
           (equal (rest x) (rest y)))
      (= x y) ))

(defn matchb [pat inp bindings]
(if (cons? bindings)   ; if not, already failed
    (if (cons? pat)         ; if pat is a cons
        (and (cons? inp)   ; inp must be a cons
             (matchb (rest pat)  ; parts must match
                     (rest inp)
                     (matchb (first pat)
                             (first inp) bindings))
        (if (varp pat)     ; not a cons: a var?
            (if (assocl pat bindings)
                (and (equal inp  ; existing binding
                      (second (assocl pat bindings)
                     bindings)
                (cons (list pat inp) bindings))
            (and (= pat inp) bindings)))) )

(defn match [pat inp]  (matchb pat inp '((t t))))
```

Matching and Substitution

`match` and `sublis` are inverse operations:

- (`match` *pattern instance*) = *bindings*
- (`sublis` *bindings pattern*) = *instance*

If we use a transformed pattern in the second equation, we can form a rule that transforms instances:

- (`match` *pattern instance*) = *bindings*
- (`sublis` *bindings newpattern*) = *newinstance*
- *transformationrule*: *pattern* $\rightarrow$ *newpattern*
- (`transform` *rule instance*)
  = (`sublis (match` *pattern instance*) *newpattern*)

Transformation by Patterns

Matching and substitution can be combined to *transform* an input using a transformation rule `transrule`: a list of an input pattern and an output pattern.

```
(defn transform [transrule input]
  (let [bindings (match (first transrule)
                        input)]
    (if bindings
        (sublis bindings (second transrule)) ) ))

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

            '(I aint got no bananas) )

(I do not have any bananas)
```

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).

Symbolic Differentiation

Symbolic differentiation is easy to do with patterns because there is a list of reduction patterns in calculus books:

$$d/dx(u+v) = d/dx(u) + d/dx(v)$$
$$d/dx(u*v) = v*d/dx(u) + u*d/dx(v)$$

```
( (deriv (+ ?u ?v) ?x)  (+ (deriv ?u ?x)
                           (deriv ?v ?x)) )
( (deriv (* ?u ?v) ?x)  (+ (* ?v (deriv ?u ?x))
                           (* ?u (deriv ?v ?x))) )
```

These formulas have the properties:

- the formulas are recursive
- the formula whose derivative is being taken gets smaller at each step.

These features guarantee that the process of taking a derivative must terminate in a finite number of steps.

Repetitive Transformation

One transformation may expose another opportunity for transformation:

```
(+ (* x 0) y)
(+ 0 y)
y
```

An easy way to handle this is to walk through the tree, transforming what can be done, until no further transformations are possible; this is called a *fixed point* or *fixpoint*.

The function `transformfp` transforms an expression repeatedly until it reaches a fixpoint.

Optimization by Patterns

```
(def optpatterns
  '( ((+ ?x 0)          ?x )
     ((* ?x 0)          0 )
     ((* ?x 1)          ?x )
     ((- (- ?x ?y))     (- ?y ?x) )
     ((- 1 1)           0 )
     ... ))
```

While humans are unlikely to write code such as $x + 0$, symbolic computations such as symbolic differentiation and automatic programming by substitution into design patterns can often generate such expressions.

```
deriv:  (deriv2 (+ (expt x 2) (+ (* 3 x) 6)) x)

  der:  (+ (+ (* 2 (+ (* (expt x (- 2 1)) 0)
        (* 1 (* (- 2 1) (* (expt x (- (- 2 1)
        1)) 1))))) (* (* (expt x (- 2 1)) 1) 0))
        (+ (+ (+ (* 3 0) (* 1 0))
        (+ (* x 0) (* 0 1))) 0))

  opt:  2
```

Constant Folding

Constant folding is performing operations on constants at compile time:

```
(/ (* angle 3.1415926) 180.0)
```

```
(sqrt 2.0)
```

The savings from doing this on programmer expressions is minor. However, there can be savings by optimizing the results of program manipulations.

Constant folding must be used with care: operators that have *side effects* should not be folded.

```
>(println "foo")
foo
nil
```

We do not want to replace this print with `nil`.

Correctness of Transformations

It is not always easy to be certain that transformed code will give *exactly* the same results.

```
( (> (* ?n ?x)
     (* ?n ?y))          (> ?x ?y) )

( (not (not ?x))         ?x )

( (= (if ?p ?qu ?qv)
     ?qu)                ?p )
```

These transformations are *usually* correct, but it is possible to construct an example for each in which the transformation changes the result. We must be careful to use only correct transforms.

Knuth-Bendix Algorithm

The Knuth-Bendix algorithm[9] describes how to derive a complete set of rewrite rules R from an equational theory E, such that:

> If E implies that two terms s and t are equal, then the reductions in R will rewrite both s and t to the same irreducible form in a finite number of steps.

Two properties are needed:

- *Confluence*: no matter what sequence of transforms is chosen, the final result is the same.
- *Termination*: the process of applying transforms will terminate.

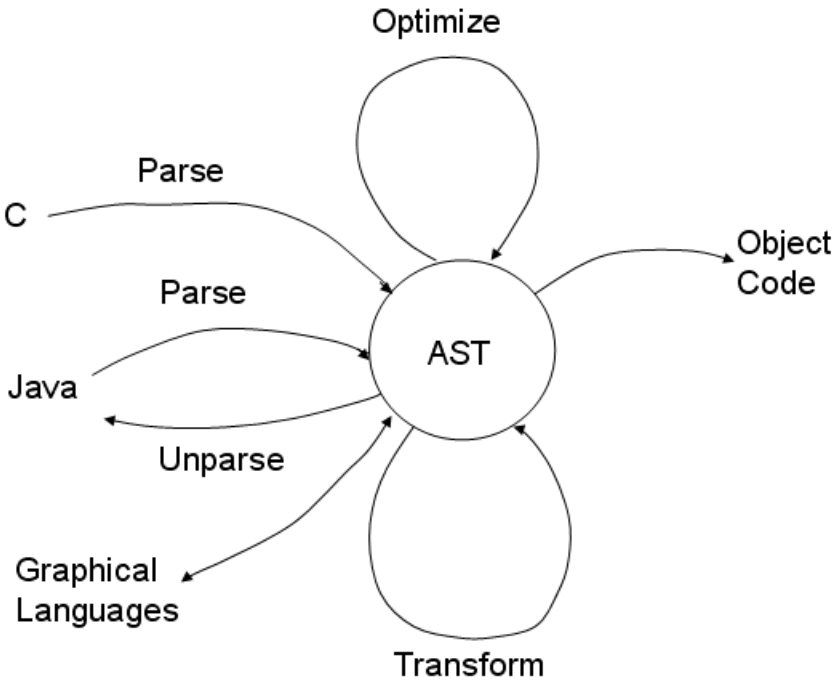
The Knuth-Bendix algorithm is based on a well-founded ordering of terms so that each rewriting step makes the result "smaller".

Unfortunately, rather simple systems do not have a Knuth-Bendix solution.

[9] Knuth, D. E and Bendix, P. E., "Simple word problems in universal algebras", in J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergammon Press, 1970, pp. 263-297.

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.



Macros

A *macro* is a function from code to code, usually turning a short piece of code into a longer code sequence.

Lisp macros produce Lisp code as output; this code is executed or compiled.

```
(defn neq [x y] (not (= x y)))  ; like !=

(defmacro neq [x y] (list 'not (list '= x y)))

> (neq 2 3)
true
> (macroexpand '(neq 2 3))
(not (= 2 3))
```

If a macro uses its own variables, it is important to generate new ones with `gensym` to avoid *variable capture* or name conflicts with calling code.

```
> (gensym 'foo)
foo2382
```

In-line Compilation

In-line or *open* compilation refers to compile-time expansion of a subprogram, with substitution of arguments, in-line at the point of each call.

Advantages:

- Eliminates overhead of procedure call
- Can eliminate method lookup in an object-oriented system
- Can expose opportunities for optimization across the procedure call, especially with OOP: more specific types become exposed.
- Relative saving is high for small procedures

Disadvantages:

- May increase code size

Partial Evaluation

Partial evaluation is the technique of evaluating those parts of a program that can be evaluated at compile time, rather than waiting for execution time.

For example, the rotation of a point in homogeneous coordinates by an angle θ around the x axis is accomplished by multiplying by the matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Many of the cycles consumed in the matrix multiply would be wasted because they would be trivial computations (*e.g.*, multiplying by `1` or adding `0`).

By unrolling the loops of matrix multiply, substituting the values from the coefficient matrix, and performing partial evaluation on the result, a specialized version of the matrix multiply can be obtained. This version saves many operations:

Version:	Load	Store	Add/Sub	Mul	Total
General	128	16	48	64	256
Specialized	24	16	8	16	64

Partial Evaluation[10]

Partial evaluation specializes a function with respect to arguments that have known values. Given a program $P(x, y)$ where the values of variables x are constant, a specializing function `mix` transforms $P(x, y) \rightarrow P_x(y)$ such that $P(x, y) = P_x(y)$ for all inputs y. $P_x(y)$ may be shorter and faster than $P(x, y)$. We call x *static data* and y *dynamic data*.

Partial evaluation involves:

- *precomputing* constant expressions involving x,
- *propagating* constant values,
- *unfolding* or *specializing* recursive calls,
- *reducing* symbolic expressions such as $x * 1$, $x * 0$, $x + 0$, $(if\ true\ S_1\ S_2)$.

A good rule of thumb is that an interpreted program takes ten times as long to execute as the equivalent compiled program. Partial evaluation removes interpretation by increasing the *binding* between a program and its execution environment.

[10]Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993; *ACM Computing Surveys*, vol. 28, no. 3 (Sept. 1996), pp. 480-503.

Example

Suppose we have the following definition of a function `power(x,n)` that computes x^n :

```
(defun power (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (power x (/ n 2)))
          (* x (power x (- n 1))))))
```

If this is used with a constant argument `n`, as is often the case, the function can be partially evaluated into more efficient code:

```
(gldefun t3 ((x real)) (power x 5))

(LAMBDA (X) (* X (SQUARE (SQUARE X))))
```

The recursive function calls and interpretation (`if` statements) have been completely removed; only computation remains. Note that the constant argument `5` is gone and has been converted into control.

Simple Partial Evaluator

```
(defun mix (code env)
  (let (args test fn)
    (if (constantp code)              ; a constant
        code                          ;   evaluates to itself
        (if (symbolp code)            ; a variable
            (if (assoc code env)      ;   bound to a constant
                (cdr (assoc code env)) ; evals to that consta
                code)                  ; else to itself
            (if (consp code)
                (progn
                  (setq fn (car code))
                  (if (eq fn 'if)          ; if is handled
                      (progn               ;     specially
                        (setq test (mix (cadr code) env))
                        (if (eq test t)             ; if true
                            (mix (caddr code) env) ; then par
                            (if (eq test nil)      ; if false
                                (mix (cadddr code) env) ; els
                                (cons 'if
                                  (cons test
                                    (mapcar #'(lambda (x)
                                                (mix x env))
                                            (cddr code))))))))
```

Simple Partial Evaluator...

```
      (progn                                  ; (fn args)
        (setq args (mapcar #'(lambda (x)
                               (mix x env)) ; mix the args
                           (cdr code)))
        (if (and (every #'constantp args)   ; if all constant
                 (not (member fn '(print    ; and no
                        prin1 princ error   ; compile-time
                        format))))          ; side-effects
            (kwote (eval (cons fn args)))   ; eval it now
            (if (and (some #'constantp args); if some constan
                     (fndef fn))            ;  & symbolic fn
                (fnmix fn args)             ;  unfold the fn
                (fnopt (cons fn args)))))))) ; optimize result

(cons 'bad-code code)) ) ) ))
```

Examples

```
>(load "/u/novak/cs394p/mix.lsp")
>(mix 'x '((x . 4)))
4
>(mix '(if (> x 2) 'more 'less) '((x . 4)))
'MORE

(defun power (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (power x (/ n 2)))
          (* x (power x (- n 1)))) ) )

>(fnmix 'power '(x 3))
(* X (SQUARE X))

>(specialize 'power '(x 3) 'cube)
>(fndef 'cube)
(LAMBDA (X) (* X (SQUARE X)))
> (cube 4)
64

>(fnmix 'power '(x 22))
(SQUARE (* X (SQUARE (* X (SQUARE (SQUARE X))))))
```

Examples

```
; append two lists
(defun append1 (l m)
  (if (null l)
      m
      (cons (first l) (append1 (rest l) m))))


>(fnmix 'append1 '('(1 2 3) m))
(CONS 1 (CONS 2 (CONS 3 M)))
```

Binding-Time Analysis

Binding-time analysis determines whether each variable is static (S) or dynamic (D).

- Static inputs are S and dynamic inputs are D.
- Local variables are initialized to S.
- Dynamic is contagious: if there is a statement
  $v = f(...D...)$
  then v becomes D.
- Repeat until no more changes occur.

Binding-time analysis can be *online* (done while specialization proceeds) or *offline* (done as a separate preprocessing phase). Offline processing can *annotate* the code by changing function names to reflect whether they are static or dynamic, e.g. `if` becomes `ifs` or `ifd`.

Futamura Projections[11]

Partial evaluation is a powerful unifying technique that describes many operations in computer science.

We use the notation $[\![\mathtt{P}]\!]_{\mathtt{L}}$ to denote running a program `P` in language `L`. Suppose that `int` is an interpreter for a language `S` and `source` is a program written in `S`. Then:

- $$\begin{aligned} \mathtt{output} &= [\![\mathtt{source}]\!]_{\mathtt{s}}[\mathtt{input}] \\ &= [\![\mathtt{int}]\!][\mathtt{source}, \mathtt{input}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]]\!][\mathtt{input}] \\ &= [\![\mathtt{target}]\!][\mathtt{input}] \end{aligned}$$

  Therefore, $\mathtt{target} = [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]$.

- $$\begin{aligned} \mathtt{target} &= [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}]]\!][\mathtt{source}] \\ &= [\![\mathtt{compiler}]\!][\mathtt{source}] \end{aligned}$$

  Thus, $\mathtt{compiler} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}] = [\![\mathtt{cogen}]\!][\mathtt{int}]$

- Finally, $\mathtt{cogen} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{mix}] = [\![\mathtt{cogen}]\!][\mathtt{mix}]$ is a *compiler generator*, i.e., a program that transforms interpreters into compilers.

[11] Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler", *Systems, Computers, Controls*, 2(5):45-50, 1971. The presentation here follows Jones *et al.*

Interpreter

This program is an interpreter for arithmetic expressions using a simulated stack machine.

```
(defun topinterp (exp)   ; interpret, pop result
  (progn (interp exp)
         (pop *stack*)))

(defun interp (exp)
  (if (consp exp)                ; if op
      (if (eq (op exp) '+)
          (progn (interp (lhs exp))  ; lhs
                 (interp (rhs exp)) ; rhs
                 (plus))              ; add
          (if ...))      ; other ops
      (pushopnd exp)))   ; operand

(defun pushopnd (arg) (push arg *stack*))

(defun plus ()
  (let ((rhs (pop *stack*)))
    (pushopnd (+ (pop *stack*) rhs))))

>(topinterp '(+ (* 3 4) 5))
17
```

Specialization

The interpreter can be specialized for a given input expression, which has the effect of compiling that expression.

