Artificial Intelligence: Physics Problem Solving

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

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

Artificial Intelligence as Science

Intelligence should be placed in the context of biology:

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

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

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

Goals of AI:

- understand the computations required for intelligent behavior
- produce computer systems that exhibit intelligence.

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

AI as Engineering

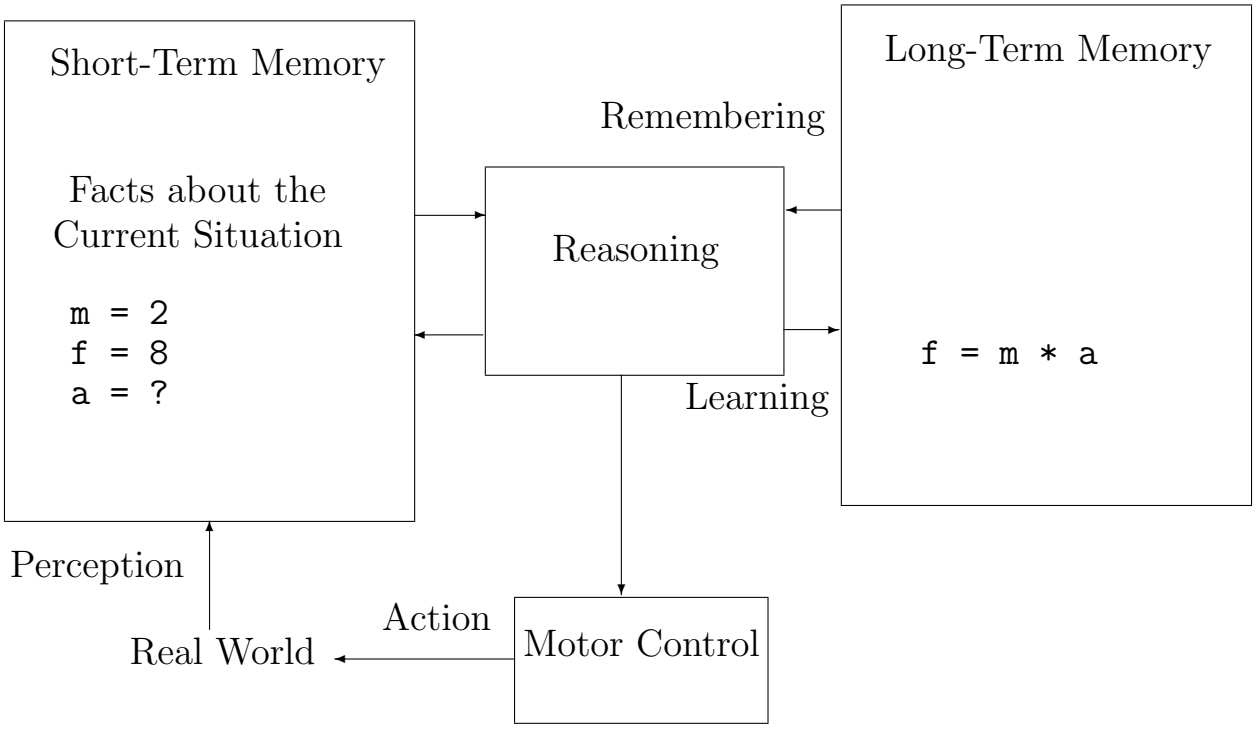
How can we make computer systems more intelligent?

- *Perception* to get input directly from the real world.
- *Autonomy* to perform tasks without a human operator.
- *Flexibility* to deal with a changing environment.
- *Ease of use*: computers that are able to understand what the user wants from limited instructions in natural languages such as English.
  - `Q:` *How can I get more file space?*
  - `A:` *Type in* `rm *`
- *Learning* from experience.

Areas of Success in AI

- Perception
  - Vision: drive vehicle across the desert, in traffic
  - Understanding speech
- Robotics
- Natural Language Processing
  - Natural Language Understanding
  - Machine Translation
- Expert Systems: capture human expertise to create a program that can diagnose blood infections
- Theorem Proving: prove that the Divide instruction works correctly
- Symbolic Mathematics: algebra, calculus
- Game Playing: world chess champion

Simple Model of Intelligence



Neuroscience tries to understand how these actually are implemented in the brain.