```
>(topinterp '(+ (* 3 4) 5))
17

>(specialize 'topinterp
             '('(+ (* a b) c))
             'expr1 '(a b c))

>(pp expr1)

(LAMBDA-BLOCK EXPR1 (A B C)
  (PROGN
    (PUSH A *STACK*)
    (PUSH B *STACK*)
    (TIMES)
    (PUSH C *STACK*)
    (PLUS)
    (POP *STACK*)))

>(expr1 3 4 5)
17
```

Parameterized Programs

A highly parameterized program is easier to write and maintain than many specialized versions for different applications, but may be inefficient.

Example: Draw a line: (x_1, y_1) to (x_2, y_2).
Options include:

- Width of line (usually 1)
- Color
- Style (solid, dashed, etc.)
- Ends (square, beveled)

If all of these options are expressed as parameters, it makes code longer, makes calling sequences longer, and requires interpretation at runtime. Partial evaluation can produce efficient specialized versions automatically.

Pitfalls of Partial Evaluation

There are practical difficulties with partial evaluation:

- To be successfully partially evaluated, a program must be written in the right way. There should be good *binding time separation*: avoid mixing static and dynamic data (which makes the result dynamic).

```
(lambda (x y z)
  (+ (+ x y) z))
```

```
(lambda (x y z)
  (+ x (+ y z)))
```

- The user may have to give advice on when to unfold recursive calls. Otherwise, it is possible to generate large or infinite programs.

  One way to avoid this is to require that recursively unfolding a function call must make a constant argument smaller according to a well-founded ordering. Branches of dynamic `if` statements should not be unfolded.

Pitfalls ...

- Repeating arguments can cause exponential computation duplication: [12]

```
(defun f (n)
  (if (= n 0)
      1
      (g (f (- n 1)) ) ) )

(defun g (m) (+ m m))
```

- The user should not have to understand the logic of the output program, nor understand how the partial evaluator works.
- Speedup of partial evaluation should be predictable.
- Partial evaluation should deal with typed languages and with symbolic facts, not just constants.

[12] Jones *et al.*, p. 119.

Language Translation

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;
    }
```

Max and Min of a Function

A minimum or maximum value of a function occurs where the derivative of the function is zero (i.e. on a graph of the function, the line will be horizontal).

We can derive a symbolic expression for the min/max or a function as follows:

1. Find the derivative of the rhs of the equation with respect to the independent variable.
2. Make a new equation, setting the derivative to zero.
3. Solve the new equation for the independent variable.
4. Simplify the rhs of the equation.

```
(def cannonball
     '(= y (- (* (* v (sin theta)) t)
              (* (/ g 2) (expt t 2)))) )
```

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.

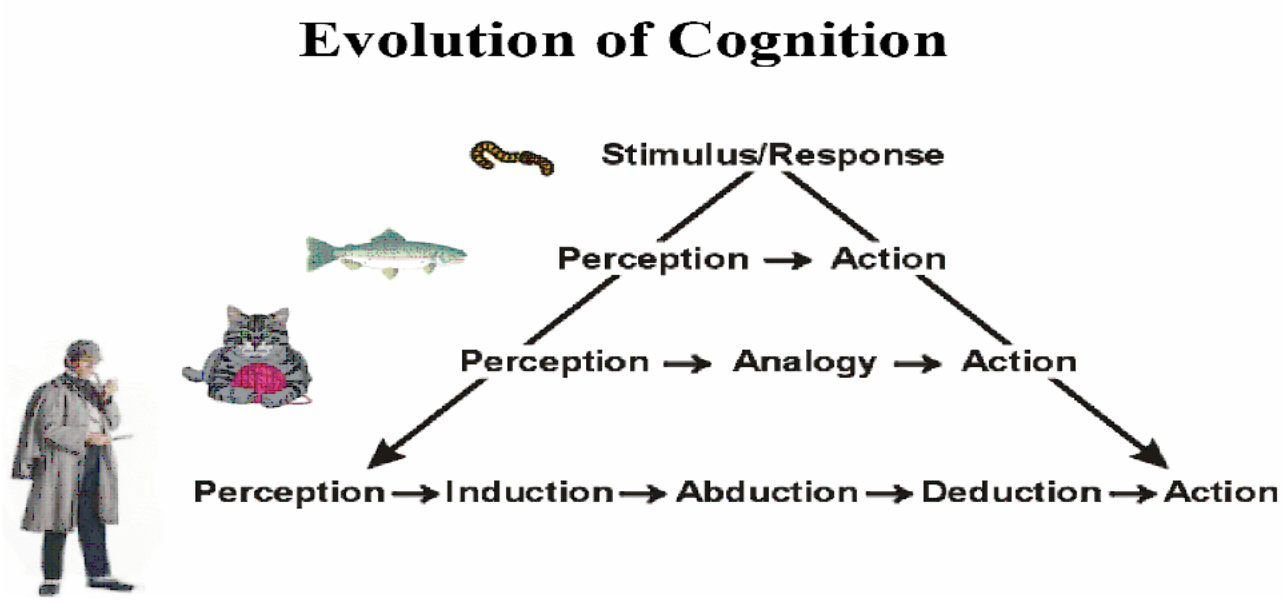
The storage and reasoning mechanisms are usually closely coupled.

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[13]
- *reasoning* by processing symbol structures, resulting in other symbol structures.

A central problem of A.I. is to understand the symbolic representations and reasoning processes.[14]



[13]Newell, A., Physical Symbol Systems, *Cognitive Science*, 1980, *4*, 135-183.
[14]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).[15]

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*.

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

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):

	Math	C		meaning
Negation	$\neg$ or $\sim$	`!`		not
Conjunction	$\wedge$ ("wedge")	`&&`		and
Disjunction	$\vee$ ("vee")	`||`		or
Implication	$\rightarrow$ or $\supset$	`?`	`:`	implies, if-then
	$\leftrightarrow$	`==`		iff (if and only if)
				$\leftarrow \wedge \rightarrow$

- Constants: $True$ or filled-in 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: $P \vee \neg P$. 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*: $P \wedge \neg P$.

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:
  $F \to G = \neg F \vee G$
  $F \leftrightarrow G = (F \to G) \wedge (G \to F)$
  $A \wedge B \wedge C \to D = \neg A \vee \neg B \vee \neg C \vee D$

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

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

Inference Rules

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

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.

Rules for Backward Chaining

Backward chaining assumes rules of the form:

$$A \wedge B \rightarrow C$$

Such a rule is called a *Horn clause*; a Horn clause has at most one positive literal when written in Conjunctive Normal Form as a disjunction (or) of literals:

$$A \wedge B \rightarrow C$$

$$(A \wedge B) \rightarrow C$$

$$\neg(A \wedge B) \vee C$$

$$\neg A \vee \neg B \vee C$$

Backward Chaining

Suppose that we have formulas:
A
B
D
$A \wedge B \rightarrow C \quad$ or $\quad C \leftarrow A \wedge B$
$C \wedge D \rightarrow E \quad$ or $\quad E \leftarrow C \wedge D$

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 as its 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.

Backchaining

Backchaining is easily implemented as a recursive tree search (file `backch.clj`).

The function `(backchain goal rules facts)` tries to prove a goal given a set of rules and facts.

`goal` is a symbol (atom or propositional variable).

`facts` is a list of atoms that are known to be true.

`rules` is a list of rules of the form `(`*conclusion* $prem_1$ ... $prem_n$`)`; each rule states that the conclusion is true if all of the premises are true.

For example, the rule $A \wedge B \rightarrow C$ would be written $C \leftarrow A \wedge B$ or `(c a b)`, similar to Prolog, `c :- a,b` .

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 backchain [goal rules facts]  ; true if
 (or (member goal facts)    ; goal is known fact
     (some             ; or there is some rule
       (fn [rule]      ;  that concludes
         (and (= (first rule) goal) ;  goal
              (every?  ; and every premise
                (fn [premise]  ; can be proved
                  (backchain premise rules facts))
                (rest rule))))
       rules)) )
```

```
>(backchain 'e '((c a b) (e c d)) '(a b d))
true
```

This form of backchaining is useful when there are relatively few rules but many facts, e.g. stored in a separate database.

(backchain goal rules facts)

```
user=> (backchain 'e '((c a b) (e c d)) '(a b d))
TRACE t254: (backchain e ((c a b) (e c d)) (a b d))
TRACE t255: | (member e (a b d))
TRACE t255: | => nil
            ; trying rule (e c d): e if c and d
TRACE t259: | (backchain c ((c a b) (e c d)) (a b d))
TRACE t260: | | (member c (a b d))
TRACE t260: | | => nil
              ; trying rule (c a b): c if a and b
TRACE t264: | | (backchain a ((c a b) (e c d)) (a b d))
TRACE t265: | | | (member a (a b d))
TRACE t265: | | | => (a b d)
TRACE t264: | | => (a b d)
TRACE t266: | | (backchain b ((c a b) (e c d)) (a b d))
TRACE t267: | | | (member b (a b d))
TRACE t267: | | | => (b d)
TRACE t266: | | => (b d)
TRACE t259: | => true
TRACE t269: | (backchain d ((c a b) (e c d)) (a b d))
TRACE t270: | | (member d (a b d))
TRACE t270: | | => (d)
TRACE t269: | => (d)
TRACE t254: => true
true
```

Backchaining Code, version 2

We can add facts to our list of clauses by making a fact a premise clause with no antecedents; this is the form used in Prolog. Since the premise list is empty, `every?` returns `true`.

```
(defn backch [goal]  ; goal is true
  (some (fn [clause] ; if there is some clause
          (and (= goal (first clause))
                       ; that concludes goal
               (every? backch (rest clause))))
                       ; and every premise is true
        clauses))

(def clauses '((a) (b) (d) (c a b) (e c d)) )

user=> (backch 'e)
true
```

Fact = Rule with No Premises

```
(def clauses '((a) (b) (d) (c a b) (e c d)) )

user=> (backch 'e)
TRACE t290: (backch e)
TRACE t291: | (backch c)
TRACE t292: | | (backch a)
TRACE t292: | | => true
TRACE t293: | | (backch b)
TRACE t293: | | => true
TRACE t291: | => true
TRACE t294: | (backch d)
TRACE t294: | => true
TRACE t290: => true
true
```

Normal Forms

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

A formula F is in *conjunctive normal form* (CNF) if F is of the form $F = F_1 \wedge F_2 \wedge ... \wedge F_n$ where each F_i is a clause, i.e. a disjunction ($\vee$, OR) of literals.

Example:

$$\begin{array}{rrl} & (\neg P \vee Q) & \text{clause 1} \\ \wedge & (P) & \text{clause 2} \\ \wedge & (\neg Q) & \text{clause 3} \end{array}$$

CNF is used for resolution and for SAT solvers. There is also a *disjunctive normal form*, less often used.

Satisfiability Checking (Model Checking)

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

Suppose that there is an election, with candidates Alice and Bob. One of them will win, but they cannot both win. We can express this as two clauses in CNF, assumed to be ANDed together:

$$(A \vee B)$$

$$\wedge(\neg A \vee \neg B)$$

There are two interpretations (models) that satisfy all of the clauses:

$$A, \neg B$$

$$\neg A, B$$

A special case of SAT is 3SAT, where all clauses have at most 3 literals, corresponding to rules that have two premises and a single conclusion. For example, $A \wedge B \rightarrow C$ becomes $\neg A \vee \neg B \vee C$ in CNF.

SAT Solvers

There is a trivial algorithm for satisfiability checking: for each possible interpretation, check whether the conjunction of clauses is satisfied. The problem is that this is $O(2^n)$ when there are n literals.

However, there are some efficient algorithms:

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

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

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

Uses of SAT Solvers

There are practical uses of SAT solvers:

- Constraint satisfaction problems, e.g. FPGA routing.
- Circuit checking: two Boolean functions f_1 and f_2 are equal iff $(f_1 \vee f_2) \wedge (\neg f_1 \vee \neg f_2)$ is unsatisfiable.
- Safety checking: show that an instance where two trains are going in opposite directions on the same track is not satisfiable.
- Show that a request will eventually be answered; this may be approximated by *unrolling* it into k time steps.
- Show that a case where a distributed memory system will give the wrong value for a memory request is unsatisfiable.

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. A constant is equivalent to a function of no arguments.
  - Functions: $f(x)$ whose arguments are terms.
- Quantifiers: $\forall$ ("for all") (*cf.* `every`) and $\exists$ ("there exists" or "for some") (*cf.* `some`) quantify variables: $\forall x$, $\exists y$. If a variable is in the scope of a quantifier, it is *bound*; otherwise, it is *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:

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

   and Skolemized:

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

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

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

   and Skolemized:

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

Skolemization

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

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

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

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

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

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

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 $C \leftarrow A \wedge B$, 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*.

Unification Algorithm

Given two predicates that initially have no (universally quantified) variables in common, a unification algorithm should:

- Find a substitution of terms for variables that will make the two predicates identical, or
- Report that no such substitution exists:
  the predicates do not unify.

```
user=> (unify '(p  x   (a))
              '(p (b)   y )   )

( (y (a))   (x (b))   (t t))

user=> (unify '(q (a))
              '(q (b)))
nil
```

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 function `sublis`. `(sublis` *alist form* `)` performs the substitutions specified by *alist* in the formula *form*. *alist* is of the form `((` *var term* `) ... )`.

Suppose we want to substitute $\{a/x, f(b)/y\}$ in $P(x, y)$. As a call to `sublis`, this is:

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

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

Unification Code

```
(defn unify [u v] (unifyb u v '((t t))))

; unify terms: subst list, nil if failure.
(defn unifyb [u v subs]      ; unification works if
(and subs
     (or (and (= u v) subs)      ; identical vars
         (varunify v u subs)     ; u is a var
         (varunify u v subs)     ; v is a var
         (and (cons? u) (cons? v) ; functions
              (= (first u) (first v)) ; same name
              (unifyc (rest u) (rest v) subs)) ) )
                                  ; and args unify

; unify variable and term if possible
; adds (var term) to subs, or nil
(defn varunify [term var subs]
  (and var subs (symbol? var)
       (not (occurs var term))
       (cons (list var term)
             (subst term var subs))))
```

Unification Code ...

```
; unify lists of arguments
; lists must be of same length
(defn unifyc [args1 args2 subs]
  (if (empty? args1)
      (if (empty? args2)
          subs)   ; return subs, else fail
      (and args2 subs
       (let [newsubs (unifyb (first args1)
                             (first args2) subs)]
         (if newsubs
             (unifyc (sublis newsubs (rest args1))
                     (sublis newsubs (rest args2))
                     newsubs))) ) ) )
```

Unification Examples

```
user=> (unify '(p x) '(p (a)))
((x (a)) (t t))

user=> (unify '(p x     (g x) (g (b)))
              '(p (f y) z     y))
((y (g (b)))
 (z (g (f (g (b)))))
 (x (f (g (b))))        (t t))

user=> (unify '(p x (f x)) '(p (f y) y))
nil
```

Soundness and Completeness

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

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

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

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

Resolution for predicate calculus is:

- *sound:* If $\square$ 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.

Resolution

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

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

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

This assumes that the premises are consistent.

Conjunctive Normal Form

For a resolution program, we want to eliminate as much of the logic notation as possible. This is done in the following ways:

- Universal $\forall$ quantifiers are eliminated by assuming that any variable is universally quantified: `x` .
- Existential $\exists$ quantifiers are eliminated by Skolemizing and turning existential variables into constants, which are functions of no arguments: `(b)` .
- Predicates in a clause are assumed to be connected by $\vee$.
- Clauses are assumed to be connected by $\wedge$.
- The only operator remaining is `not`
- The conclusion is negated and added to the set of clauses.

A clause such as "all hounds howl" becomes:
$\forall x Hound(x) \rightarrow Howl(x)$
$\neg Hound(x) \vee Howl(x)$
`( (not (hound x)) (howl x) )`

Resolution Example

1. All hounds howl at night.
2. Anyone who has any cats will not have any mice.
3. Light sleepers do not have anything which howls at night.
4. John has either a cat or a hound.
5. (Conclusion) If John is a light sleeper, then John does not have any mice.

Resolution Example

```
>(linres *hounds* *houndc*)
              1.  ((NOT (HOUND X)) (HOWL X))
              2.  ((NOT (LS X)) (NOT (HAVE X Y))
                   (NOT (HOWL Y)))
              3.  ((NOT (HAVE X Y)) (NOT (CAT Y))
                   (NOT (HAVE X Z)) (NOT (MOUSE Z)))
              4.  ((HAVE (JOHN) (A)))
              5.  ((CAT (A)) (HOUND (A)))
              6.  ((LS (JOHN)))
              7.  ((HAVE (JOHN) (B)))
              8.  ((MOUSE (B)))
(  6,  2):    9.  ((NOT (HAVE (JOHN) Y)) (NOT (HOWL
(  9,  1):   10.  ((NOT (HAVE (JOHN) X)) (NOT (HOUND
( 10,  4):   11.  ((NOT (HOUND (A))))
( 11,  5):   12.  ((CAT (A)))
( 12,  3):   13.  ((NOT (HAVE X (A))) (NOT (HAVE X Z
                   (NOT (MOUSE Z)))
( 13,  4):   14.  ((NOT (HAVE (JOHN) Z)) (NOT (MOUSE
( 13,  4):   15.  ((NOT (HAVE (JOHN) (A)))
                   (NOT (MOUSE (A))))
( 14,  4):   16.  ((NOT (MOUSE (A))))
( 14,  7):   17.  ((NOT (MOUSE (B))))
( 17,  8):   18.  NIL
PROVED
```

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 integer is odd or even" becomes:
$Integer(x) \rightarrow Odd(x) \vee Even(x)$
$\neg Integer(x) \vee Odd(x) \vee Even(x)$
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 Odd) 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.  ((FATHER (ZEUS) (ARES)))
   2.  ((MOTHER (HERA) (ARES)))
   3.  ((FATHER (ARES) (HARMONIA)))
   4.  ((PARENT X Y) (MOTHER X Y))
   5.  ((PARENT X Y) (FATHER X Y))
   6.  ((GRANDPARENT X Y) (PARENT Z Y) (PARENT X Z))

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

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

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

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

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

Deductive Composition of Astronomical Software from Subroutine Libraries [16] [17]

Amphion: Compose programs from a subroutine library, based on a graphical specification, using deduction.

SPICE: subroutine library for solar-system geometry.

- Various systems of time: ephemeris time, spacecraft clock time, etc.
- Various frames of reference
- Light does not travel instantaneously over astronomical distances

Example task: observe the position of a moon of a nearby planet to determine position of the spacecraft.

[16] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood: "Deductive Composition of Astronomical Software from Subroutine Libraries", *Proc. 12th Int. Conf on Automated Deduction (CADE'94)*, Nancy (France), June 994, *LNAI 814*, Springer Verlag, pp. 341-355.

[17] Steve Roach and Jeffrey Van Baalen, "Experience Report on Automated Procedure Construction for Deductive Synthesis", *Proc. Automated Software Engineering Conf.*, Sept. 2002, pp. 69-78.

Difficulty of Programming

- Subroutines may not be well documented
- User must understand documentation
- Many subroutines: takes time to become familiar with the collection
- User may rewrite subroutine rather than reusing it
- User might make mistakes, e.g. wrong type of units of argument

Constructive proof: given a theorem $\forall x \exists y \ P(x, y)$, prove it by constructing a y that satisfies the theorem.

During program synthesis, *witnesses* are constructed for existential terms. The witnesses correspond to subroutines in the SPICE library (*concrete* terms).

$specification \rightarrow theorem \rightarrow proof \rightarrow program$

Domain Theory

A *domain theory* provides a logical language for the application domain:

- Time: *time* is *abstract*, but has several *concrete* representations (ephemeris time, UTC, spacecraft clock time)
- Points, rays, planes
- Photon travel
- Events (space-time points)
- Celestial bodies, e.g. Saturn
- Axioms that relate abstract and concrete terms.

$$\forall tc\ (=\ (absctt\ UTC\ tc)\ (absctt\ Ephemeris\ (UTC2Ephemeris\ tc)))$$

$$\forall tc\ (=\ (absctt\ Ephemeris\ tc)\ (absctt\ UTC\ (Ephemeris2UTC\ tc)))$$

absctt abstracts from a time system and time coordinate to an abstract time. These axioms specify what the conversion functions such as $Ephemeris2UTC$ do.

Representation conversions are combinatorially explosive because they can loop.

Astronomical Domain

About 200 axioms are used to describe the domain of astronomy.

- *lightlike?*(e_1, e_2) holds if a photon could leave the event (position and time) e_1 and arrive at event e_2.
- *ephemeris-object-and-time-to-event* yields an event corresponding to the position of a given astronomical object (planet or spacecraft) at a given time.
- *a-sent*(o, d, ta) computes the time a photon must leave object o to arrive at destination d at time ta.
- Axiom *lightlike?-of-a-sent*:

```
(all (o d ta)
  (lightlike?
    (ephemeris-object-and-time-to-event
      o (a-sent o d ta))
    (ephemeris-object-and-time-to-event d ta)))
```

  - `o` = origin
  - `d` = destination
  - `ta` = time of arrival

Problem Difficulty

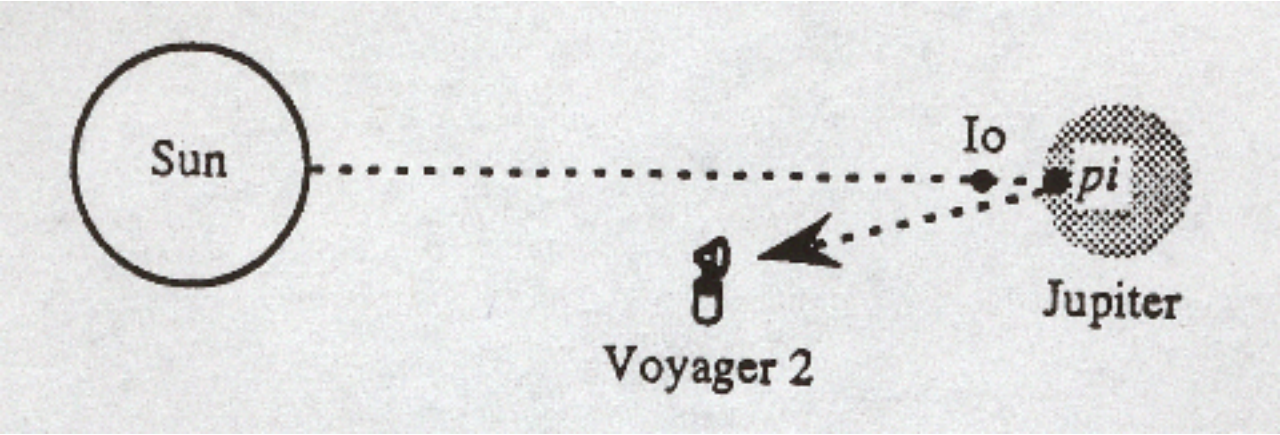
The program out-performs human experts and significantly out-performs non-experts:

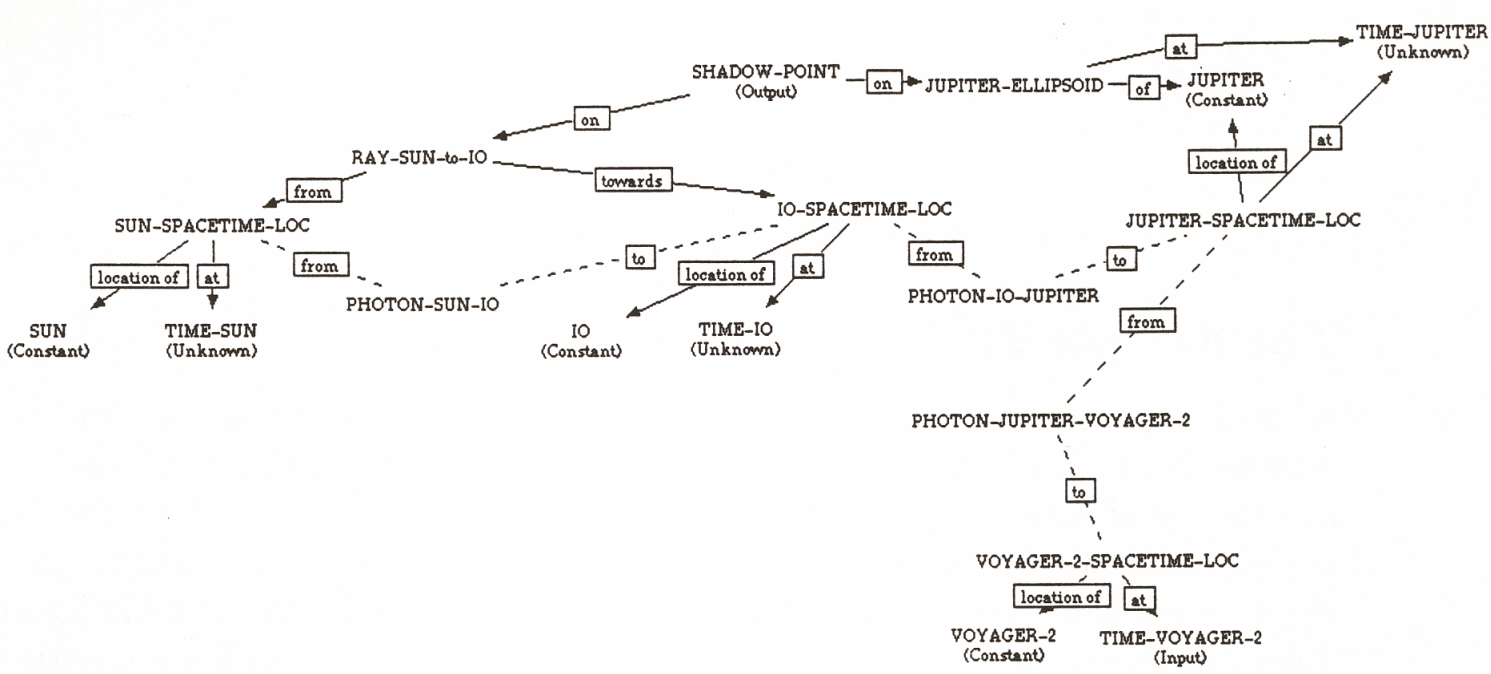
- Expert who knows subroutine library: 30 minutes
- Non-expert: several days
- Program: 3 minutes

Time to construct specification:

- Expert: a few minutes
- Non-expert: 30 minutes

Where is the shadow of Io on Jupiter?





Dotted lines indicate photon motion connections, i.e. *lightlike*?.

Shadow of Io Theorem

```
(all (time-voyager-2-c)
 (find (shadow-point-c)
  (exists
   (time-sun sun-spacetime-loc time-io io-spacetime-loc
    time-jupiter jupiter-spacetime-loc time-voyager-2
    voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
    ray-sun-to-io)
  (and
   (= ray-sun-to-io
      (two-points-to-ray
        (event-to-position sun-spacetime-loc)
        (event-to-position io-spacetime-loc)))
   (= jupiter-ellipsoid
      (body-and-time-to-ellipsoid jupiter time-jupiter))
   (= shadow-point
      (intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
   (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
   (lightlike? io-spacetime-loc jupiter-spacetime-loc)
   (lightlike? sun-spacetime-loc io-spacetime-loc)
   (= voyager-2-spacetime-loc
      (ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
   (= jupiter-spacetime-loc
      (ephemeris-object-and-time-to-event jupiter time-jupiter))
   (= io-spacetime-loc
      (ephemeris-object-and-time-to-event io time-io))
   (= sun-spacetime-loc
      (ephemeris-object-and-time-to-event sun time-sun))
   (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
   (= time-voyager-2
      (abs ephemeris-time-to-time time-voyager-2-c))))))
```

Shadow of Io Program

```
SUBROUTINE SHADOW ( TIMEVO, SHADOW )
DOUBLE PRECISION TIMEVO          ...
INTEGER JUPITE
PARAMETER (JUPITE = 599)         ...
DOUBLE PRECISION RADJUP ( 3 )    ...
CALL BODVAR ( JUPITE, 'RADII', DMYO, RADJUP )
TJUPIT = SENT ( JUPITE, VOYGR2, TIMEVO )
CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY20 )
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
TIO = SENT ( IO, JUPITE, TJUPIT )
CALL FINDPV ( IO, TIO, PIO, DMY30 )
TSUN = SENT ( SUN, 10, TIO )
CALL FINDPV ( SUN, TSUN, PSUN, DMY40 )
CALL VSUB ( PIO, PSUN, DPSPI )
CALL VSUB ( PSUN, PJUPIT, DPJPS )
CALL MXV ( MJUPIT, DPSPI, XDPSPI )
CALL MXV ( MJUPIT, DPJPS, XDPJPS )
CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ),
              RADJUP, RADJUP ( 3 ), P, DMY90 )
CALL VSUB ( P, PJUPIT, DPJUPP )
CALL MTXV ( MJUPIT, DPJUPP, SHADOW )
END
```

Performance

Overall system performance is a win:

- Easy to use, even by novices.
- Easier to revise a stored specification than to make a new one.
- Easy to expand axiom set for new subroutines.

Most programs produced are 2-3 pages of Fortran, consisting mainly of declarations and subroutine calls. There are no `if` statements or loops.

Deductive Composition of Programs

Logic can be used to construct programs by composition of subroutines from a library. We write axioms that describe what the subroutines do, then write a theorem about the goal of the program we want. Proving the theorem constructs a way to accomplish the goal by composition of function calls; these function calls can then be easily converted into a program.

We express what subroutines do with rules; in logic notation:

$$cartesian(p) \ \wedge \ cartesian(q) \\ \rightarrow distance(p, q, euclidist(p, q))$$

This expresses the axiom that if p and q are points in cartesian coordinates, then the distance between them is given by the function $euclidist(p, q)$. This axiom specifies the required inputs of the function $euclidist$, as well as what the function does.

Logic Form of Rules

Our rule in logic form is:

$$cartesian(p) \ \wedge \ cartesian(q) \\ \rightarrow distance(p, q, euclidist(p, q))$$

For backchaining, we put the conclusion on the left:

```
((distance p1 p2 (euclidist p1c p2c))  ; dist is
   (cartesian p1 p1c)   ; if p1c is cart of p1
   (cartesian p2 p2c))  ;  and p2c is cart of p2
```

`cartesian` is now a two-place predicate:
`cartesian(p, pc)` is true if `pc` is the Cartesian form of `p`.

Navigation in the Plane

We will use navigation in the plane as a domain in which to explore deductive program synthesis; this domain can be considered a mini version of the Amphion domain of interplanetary navigation.

Many forms of data are used in solving navigation problems:

- `xy-data` is a list `(x y)`. The corresponding predicate is cartesian: `(cartesian p q)` means that `q` is the Cartesian equivalent of `p`.
- `rth-data` is a list `(r theta)` where theta is in radians, measured counter-clockwise from the x axis. The corresponding predicate is `polar`.
- `rb-data` is a list `(range bearing)` where bearing is in degrees, measured clockwise from north.
- `dd-data` is a list `(distance direction)` where direction is a compass direction such as `n`, `s`, `e`, `w`, `ne`, etc.

Navigation Predicates

- `lat-long` is a list `(latitude longitude)` where the values are in floating degrees; negative longitude denotes west longitude.
- `UTM` or Universal Transverse Mercator is a way of representing positions on Earth that locally maps locations to a flat, Cartesian x-y grid. UTM is a list `(easting northing)` where the values are in meters. `northing` is meters north of the equator and `easting` is meters east of center on a six-degree strip of longitude, plus 500000.

  `((cartesian p (ll2utm q))  (lat-long p q))`

  "The cartesian form of p is `(ll2utm` q `)` if the lat-long form of p is q."
- `city` is a symbol, such as `austin`. An alist maps a city to its lat-long.
- `(movefrom` a b c `)` specifies that c is the result of moving from a by an amount b.

Example Navigation Problem

These predicates can be used for problems such as:

> A helicopter starts at Austin and flies 80000 meters at bearing 20 to pick up a clue; then it flies 100000 meters NW and picks up a treasure. Find the range and bearing to take the treasure to Dallas.

This problem might be nontrivial for a human programmer, even with documentation of the subroutine library. The resulting program is longer than the logic specification.

Difficulties with Deductive Synthesis

There are some difficulties with using logic for deductive synthesis:

- It is possible to get into loops: a Cartesian point can be converted to Polar, and then back to Cartesian again.
- We have the usual problem of combinatoric explosion in searching for a proof.
- The resulting program may not be as efficient as possible: there may be recomputation of the same quantities.

Knowledge Rep. in Predicate Calculus

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

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

Facts can be retrieved in response to *patterns:*

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

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

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

or

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

Rules

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

`If <premises> then <conclusion>`

`If <condition> then <action>`

These forms correspond to the logical implication form:

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

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

Backward Chaining

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

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

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

Problems:

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

Importance of Backchaining

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

Example:

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

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

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

PROLOG

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

Problems:

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

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

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.

Unit Conversion

There are hundreds of units of measurement in common use.[18]

Conversion between different units is an important problem:

- Most programming languages do not support or check units.
- Humans have difficulty converting units. ("Police estimated that the bomb contained 22 pounds of explosive.")
- Use of the wrong units caused a $327 million spacecraft to crash into Mars.

  `en.wikipedia.org/wiki/Mars_Climate_Orbiter`
- Although unit conversion is actually easy, several complex and costly methods have been published.

[18] G. Novak, "Conversion of Units of Measurement", *IEEE Transactions on Software Engineering*, vol. 21, no. 8 (August 1995), pp. 651-661.

Conversion Using SI Units

The scientific standard for units is the *Systeme Internationale*

`en.wikipedia.org/wiki/International_System_of_Units`

previously known as the meter-kilogram-second (MKS) system.

Conversion of units is simple: Each unit is assigned a number that converts it to the corresponding SI unit. Given two units, *source* and *goal*,

$$source * f_{source} = SI = goal * f_{goal}$$

$$goal = source * (f_{source} / f_{goal})$$

When there are multiple units, their factors are multiplied or divided as above. This process is $O(n)$ for a quotient involving n units.

Unit Checking

It is necessary to check that a unit conversion is correct: a length cannot be converted to a mass. Each unit has a corresponding abstract unit, which is a quotient of two products. For example, a force such as `newton` has an abstract unit:

```
(/ (* mass length) (* time time) )
```

If the abstract units for source and goal are divided, and corresponding units in numerator and denominator are cancelled, the result should be 1.

Symbolically, if the terms are collected into a quotient of two lists and the lists are sorted, the two lists should be equal.

For efficiency, the units can be encoded as 32-bit integers that are added and subtracted, giving a result of 0 for a correct conversion.

Special Conversions

Two special conversions of incompatible units are often seen:

- Mass to weight (force)
- Mass to energy

These can be detected by the pattern of abstract units, allowing the correct conversion factor to be applied.

Unit Simplification

A combination of units can be simplified symbolically as follows:

- Cancel corresponding units in numerator and denominator.
- Search for the known composite unit that covers the most terms in the existing expression.
- Repeat until all terms are covered.

```
>(glsimplifyunit '(/ volt ohm))

AMPERE
```

Problem Solving by Unit Conversion

Some physics problems are just unit conversions:

How many Watts is a person on average?

```
To convert from source unit:
(/ (* 2000 KILO CALORIE) DAY)
to goal unit:
WATT
multiply source quantity by: 96.85185185
```

Fixing Conversion Errors

Experience with an on-line conversion system showed that users would ask for impossible conversions, e.g. convert amps to horsepower.

However, it is possible to lead the user to a correct specification:

- Divide the goal unit by the source unit symbolically
- Simplify the resulting unit expression
- Present the result to the user:

  > The units could be converted if you multiplied by an appropriate quantity of VOLT

Units in Programming Languages

It is possible to incorporate units into the type system, make legal conversions automatically, and detect errors:

```
>(gldefun test ( (x (units real meter))
                 (z (units real inch)) )
            (z = x) )

result type: (UNITS REAL INCH)
(LAMBDA (X Z) (SETQ Z (* 39.37007874015748 X)))
```

```
(gldefun testb ( (x (units real meter))
                 (z (units real kilogram)) )
           (z = x) )

glisp error detected in function TESTB
Cannot convert METER to KILOGRAM
in expression: (Z = X)
```

Expert Systems[19]

Expert systems attempt to capture the knowledge of a human expert and make it available through a computer system.[20]

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

[19]These slides jointly authored with Bruce Porter.

[20]Duda, R. O. and Shortliffe, E. H., "Expert Systems Research", Science, vol. 220, no. 4594,15 April 1983, pp. 261-268.

Power-Based Strategy

Some have hoped that powerful theorem-proving methods on fast computers would allow useful reasoning from a set of axioms. Several problems have kept this power-based strategy from succeeding.

- Combinatoric explosion: blind search using even a small axiom set takes excessive time.
- Knowledge representation: few real-world relationships are universally true.
- Lack of inputs: many problems lack some inputs, but require fast action anyway.

Knowledge-Based Strategy

> "In the Knowledge Lies the Power"

The knowledge-based strategy is to include within the program a great deal of knowledge to cover particular cases.

The surprising finding:

> A thousand rules can provide significant performance within a limited domain.

Expert Reasoning

Expert reasoning typically has special characteristics:

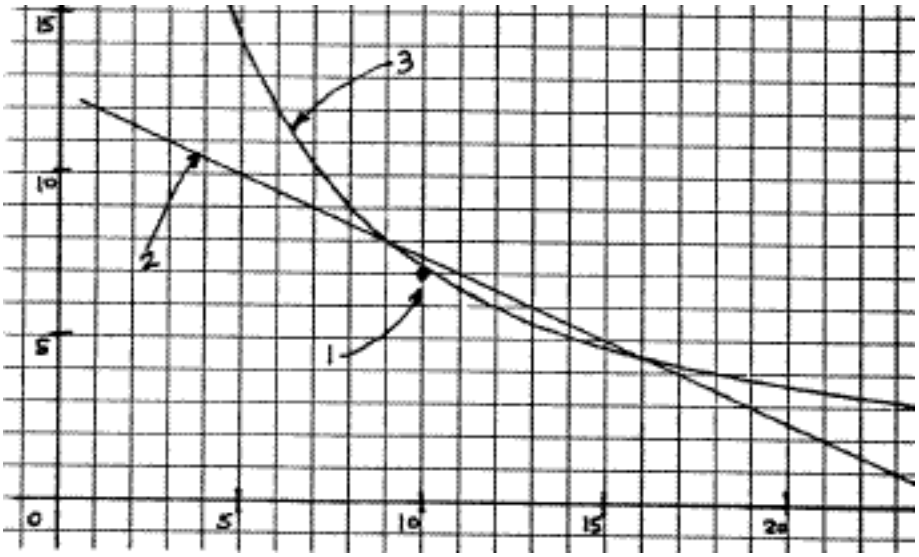
- Use of specialized representations appropriate to the domain and specialized problem-solving methods based on those representations.
- Translation of observables into specialized terminology and representations (e.g., "person has turned blue" into "patient is cyanotic").
- Use of empirical rules of thumb (e.g., "to blow out a tree stump, use one stick of dynamite per 4 inches of stump diameter"[21]).
- 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).

[21]Parker, T., Rules of Thumb, Boston, MA: Houghton Mifflin Publishers, 1983.

Expert Knowledge

Expert knowledge is highly idiosyncratic:

- Build stair steps 7 inches high and 10 inches wide.
- Two times height plus width should equal 25 inches.
- Width times height should equal 72 inches.[22]



- 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.

[22]Parker, T., Rules of Thumb, Boston, MA: Houghton Mifflin Publishers, 1983.

Choosing a Domain

A domain chosen for an expert system (especially a first one) should have the following characteristics:

- Task takes from a few minutes to a few hours for human experts.
- Specialized task (avoid commonsense reasoning).
- Expertise in the area exists and can be identified.
- An expert who is willing to commit significant amounts of time over a long period is available.
- Opportunity for large payoff.

Problem Characteristics

- Complexity: significant expertise required.
- Lack of algorithmic solution to the problem.
- Data may be unavailable or uncertain.
- "Judgment" may be used in reaching conclusion.
- Many different kinds of *knowledge sources* involved in performing task.

Rule-Based Systems

One of the most popular methods for representing knowledge is in the form of Production Rules. These are in the form of:

`if` *conditions* `then` *conclusion*

Example: MYCIN[23]

```
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.
```

[23]Shortliffe, Edward H., Computer Based Medical Consultations: MYCIN, American Elsevier, 1976. Buchanan, Bruce G. and Shortliffe, Edward H., Rule-Based Expert Systems, Addison-Wesley, 1984.

Advantages of Rules

- Knowledge comes in meaningful chunks.
- New knowledge can be added incrementally.
- Rules can make conclusions based on different kinds of data, depending on what is available.
- Rule conclusions provide "islands" that give multiplicative power.
- Rules can be used to provide explanations, control problem-solving process, check new rules for errors.

EMYCIN

EMYCIN was the first widely used expert system tool.

- Good for learning expert systems
- Limited in applicability to "finite classification" problems:
  - Diagnosis
  - Identification
- Good explanation capability
- Certainty factors

Several derivative versions exist.

Rule-Based Expert Systems[24]

MYCIN diagnoses infectious blood diseases using a backward-chained (exhaustive) control strategy.

The algorithm, ignoring certainty factors, is basically backchaining:

Given:

1. list of diseases, Goal-list
2. initial symptoms, DB
3. Rules

For each g $\in$ Goal-list do
If prove(g, DB, Rules) then Print ("Diagnosis:", g)

Function prove (goal, DB, Rules)
If goal $\in$ DB then return True
elseif $\exists r \in$ Rules such that r_{RHS} contains goal
then return provelist(LHS, DB, Rules)[25]
else Ask user about goal and return answer

[24]Shortliffe, E. Computer-based medical consultations: MYCIN. New York: Elsevier, 1976.
[25]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
$P(A\|B)$	Condiional probability of event A given that event B has occurred: $P(rain\|cloudy)$

If A and B are *independent*, then $P(A|B) = P(A)$ and $P(AB) = P(A) * P(B)$.

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

Bayes' Theorem

$P(AB) = P(A|B) * P(B) = P(B|A) * P(A)$

therefore $P(A|B) = P(B|A) * P(A)\ /\ P(B)$

Uses of Bayes' Theorem

In doing an expert task, such as medical diagnosis, the goal is to determine identifications (diseases) given observations (symptoms). Bayes' Theorem provides such a relationship.

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

Suppose: $A =$ Patient has measles, $B =$ has a rash
Then: $P(measles/rash) =$
$$P(rash/measles) * P(measles)/P(rash)$$

The desired diagnostic relationship on the left can be calculated based on the known statistical quantities on the right.

Joint Probability Distribution

Given a set of random variables $X_1 ... X_n$, an *atomic event* is an assignment of a particular value to each X_i.

The *joint probability distribution* is a table that assigns a probability to each atomic event. Any question of conditional probability can be answered from the joint.[26]

	Toothache	¬ Toothache
Cavity	0.04	0.06
¬ Cavity	0.01	0.89

Problems:

- The size of the table is combinatoric: the product of the number of possibilities for each random variable.
- The time to answer a question from the table will also be combinatoric.
- Lack of evidence: we may not have statistics for some table entries, even though those entries are not impossible.

[26] Example from Russell & Norvig.

Chain Rule

We can compute probabilities using a chain rule as follows:

$$P(A \wedge B \wedge C) = P(A|B \wedge C) * P(B|C) * P(C)$$

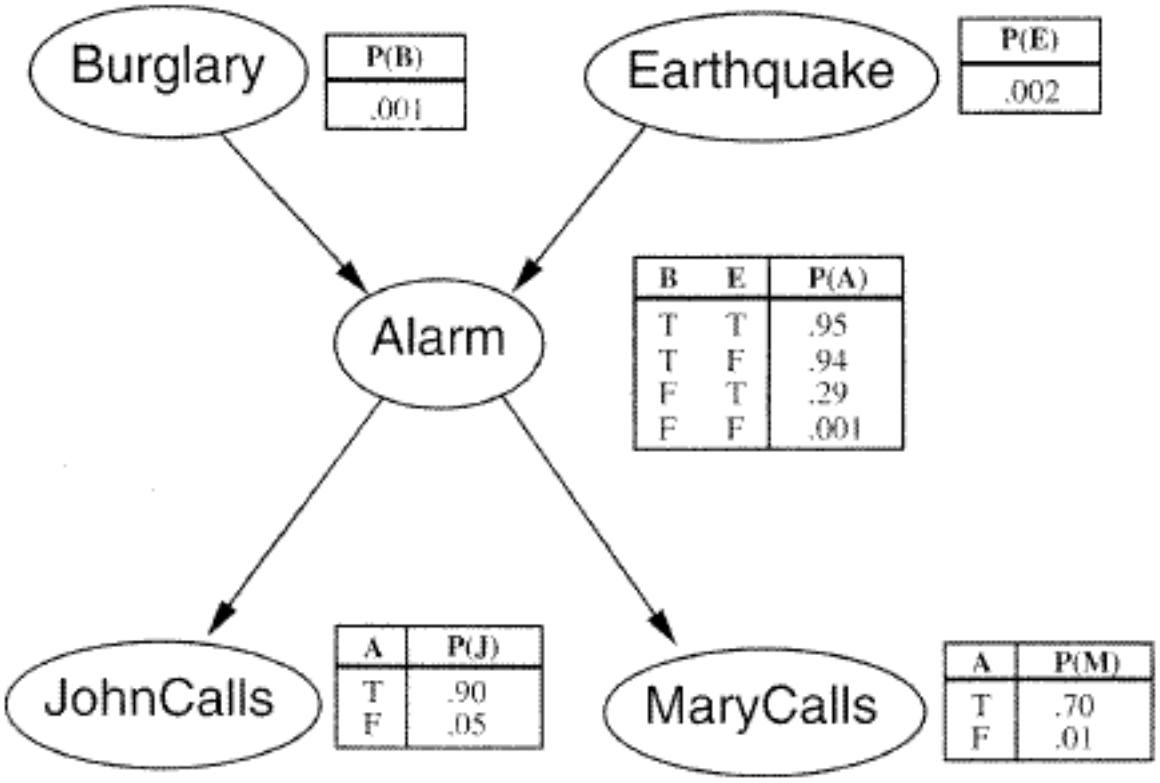
If some conditions $C_1 \wedge ... \wedge C_n$ are independent of other conditions U, we will have:

$$P(A|C_1 \wedge ... \wedge C_n \wedge U) = P(A|C_1 \wedge ... \wedge C_n)$$

This allows a conditional probability to be computed more easily from smaller tables using the chain rule.

Bayesian Networks

Bayesian networks, also called *belief networks* or *Bayesian belief networks*, express relationships among variables by directed acyclic graphs with probability tables stored at the nodes.[27]



[27]Example from Russell & Norvig.

Computing with Bayesian Networks

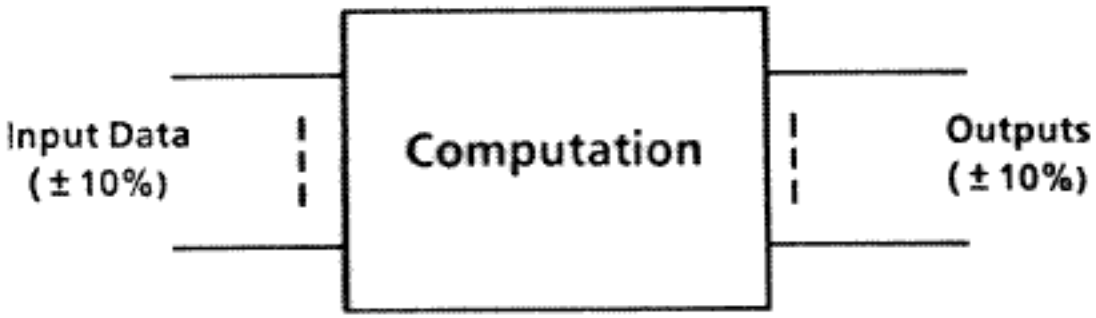
If a Bayesian network is well structured as a poly-tree (at most one path between any two nodes), then probabilities can be computed relatively efficiently.

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) ±10%?

It's hard to argue that extreme accuracy in the computation is required.

Remember:

- Use CF's as a last resort, when a good guess is the best you can do.
- Never trust a CF to have more than one digit of accuracy.

EMYCIN's Certainty Factors

EMYCIN's methods of doing Certainty Factor calculations represent a good set of engineering choices. They have been criticized, but represent a useful technique worthy of study.

Several kinds of CF's are involved:

- Data CF
- CF from antecedent of a rule
- CF due to rule as a whole
- Combination of CF's from multiple rules.

There is a further question of what a CF is supposed to mean.

Certainty Factor Meaning

Traditional probability values are on a scale of 0-1. Shortliffe argues this does not support "ruling out" reasoning of the kind done in medicine.

EMYCIN CF's are on a scale of -1 to +1. A CF combines both a "positive probability" and a "negative probability".

MB	0 - 1	Measure of Belief
MD	0 - 1	Measure of Disbelief
CF = MB - MD	-1 to 1	Certainty Factor
	-1	Definitely False
	0	No information, or cancellation
	+1	Definitely True

When a data parameter is True/False, "False" is represented as "True" with a CF of -1.

EMYCIN Data CF's

Each piece of data has a CF associated with it; even if the parameter is single-valued, there may be multiple possibilities:

COLOR = ((RED .6) (BLUE .3))

Data is referenced using predicates that differ on:

- the CF values that cause the predicate to be "true"
- the CF value returned by the predicate.

The two most commonly used predicates are:

- `SAME`: "True" if data $CF > .2$; returns data CF.
- `KNOWN`: "True" if data $CF > .2$; returns 1.0

EMYCIN Antecedent CF

The *antecedent* ("if part") of a rule is usually a conjunction of conditions, using the EMYCIN `$AND` function:

```
($AND (SAME CNTXT GRAM GRAMNEG)
      (SAME CNTXT MORPH COCCUS))
```

`$AND` operates as follows:

1. If any clause is false (`nil`) or has $CF \leq .2$, `$AND` returns false (`nil`). Thus, .2 is used as a cutoff threshold. Any data believed less strongly than .2 is considered to be false.
2. If every clause has $CF > .2$, `$AND` returns the minimum of the clause CF values.

There is also a function `$OR` that returns the maximum of its argument CF values.

Rule Certainty Factors

Premise:

```
($AND (SAME CNTXT SITE BLOOD)
      (SAME CNTXT GRAM GRAMNEG)
      (SAME CNTXT MORPH ROD)
      (SAME CNTXT BURNED))
```

Action:

```
 (CONCLUDE CNTXT
           IDENTITY PSEUDOMONAS-AERUGINOSA
           TALLY 400)
```

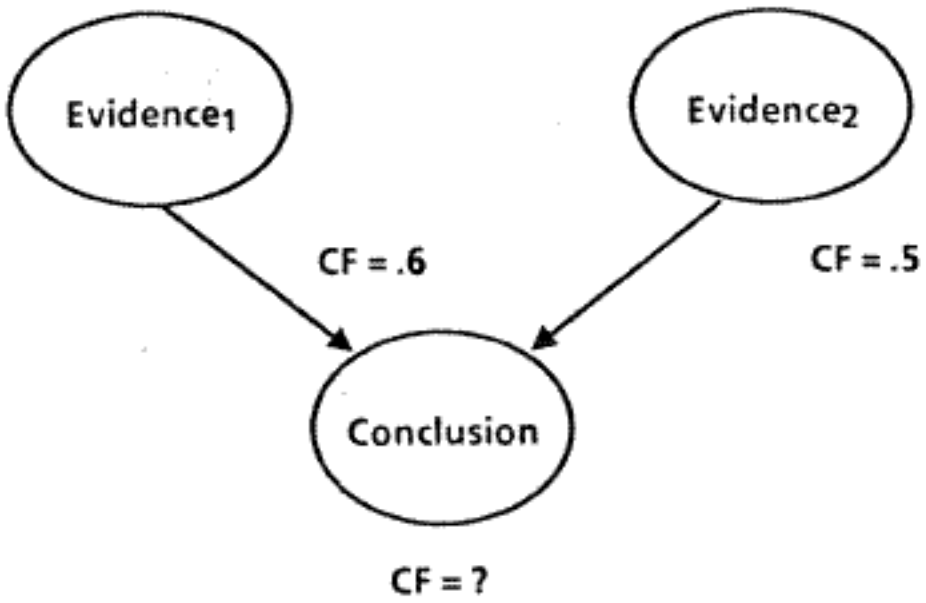
The result of the rule as a whole is calculated as follows:

1. `$AND` sets the global variable `TALLY` to the minimum CF of its components.
2. The Rule CF is `TALLY` times the CF specified in the `CONCLUDE` line, divided by 1000. In this example, the rule CF is 0.4 .

This forms the input to the CF Combination algorithm.

Certainty Factor Combination

When two sets of evidence imply the same conclusion, there is a need to compute the total certainty factor based on the certainties of the sets of evidence.



A CF combination method should be:

- Commutative: $A \cdot B = B \cdot A$
- Associative: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

This will make the resulting CF independent of the order in which pieces of evidence are considered.

Certainty Factor Combination

If a datum's previous certainty factor is CF_p and a new rule computes a certainty factor CF_n, the combined certainty factor is given by:

$$cfcombine(CF_p, CF_n) =$$

$$\begin{cases} CF_p + CF_n * (1 - CF_p) & CF_p > 0,\ CF_n > 0 \\ (CF_p + CF_n) \\ \quad /(1 - min(|CF_p|, |CF_n|)) & \text{signs differ} \\ -cfcombine(-CF_p, -CF_n) & CF_p < 0,\ CF_n < 0 \end{cases}$$

This algorithm has a desirable feature: it is associative and commutative; therefore the result is independent of the order in which rules are considered.

A CF of + 1 or -1 is dominant and sets the combined CF to that value.

Summary of CF Computations

- $CF > .2$ threshold
- `$AND` takes minimum CF in premise
- `conclude` CF $= CF_{premise} * CF_{rule}$
- CF combination algorithm

Examples:

```
If:     A (.6)
    and B (.3)
    and C (.4)
Then: conclude D tally 700
```

The resulting rule value for `D` is the minimum premise CF `tally` (.3) times the rule CF (.7), or 0.21 .

Suppose that two separate rules reach the same conclusion with CF's of 0.5 and 0.6 ; the resulting CF is $.5 + .6 * (1 - .5) = .8$. This could also be computed as $.6 + .5 * (1 - .6) = .8$.

Contradictions:

EMYCIN's CF calculations allow contradictory rules to cancel one another. While in logic a contradiction is intolerable, use of rules and exceptions, expressed as contradicting rules, seems to be the way humans often think.

Example: FUO program for diagnosing Fever of Unknown Origin. Patient is a 17 year old female with persistent high fever, headache, lethargy, cardiac symptoms ...

Results: Lung cancer (.81), Endocarditis (.7)

Endocarditis was the correct diagnosis. The physician expert remarked that lung cancer was consistent with the symptoms, but that patients that young never get lung cancer.

One way to handle this in EMYCIN is to add a rule that "rules out" certain cancers in young patients; the CF can be made a function of the patient's age.

Duplicate Rules

Duplicated rules in EMYCIN are harmful because each of the rules will fire; this will cause the CF's of the rules to be combined, giving a larger CF than was intended.

In a large expert system, it is easy for duplicate rules to be created by different rule-writers (or even the same one).

In logic, duplicated rules have no effect.

Rule Subsumption

A common programmer error is to leave out one or more clauses in the antecedent of a rule. This causes the rule to be over-broad in its application and may cause it to subsume other rules.

Example: SACON structural analysis consultant[28]

```
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.

[28]Bennett, J. S. and Engelmore, R. S., "SACON: a Knowledge-based Consultant for Structural Analysis", Proc. IJCAI-79, pp. 47-49, Morgan Kaufmann Publishers.

Increasing Certainty

EMYCIN systems sometimes include rules of the form: $A \wedge B \rightarrow A$

Such a rule is logically redundant, but may serve to increase the CF of the conclusion based on additional evidence.

```
If: 1) The identity of the snake is
       rattlesnake, and
    2) It bites someone, and
    3) He dies
Then: There is strongly suggestive
      evidence (.8) that the identity of
      the snake is rattlesnake.
```

EMYCIN CF vs. Probability Theory

EMYCIN certainty factor calculations differ in significant ways from standard probability theory. These differences have attracted criticism, but often have good practical motivations.

CF Threshold: EMYCIN generally considers anything with CF $< .2$ to be false.

Q: "Can't this prevent several pieces of weak support from adding up to a significant level of support?"

A: Yes, it is possible, but doesn't seem to be a problem in practice.

The benefit of the CF threshold is that it keeps the system from asking a lot of dumb questions:

```
If: A (.01)
and B (.02)
and C (.001)
and D (as yet unknown) ...
```

In such a case, we don't want the system to ask questions about D, which is a most unlikely prospect. Asking dumb questions will quickly discourage potential users of the system.

Sensitivity Analysis

In general, sensitivity analysis attempts to determine the sensitivity of the output of a computation to small changes in the input.

An expert system should be relatively insensitive to small changes in the input or CF's; high sensitivity indicates bad design.

The sensitivity of MYCIN to small changes in CF values has been empirically tested; MYCIN was found to be relatively insensitive to CF values.

In part, this is due to the fact that MYCIN "plays it safe:" it treats for all organisms found with $CF > .2$.

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.

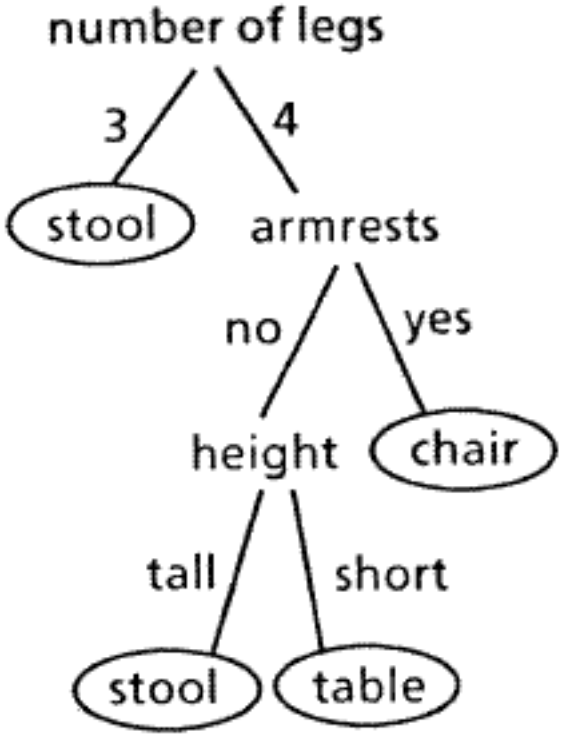
Assumption is that it is easier for expert to demonstrate his expertise than to "tell all he knows".

Input to the induction algorithm is classified examples (which corresponds to I/O of human expert):

$< f_{11}, f_{12}, ..., f_{1n} > \ \ classification_1$
$< f_{21}, f_{22}, ..., f_{2m} > \ \ classification_2$

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.

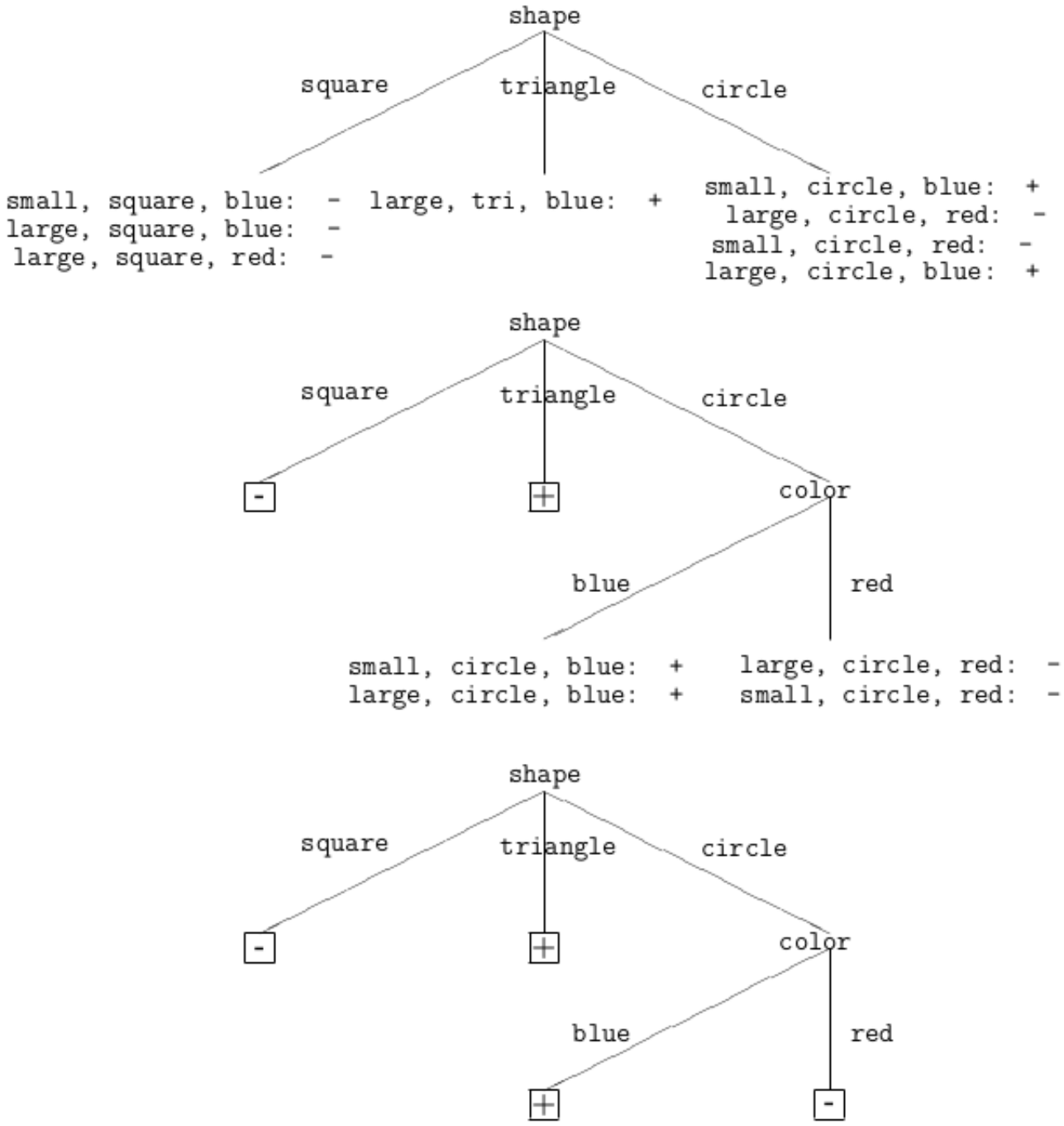
Example of Rule Induction[29]

Classifications = { +, -}
Features = {size, shape, color}
Domains:
  size = {large, small}
  shape = {square, triangle, circle}
  color = {blue, red}

Training set:
small, circle, blue: +
small, square, blue: -
large, square, red: -
large, circle, red: -
large, square, blue: -
small, circle, red: -
large, triangle, blue: +
large, circle, blue: +

[29]Quinlan, "Learning Efficient Classification Procedures", *Machine Learning*, Morgan Kaufmann Publ., 1983.