A.I. tries to invent ways in which these things could be implemented in computers.

You *vs.* Your PC

	Human	**PC**
Speed	1,000 Hz	2,000,000,000 Hz
Parallel	very	2
RAM	7	1,000,000,000
Disk	10,000,000,000	500,000,000,000
Content Addressable	Yes	No

Humans think slowly – a million times slower than computers.

The only thing that is much good about the human brain is the large content-addressable memory.

Fundamental AI Problems

- **Knowledge Representation:** How can facts about the world be represented in the computer?
- **Reasoning:** How can new facts be derived from existing facts?
- **Search:** Trying different approaches until you find one that works: a "weak method" for finding a solution to a problem when no direct method exists.
- **Learning:** Given what we observe, what should we learn?

Symbolic Representation

Most of the reasoning that people do is non-numeric. AI programs reason with *symbols* that represent objects and relationships in the real world:

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

Examples of symbolic processing:

- Understanding English:

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

- Reasoning based on general principles:

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

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

Lisp

Lisp is the main language used in AI research:

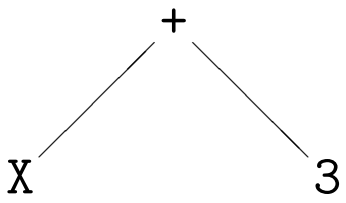
- *Symbols* are a built-in data type.
- *Recursion* with
- *Dynamic type checking*
- Programs and data are *the same thing.*

Parens go outside a function call: `(sqrt 2)`

Everything is a function: `(+ x 3)`

Lisp data:

- *Atoms:* `3` `X` `MASS`
- *Structures:* `(+ X 3)`
  - a *list* of three things: `+`, `X` and `3`
  - a *tree*



  - Lisp *code:* add `X` and `3`
  - an *algebraic expression*, $x + 3$

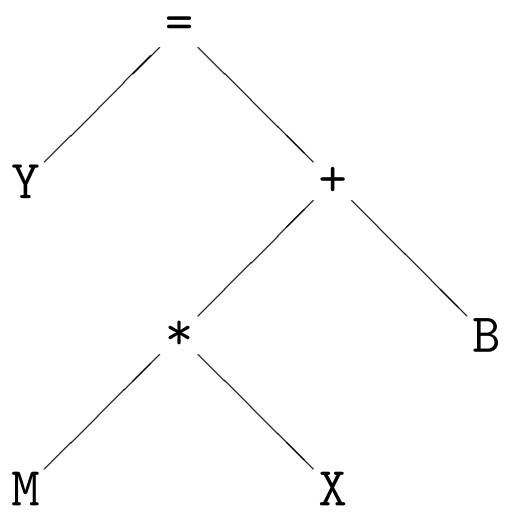
Solving Equations

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

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

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

This can be thought of as the tree:



Recursion and Search

Search allows a problem to be solved in small steps:

- If the problem is already solved, stop.
- Otherwise, try doing some operation, then try to solve the resulting subproblem.

Solving an Equation by Search

We can solve simple equations in this way:

- If only the desired variable is on the left, succeed.
- If only the desired variable is on the right, reverse the equation and succeed.
- If only an undesired variable is on the right, fail.
- Otherwise, try using an algebraic law to rewrite the equation, eliminating the top operator on the right; usually, there are two possibilities to try. Then try to solve the resulting equation; if either attempt succeeds, return that answer.

Patterns

A *pattern* is a Lisp tree structure with variable parts, indicated by `?`. We can do two things with patterns:

- **Match:** see if an input matches a pattern, and if so, return *bindings* that give the variable values that match:

```
>(match '(+ ?x ?y)
        '(+ z 3))

((?Y . 3) (?X . Z))
```

- **Substitute:** substitute bindings into a pattern:

```
>(sublis '((a . 3) (m . 4))
         '(= f (* m a)) )

(= F (* 4 3))
```

Doing Algebra with Patterns

We can express an algebraic *rewrite rule* with a pair of patterns: an equation that matches the first pattern can be rewritten as the second pattern.