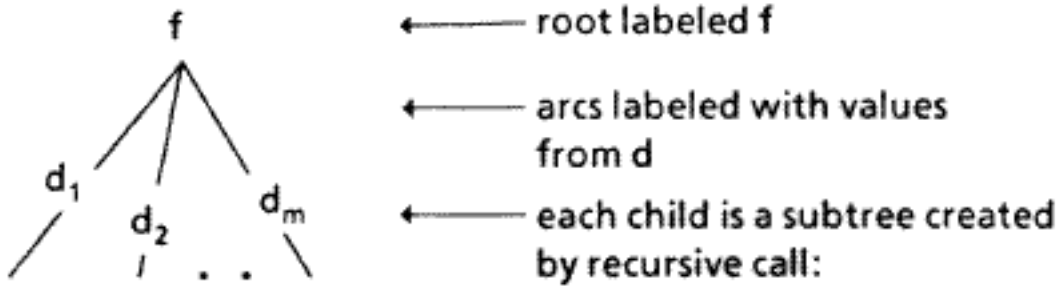
Final Decision Tree with Classifications



Algorithm for Rule Induction

Instances:	a set of training instances
Features:	feature vector $(f_1, f_2, ..., f_n)$ input from teacher
Domains:	set of domains for Features $\{d_1, d_2, ..., d_m\}$
Classes:	set of classes $\{c_1, c_2, ..., c_k\}$ (simply $\{+, -\}$ for single concept learning.)

Function formrule (Instances, Features, Domains, Classes)
For some class $\in$ Classes
If all members of Instances fall into class
then return class
else f $\leftarrow$ select-feature (Features, Instances)
d $\leftarrow$ domain from set Domains corresponding to f
return a tree of the form:



Alternatives for `select-feature`

1. Random selection: guaranteed to give a decision tree which is consistent with the training set. No guarantee of optimality.
2. Information theoretic selection: select the feature which maximally partitions the set of instances. Heuristic for finding decision tree with minimum expected classification time.
3. Minimal cost selection: allow for the fact that some features are costly to evaluate. For example, body temperature is easier to determine than lung-capacity. Put least costly features high in the tree.

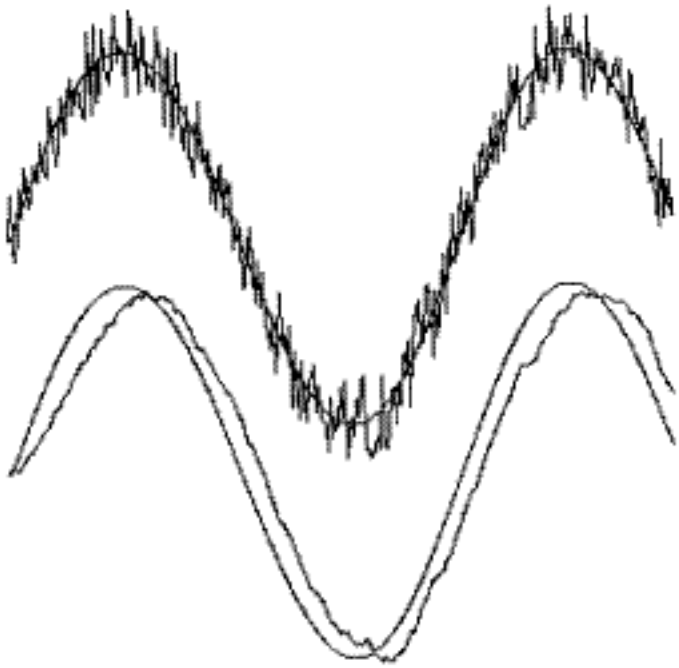
Limitations of Rule Induction

1. "Flat" classification rules produced with no justification facility
2. Lots of training examples are necessary
3. Training must be noise-free
4. Each training example must be described using all the features
5. Classifications and features are static sets
6. Rules produced do not distinguish between correlation and causality

Digital Low-Pass Filter

A simple mechanism that can be used to learn parameter values over time is the digital low-pass filter. [30] A simple digital low-pass filter is defined by:

$$out_{i+1} = \alpha * in_i + (1 - \alpha) * out_i, \text{ where } \alpha << 1.$$



This filter 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.

[30] The 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. Experts cannot "tell all they know", but they quickly spot errors. When the expert spots an error, that leads to new rules.

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.

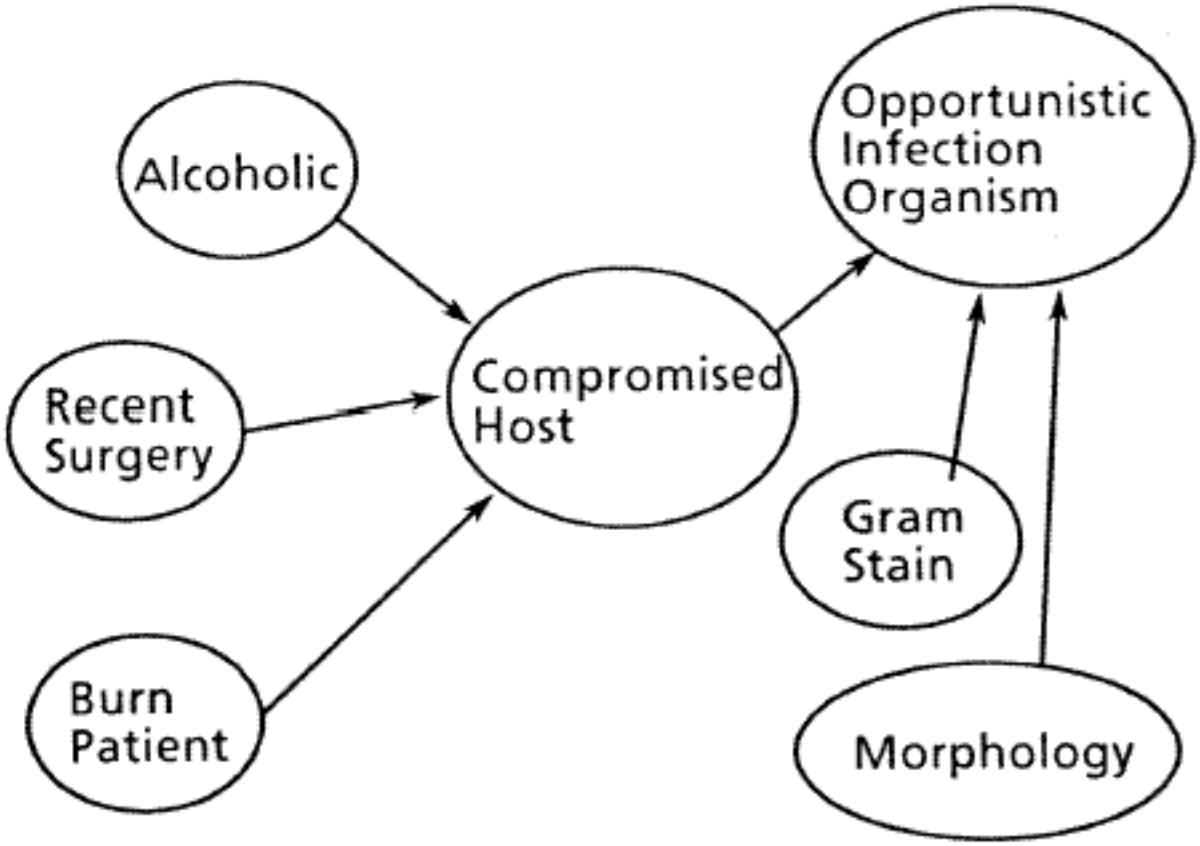
Result: new rules added to rule out certain cancers in young patients. Patient age can be used to determine certainty factor of ruling out lung cancer.

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

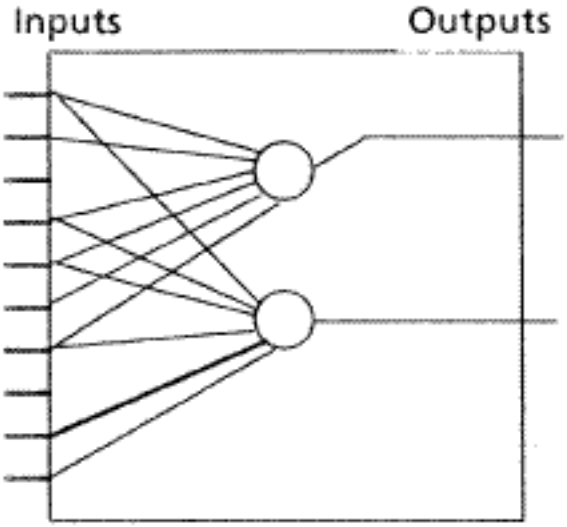
A compromised host is a patient who has been weakened and therefore cannot fight off infections as well as a normal person.



Advantages of Conceptual Islands

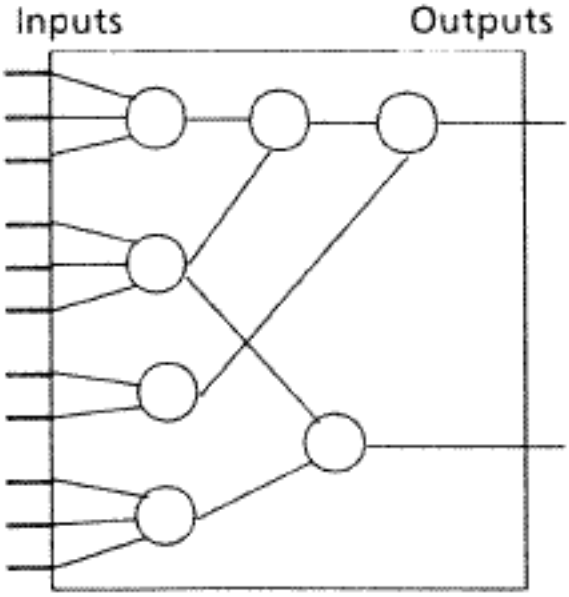
Conceptual Islands reduce the number of rules required and aid robustness.

Without Conceptual Islands:



Very Large Rules
Any input being false (or missing) causes rule to fail
Too many rules required
Unnoticed gaps in coverage
Hard to change
No explanation capability
Lack of meaningful structure

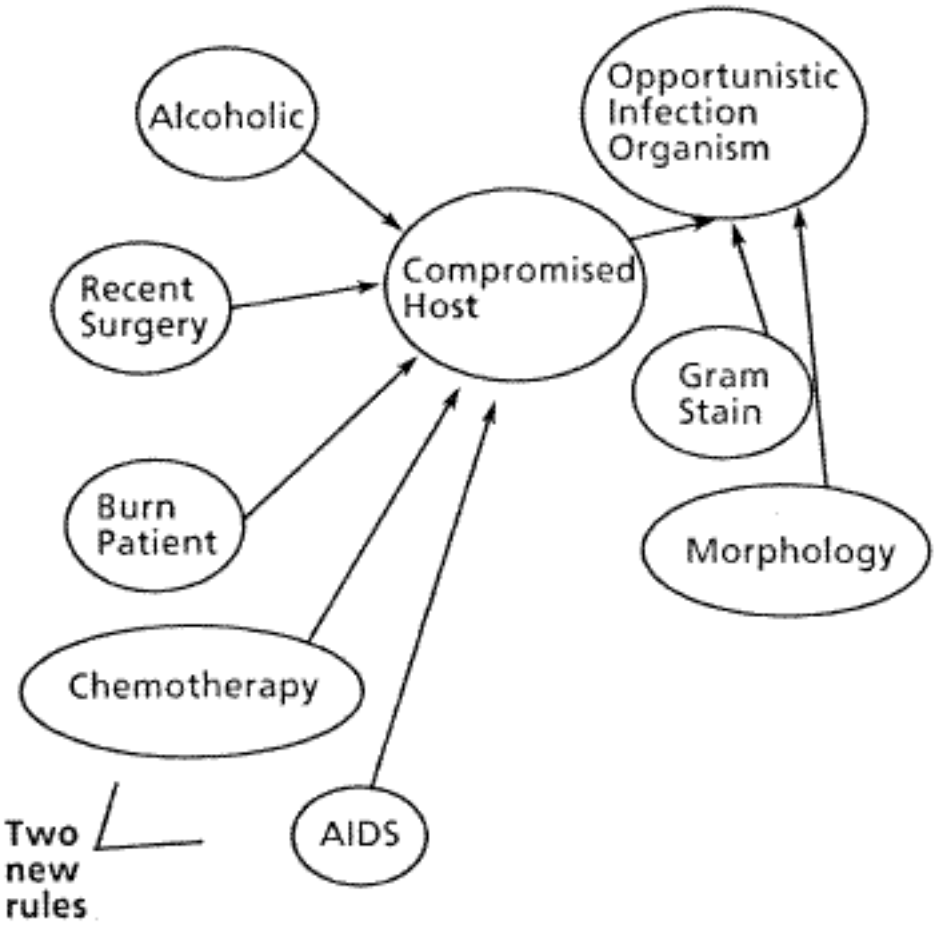
With Conceptual Islands:



Smaller Rules
Robustness: multiple ways to reach the same conclusion
Reasonable number of rules
Fuller coverage
Easy to change/expand
Explanations that make sense
Understandable structure

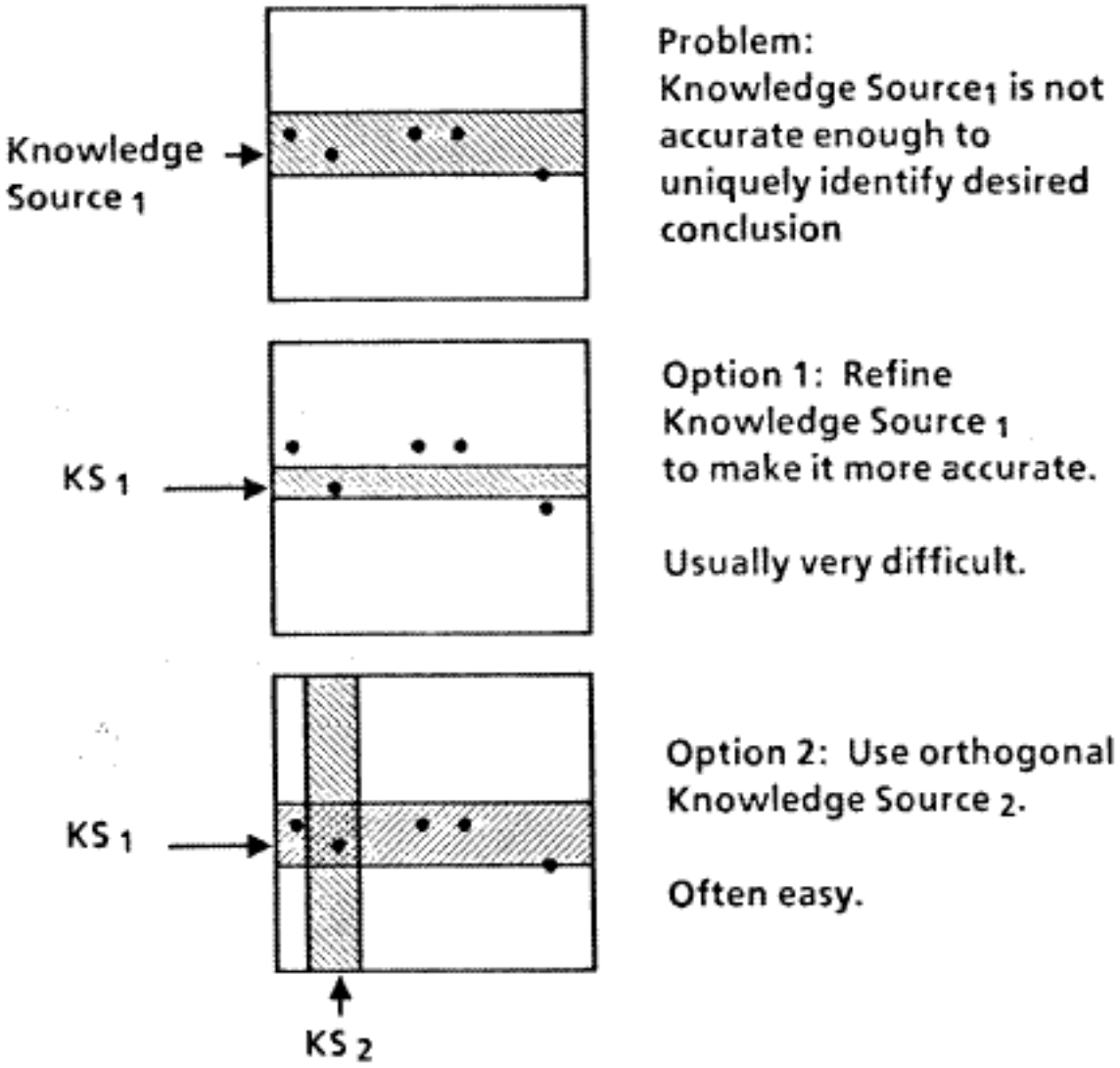
Expansion with Conceptual Islands

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: the *training set phenomenon* or *overfitting*.
- 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[31] is often used to show true positive rate *recall* plotted against false positive rate.

[31] 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:[32]

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

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

Why Study Natural Language?

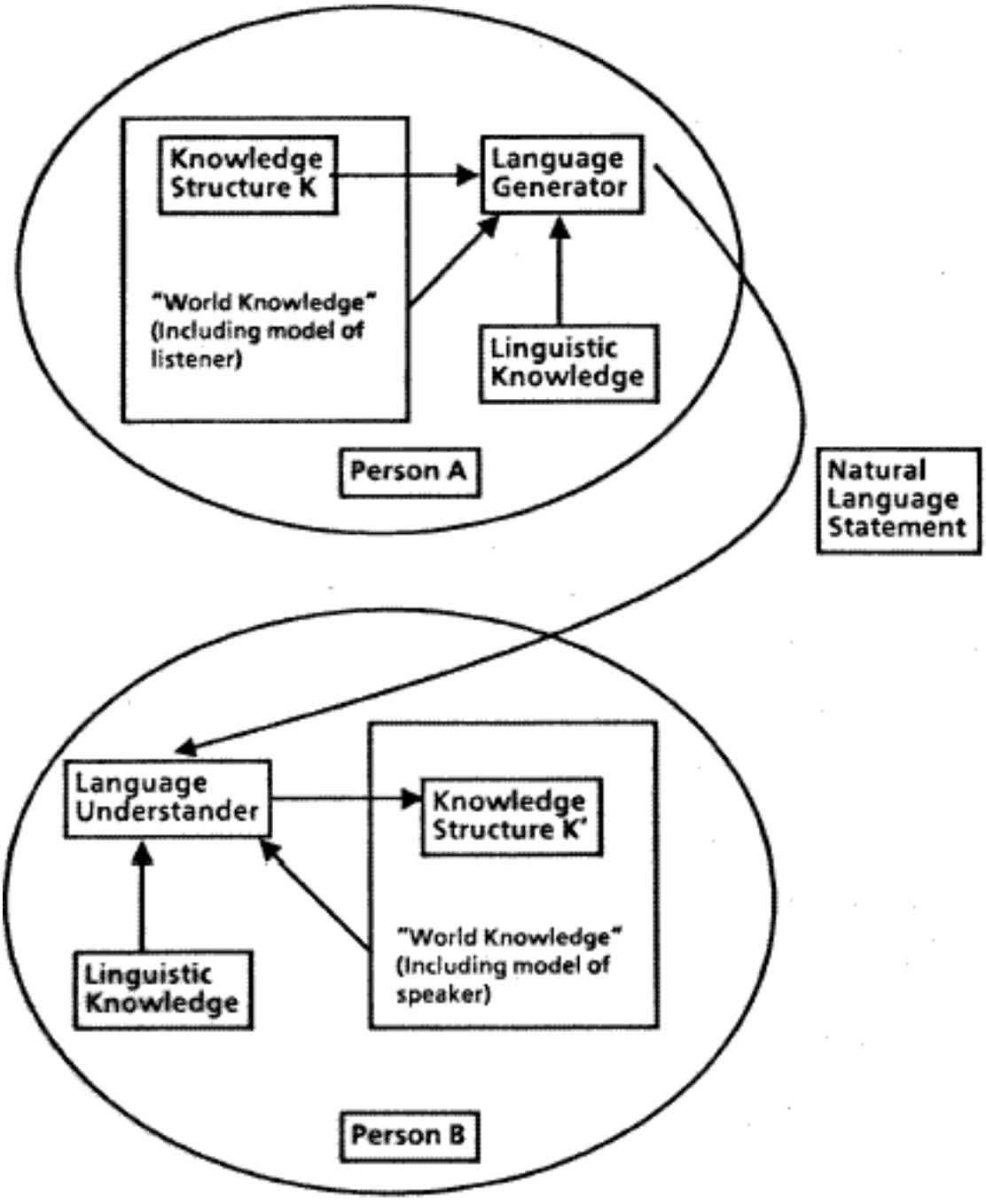
Theoretical:

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

Practical:

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

Model of Natural Language Communication



Minimality of Natural Language

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

All of our communication methods are serial:

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

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

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

Zipf's Law

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

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

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

facsimile transmission	fax
latissimus dorsae	lat
Mediterranean	Med
robot	bot

Areas of Natural Language

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

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

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

Computer Language Understanding

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

There are many problems in making such a translation:

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

    ```
    The pitcher broke his arm.

    The pitcher broke.
    ```

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

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

Problems in Understanding Language ...

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

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

- **Anaphora:**[33] Words that refer to others.

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

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

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

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

[33] The singular is *anaphor*.

Morphology

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

For English, a fairly simple suffix-stripping algorithm plus a small list of irregular forms will suffice.[34]

```
running   -->   run + ing

went      -->   go  + ed
```

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

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

Lexicon

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

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

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

Lexical Features

These features are the basis of lexical coding.[35]

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

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

[35] slide by Robert F. Simmons.

Size of Lexicon

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

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

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

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

Statistical Natural Language Processing

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

A *type* is a word form, while a *token* is each occurrence of a word type. *N-grams* are sequences of N words: *unigrams*, *bigrams*, *trigrams*, etc. Statistics on the occurrences of n-grams can be gathered from text *corpora*.[36]

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

N-gram approximations to Shakespeare:[37]

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

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

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

Part-of-Speech Tagging

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

> 'Twas brillig, and the slithy toves
> did gyre and gimble in the wabe.[38]

A *Hidden Markov Model* (HMM) tagger chooses the tag for each word that maximizes: [39]

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

For a bigram tagger, this is approximated as:

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

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

[38] from Jabberwocky, by Lewis Carroll.
[39] Jurafsky, *op. cit.*

AI View of Syntax

We need a compact and general way to describe language:

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

Unfortunately, this is not easy to achieve.

> But the English ... having such varieties of incertitudes, changes, and Idioms, it cannot be in the compas of human brain to compile an exact regular Syntaxis thereof.[40]

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

Grammar

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

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

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

Language Generation

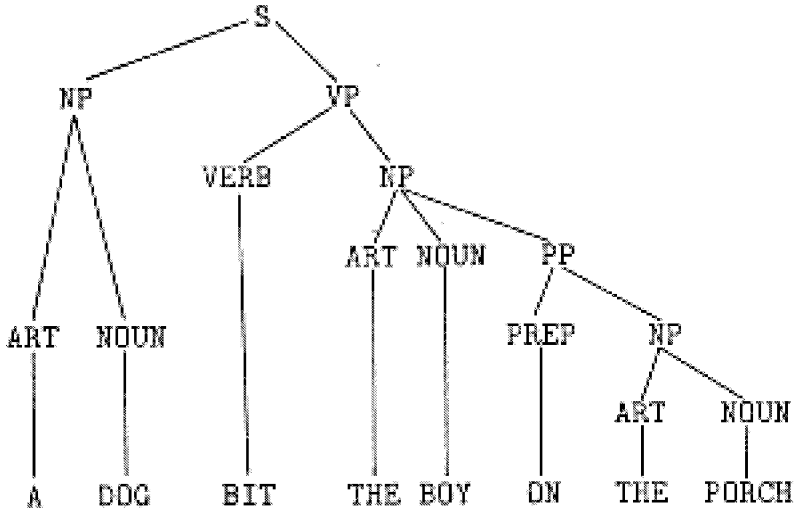
Sentences can be generated from a grammar by the following procedure:

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

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

Parsing

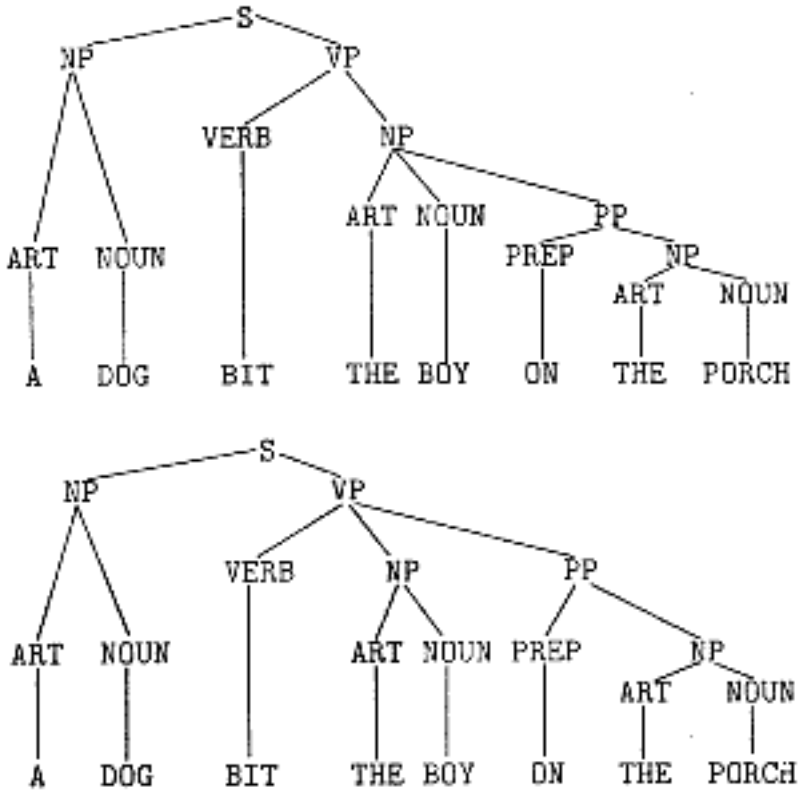
Parsing is the inverse of generation: the *assignment of structure* to a linear string of words according to a grammar; this is much like the "diagramming" of a sentence taught in grammar school.



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

Ambiguity

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



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

Sources of Ambiguity

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

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

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

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

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

Foreign Languages

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

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

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

  gloria in excelsis Deo

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

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

- Familiar, formal, honorific forms of language.

  sie *you*
  Du *Thou*

Formal Syntax

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

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

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

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

Notation

The following notations are used in describing grammars and languages:

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

Phrase Structure Grammar

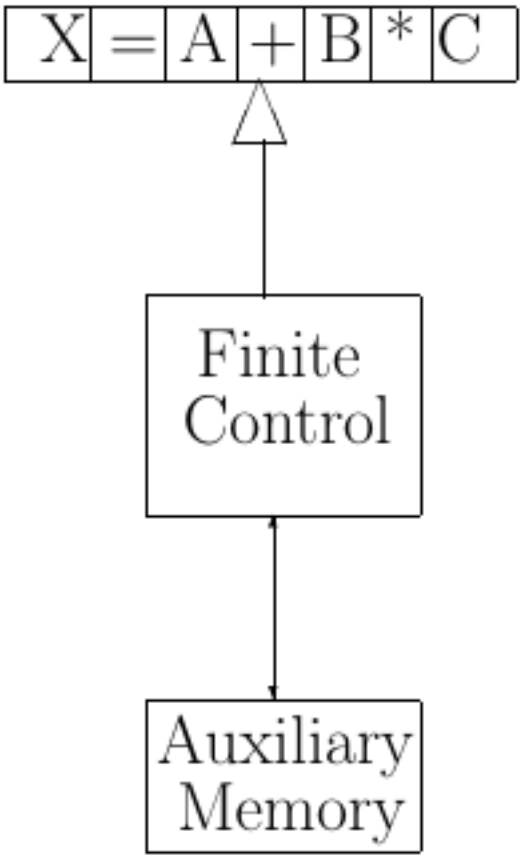
A grammar describes the structure of the sentences of a language in terms of components, or phrases. The mathematical description of phrase structure grammars is due to Chomsky.[41]

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

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

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

Recognizing Automaton



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

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

Regular Languages

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

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

Context Free Languages

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

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

What Kind of Language is English?

- English is Context Free.[42]
- English is not Context Free.[43]
- English is Regular:
  - English consists of finite strings from a finite vocabulary.
  - English is recognized by people with finite memory.
  - There is no evidence that peoples' parsing time is more then $O(n)$.