- **Algebra:** if $x = y + z$, then $x - y = z$.
- **Patterns:**

```
>( (= ?x (+ ?y ?z))   (= (- ?x ?y) ?z) )
```

```
>(defun transform (equation pair)
   (sublis (match (first pair) equation)
           (second pair) ) )
```

```
>(transform '(= v (+ 3 w))
            '( (= ?x (+ ?y ?z))
               (= (- ?x ?y) ?z) ) )
```

```
(= (- V 3) W)
```

Equation Solving Program

```
(defun solve (eqn var)
  (if (eq (second eqn) var)           ; (= var ...)
      eqn
      (if (eq (third eqn) var)        ; (= ... var)
          (list '= (third eqn) (second eqn))
          (if (consp (third eqn))     ; if op on right
              (trypats eqn var allpats) ) ) ) )

(defun trypats (eqn var pats)
  (and pats
       (or (and (match (first (first pats)) eqn)
                (solve (sublis (match (first (first pats))
                                      eqn)
                               (second (first pats)))
                       var))
           (trypats eqn var (rest pats)) ) ) )

(defvar allpats '(
  ( (= ?x (+ ?y ?z))   (= (- ?x ?y) ?z) )
  ( (= ?x (+ ?y ?z))   (= (- ?x ?z) ?y) )
  ( (= ?x (* ?y ?z))   (= (/ ?x ?y) ?z) )
  ( (= ?x (* ?y ?z))   (= (/ ?x ?z) ?y) )
  ( (= ?x (- ?y ?z))   (= (- ?y ?x) ?z) )
  ( (= ?x (- ?y ?z))   (= (+ ?x ?z) ?y) )
  ( (= ?x (/ ?y ?z))   (= (/ ?y ?x) ?z) )
  ( (= ?x (/ ?y ?z))   (= (* ?x ?z) ?y) ) ))
```

Examples of Equation Solving

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

(= X 3)

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

(= X 3)

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

  1> (SOLVE (= Y (+ X B)) X)
    2> (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))
```

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:
  `((F . 8) (M . 2))`
- a variable whose value is desired: `A`

Solving a Set of Equations by Search

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

  ```
  (= F (* M A))
  ```

- Solve the equation for that variable.

  ```
  (= A (/ F M))
  ```

- Substitute the known values into the right-hand side of the solved equation (Lisp function `sublis`).

  ```
  (= A (/ 8 2))
  ```

- Evaluate the resulting expression (Lisp function `eval`) to give the value of the new variable.

  ```
  (= A 4)
  ```

- Keep trying until you get the value of the variable you want (or quit if you stop making any progress).

English

English is a context-free language (more or less); that is, English has a structure in which phrases are composed of other phrases. This corresponds naturally to a tree.

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 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 [1]

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

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

[1] file `expenglish.lsp`

Generating English

```
%lisp
>(load "/projects/cs375/expenglish.lsp")

>(exp->english 'x)

(X)

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

(THE SUM OF X AND Y)

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

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

Parsing as Search

Parsing attempts to understand a sentence by constructing a tree that represents the meaning of the sentence; it can be done as search using a grammar.

A grammar describes how a sentence or phrase is made up of components in sequence; this forms a *parse tree*.

```
(s -> (what is (property) of (object))

      (list 'calculate $3 $5) )
```

Search:

- For each item in a rule,
  - If the item is a word, see if the input is that word.
  - If the item is a word category, see if the input is in that category.
  - If the item is a phrase, call the program that parses that phrase.
- A rule may fail, or it may succeed and return a value. The value of a compound rule is made from the values of its components.
- If there are multiple rules for a phrase, try each of them in order.

A Grammar for Math and Physics

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

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

Solving Physics Story Problems

By combining the techniques we have discussed, 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
```