A better question to ask is:

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

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

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

Parsing

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

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

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

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

There are several well-known context-free parsers:

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

Top-down Parser

A *top-down* parser begins with the Sentence symbol, `<S>`, expands a production for `<S>`, and so on recursively until words (terminal symbols) are reached. If the string of words matches the input, a parsing has been found.[44]

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

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

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

Bottom-up Parsing

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

```
     <NP>
     /  \
 <art> <noun>
   |     |
  The   man   ate   fish
```

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

Augmented Transition Networks

An ATN [45] is like a finite state transition network, but is augmented in three ways:

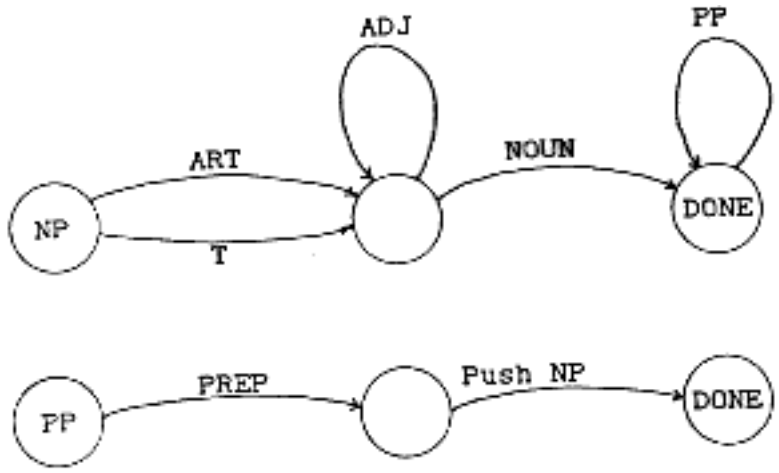
1. **Arbitrary tests** can be added to the arcs. A test must be satisfied for the arc to be traversed. This allows, for example, tests on agreement of a word and its modifier.
2. **Structure-building actions** can be added to the arcs. These actions may save information in *registers* to be used later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

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

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

Augmented Transition Networks

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

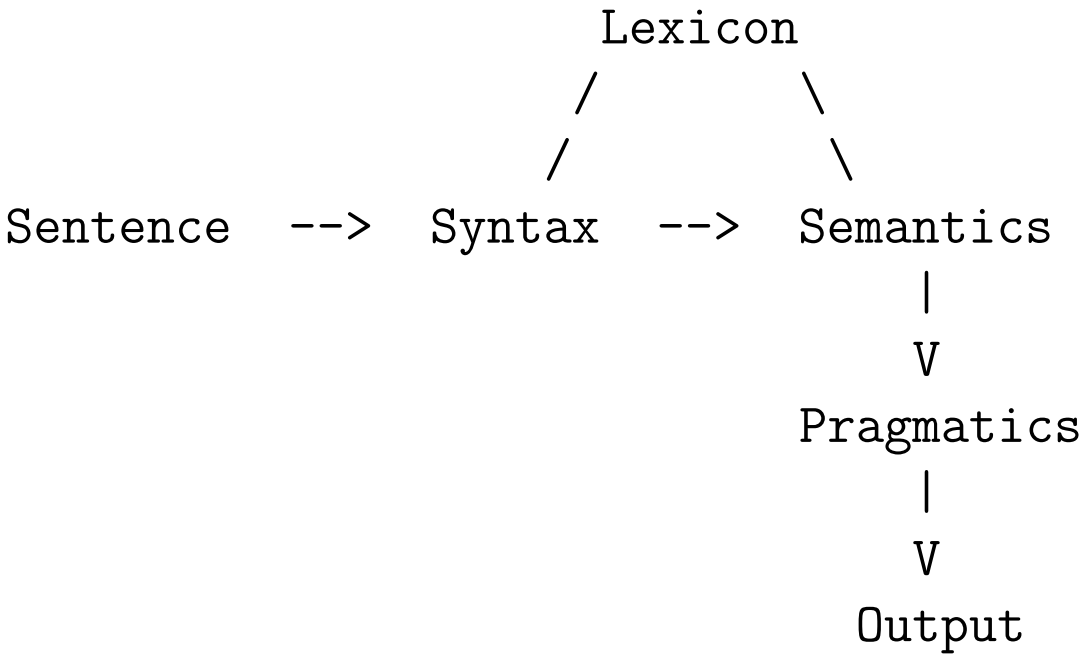


ATN's are more readable than lists of productions.

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

Separability of Components

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



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

Problems with Separability

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

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

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

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

Combining Syntax and Semantics

There are several advantages to combining syntactic and semantic processing:

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

  All your base are belong to us.

  This sentence no verb. – D. Hofstadter

How to Combine Syntax & Semantics

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

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

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

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

Natural Language as an AI Problem

Natural language understanding is a classical AI Problem:

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

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 [46]

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

(defn op->english [op]
  (list 'the
      (second (assocl op '((+ sum)
                           (- difference)
                           (* product)
                           (/ quotient)
                           (sin sine)
                           (cos cosine)))) 'of))
; expression x -> (list of words)
(defn exp->english [x]
  (if (cons? x)                  ; operator?
      (append
        (op->english (op x))
        (append (exp->english (lhs x))
                (if (null? (rest (rest x)))
                    '()          ; unary
                    (cons 'and
                      (exp->english (rhs x))))))
      (list x) ) )               ; leaf: operand
```

[46] file `expenglish.clj`

Generating English

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

>(exp->english 'x)

(X)

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

(THE SUM OF X AND Y)

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

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

Simple Language Processing: ELIZA

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

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

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

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

Problems:

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

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

Spectrum of Language Descriptions

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

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

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

Semantic Grammar

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

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

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

Advantages:

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

Semantic Grammar: Extended Pattern Matching

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

Example:

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

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

Example Semantics for a Semantic Grammar

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

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

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

This query is parsed by the top-level production

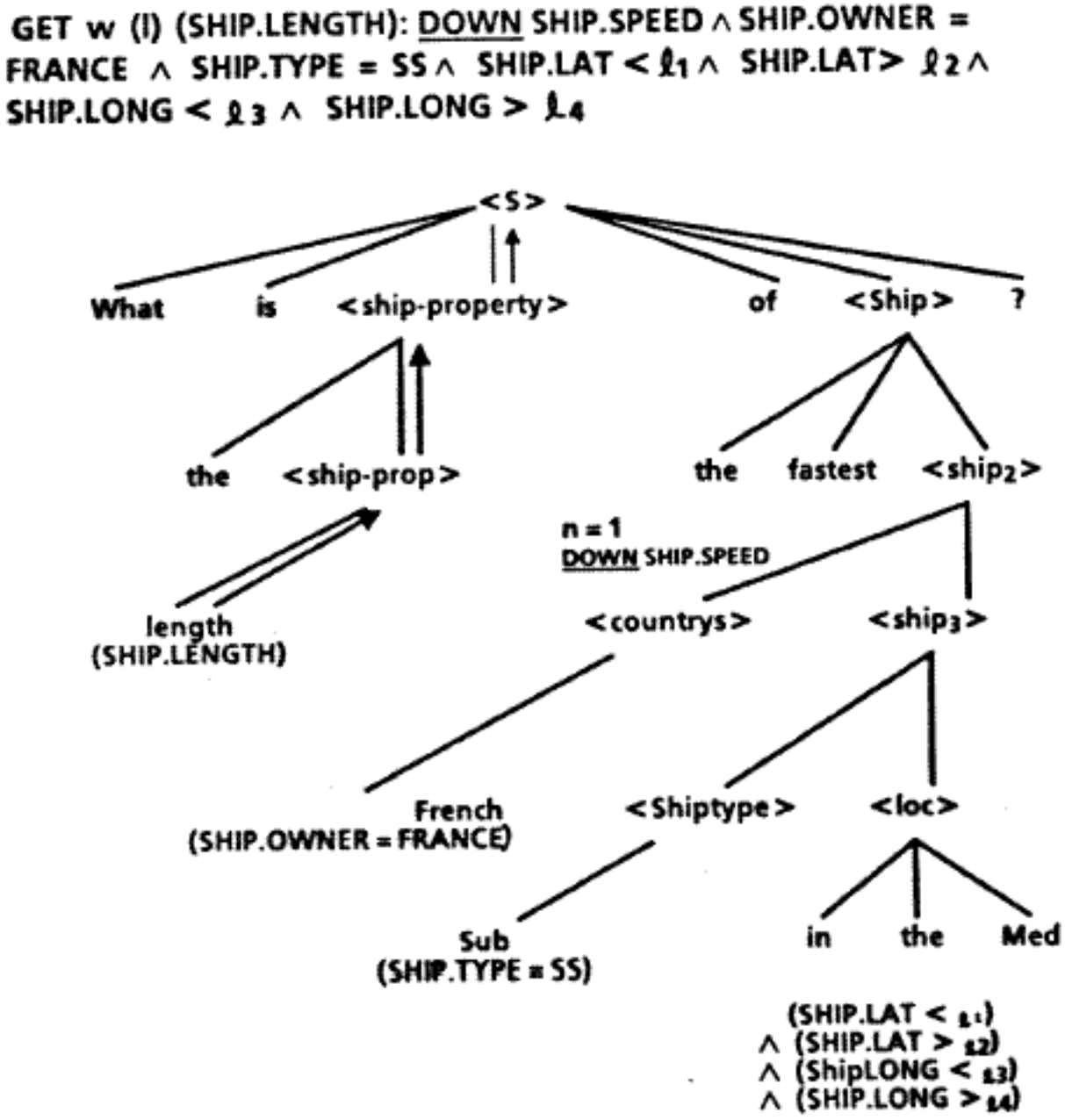
```
<S> --> What is <ship-property> of <ship>?
```

which is conveniently structured in terms of:

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

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

Compositional Semantics



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

Additional Language Features

Semantic grammar enables additional features that help users:

- Spelling correction:

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

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

- Sentence fragments:

  ```
  What is the length of Kennedy?


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

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

Recursive Descent

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

Given a grammar rule:

```
S -> NP VP
```

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

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

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

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

Parsing English

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

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

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

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

ATN Program [47]

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

[47] file `atn.clj`

Parsing Functions

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

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

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

Grammar Compiler [48]

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.

```
;              $$        $1   $2
(rulecompr '(loc ->  (in (city))
                (restrict 'city $2))
           'locfn)

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

[48] file `gramcom.clj`

Sentence Pointer Handling

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

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

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

Sentence Pointer Handling ...

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

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

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

Lexicon Example

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

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

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

Word Category Testing

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

The lexicon and category testing can do multiple tasks:

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

Database Access

Database access requires two kinds of information:

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

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

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

   ```
   (retrieve 'field)
   ```

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

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

Database Access

Our example database program takes queries of the form:

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

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

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

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

Building Database Access

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

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

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

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

Grammar Rules

A grammar rule has the form:

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

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

The allowable items on the right-hand side are:

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

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

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

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

Restaurant Database Grammar

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

Notes on Database Grammar

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

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

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

Multiple actions can be combined using `do`:

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

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

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

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

Restaurant Queries

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

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

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

Physics Problems[49]

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

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

[49] file `physgram.lsp`

Physics Queries

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

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

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

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

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

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

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

Physics Units

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

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

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

3.2429278661312964
```

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

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

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

4869.478351881704
```

Physics Changes

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

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

inverse-square

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

4.0
```

These questions can be answered easily as follows:

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

Natural Language Interfaces

Interfaces for understanding language about limited domains are easy:

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

Benefits:

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

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

Problems with NL Interfaces

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

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

  Does this mean:

  1. Nobody failed.
  2. CS 381K was not offered in Summer 2018.
  3. There is no such course as CS 381K.

- **Gaps:** Is it possible to state the desired question?

Mapping

A *mapping* $M : D \to R$ 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*.

Implementation of Mapping

A mapping can be implemented in several ways:

- A function such as `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 `TreeMap` or `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 square [x] (* x x))
(defn hypotenuse [x y]
  (Math/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 *associative* if $a \circ (b \circ c) = (a \circ b) \circ c$.

An operation $\circ$ is *commutative* if $a \circ b = b \circ a$.

If an operation $\circ$ is *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.

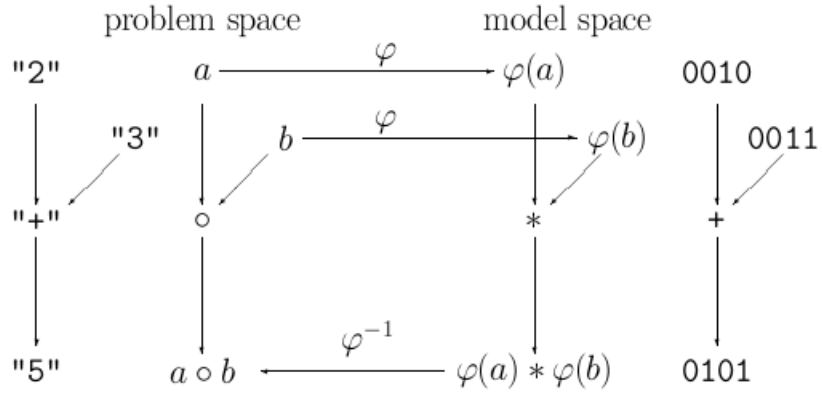
In many cases, parts of the operation (e.g. partial sums) can be done independently as well.

Computation as Simulation

It is useful to view computation as simulation, *cf.*: *isomorphism of semigroups*.[50]

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

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




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

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

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

Mapping in Lisp

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

```
>(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)
```

Mapcat and Filter

The Clojure function `mapcat` (`mapcan` in Lisp) works much like `map` (`mapcar` in Lisp), but with a different way of gathering results:

- The function called by `mapcat` returns a *list* of results (perhaps an empty list, signifying no result).
- `mapcat` concatenates the results; empty lists vanish.

A function related to `mapcat` is `filter`, Which returns a list of only those items that satisfy a predicate.

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

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

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

Input Filtering and Mapping

We can use `mapcat` 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
                      ;             (filter)

>(mapcat 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.

```
>(reduce + '(1 2 3 17))

23

>(reduce * '(1 2 3 17))

102
```

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

```
>(reduce + (mapcat testforz
                   '(z m u l e z r u l e z)))
;                   (1)       (1)       (1)
; = (reduce + '(1 1 1))

3
```

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:

```
(defn edist [pointa pointb]
  (Math/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, to "emit" a result, 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 Clojure to introduce how it works.

Mapreduce in Clojure

```
(defn mapreduce [mapfn reducefn lst]
  (let [rawresult (mapcat mapfn lst)]
    (let [sorted
          (sort (fn [x y]
                  (compare (first x) (first y)))
                rawresult)]
      (let [keyvals (combinekeys sorted)]
        (map (fn [lst]
               (list (first lst)
                     (apply reducefn (rest lst))))
             keyvals) ) ) ) )


>(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

```
>(mapreduce identity +
            '(((a 3) (b 2) (c 1))
              ((b 7) (d 3) (c 5))) t)
Mapping: ((A 3) (B 2) (C 1))
   Emitted: (A 3)
   Emitted: (B 2)
   Emitted: (C 1)
Mapping: ((B 7) (D 3) (C 5))
   Emitted: (B 7)
   Emitted: (D 3)
   Emitted: (C 5)
Reducing: D (3)    = 3
Reducing: C (5 1)  = 6
Reducing: B (7 2)  = 9
Reducing: A (3)    = 3

((D 3) (C 6) (B 9) (A 3))
```

MapReduce Example

```
(defn nutrition [food]
  (rest (assocl 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) (protein 28) (fiber 8) (fat 28)
 (carbs 40) (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 $(inputkey, inputvalue)$ pairs; for example, $inputkey$ could be the IP address of a web page (as a string) and $inputvalue$ could be the contents of that web page (all as one string).

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

If there are R Reduce Workers, the $outputkey$ 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 $(outputkey, list(mapvalue))$ 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

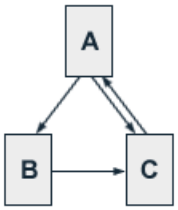
The *PageRank* algorithm used by Google expresses the ranking of a web page in terms of two components:

- a base value, $(1 - d)$, usually 0.15
- $d * \Sigma_{i \in links} PR_i/n_i$ 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:[51]

```
     A             B             C
1.00000000    1.00000000    1.00000000
1.00000000    0.57500000    1.42500000
1.36125000    0.57500000    1.06375000
1.05418750    0.72853125    1.21728125
1.18468906    0.59802969    1.21728125
1.18468906    0.65349285    1.16181809
1.13754537    0.65349285    1.20896178
1.17761751    0.63345678    1.18892571
1.16058685    0.65048744    1.18892571
1.16058685    0.64324941    1.19616374
1.16673918    0.64324941    1.19001141
1.16150970    0.64586415    1.19262615
1.16373223    0.64364162    1.19262615
  ...
1.16336914    0.64443188    1.19219898
```

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.

[51] 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.