CSC 550: Introduction to Artificial Intelligence

Spring 2004

AI programs

- Eliza
  - ➢ knowledge representation
  - ➢ pattern-matching
- Predicate Calculus
- logical deduction
  - ➢ Prolog
  - ➢ logic programming

Weizenbaum's Eliza

In 1965, Joseph Weizenbaum wrote a program called Eliza

- intended as a critique on Weak AI researchers of the time
- utilized a variety of programming tricks to mimic a Rogerian psychotherapist

```
USER:  Men are all alike.
ELIZA: In what way.
USER:  They are always bugging us about one thing or another.
ELIZA: Can you think of a specific example?
USER:  Well, my boyfriend made me come here.
ELIZA: Your boyfriend made you come here.
USER:  He says I am depressed most of the time.
ELIZA: I am sorry to hear you are depressed.
.
.
.
```

Eliza's knowledge consisted of a set of rules

- each rule described a possible pattern to the user's entry & possible responses
- for each user entry, the program searched for a rule that matched
  then randomly selected from the possible responses

- to make the responses more realistic, they could utilize phrases from the user's entry

Eliza rules in Scheme

each rule is written as a list -- SURPRISE! :

```
(USER-PATTERN1
  RESPONSE-PATTERN1-A RESPONSE-PATTERN1-B … )
```

`(VAR X)` specifies a variable – part of pattern that can match any text

```
(define ELIZA-RULES
  '((((VAR X) hello (VAR Y))
     (how do you do.  please state your problem))
    (((VAR X) computer (VAR Y))
     (do computers worry you)
     (what do you think about machines)
     (why do you mention computers)
     (what do you think machines have to do with your problem))
    (((VAR X) name (VAR Y))
     (i am not interested in names))
     .
     .
     .
    (((VAR X) are you (VAR Y))
     (why are you interested in whether i am (VAR Y) or not)
     (would you prefer it if i weren't (VAR Y))
     (perhaps i am (VAR Y) in your fantasies))     .
     .
     .
     .
    (((VAR X))
     (very interesting)
     (i am not sure i understand you fully)
     (what does that suggest to you)
     (please continue)
     (go on)
     (do you feel strongly about discussing such things))))
```


Eliza code

```
(define (eliza)
  (begin (display 'Eliza>)
         (display (apply-rule ELIZA-RULES (read)))
         (newline)
         (eliza)))
```

top-level function:
- display prompt,
- read user entry
- find, apply & display matching rule
- recurse to handle next entry

```
(define (apply-rule rules input)
  (let ((result (pattern-match (caar rules) input '())))
    (if (equal? result 'failed)
        (apply-rule (cdr rules) input)
        (apply-substs (switch-viewpoint result)
              (random-ele (cdar rules))))))
```

to find and apply a rule
- pattern match with variables
- if no match, recurse on cdr
- otherwise, pick a random response & switch viewpoint of words like me/you

e.g.,

```
> (pattern-match '(i hate (VAR X)) '(i hate my computer) '())
(((var x) <-- my computer))


> (apply-rule '(((i hate (VAR X)) (why do you hate (VAR X)) (calm down))
                ((VAR X) (please go on) (say what)))
              '(i hate my computer))
(why do you hate your computer)
```

Eliza code (cont.)

```
(define (apply-substs substs target)
  (cond ((null? target) '())
        ((and (list? (car target)) (not (variable? (car target))))
         (cons (apply-substs substs (car target))
               (apply-substs substs (cdr target))))
        (else (let ((value (assoc (car target) substs)))
                (if (list? value)
                    (append (cddr value)
                            (apply-substs substs (cdr target)))
                    (cons (car target)
                          (apply-substs substs (cdr target)))))))))


(define (switch-viewpoint words)
  (apply-substs '((i <-- you) (you <-- i) (me <-- you)
                  (you <-- me) (am <-- are) (are <-- am)
                  (my <-- your) (your <-- my)
                  (yourself <-- myself) (myself <-- yourself))
                words))
```

e.g.,

```
> (apply-substs '(((VAR X) <-- your computer)) '(why do you hate (VAR X)))
(why do you hate your computer)


> (switch-viewpoint '(((VAR X) <-- my computer)))
(((var x) <-- your computer))
```


In-class exercise

modifications to Eliza?

additions to Eliza?

Symbolic AI

"Good Old-Fashioned AI" relies on the *Physical Symbol System Hypothesis*:

intelligent activity is achieved through the use of
- symbol patterns to represent the problem
- operations on those patterns to generate potential solutions
- search to select a solution among the possibilities

an AI representation language must
- handle qualitative knowledge
- allow new knowledge to be inferred from facts & rules
- allow representation of general principles
- capture complex semantic meaning
- allow for meta-level reasoning

e.g., Predicate Calculus

Predicate calculus: syntax

the predicate calculus (PC) is a language for representing knowledge, amenable to reasoning using inference rules

the *syntax* of a language defines the form of statements
- the building blocks of statements in the PC are terms and predicates
  terms denote objects and properties
  `dave  redBlock  happy  X  Person`
  predicates define relationships between objects
  `mother/1  above/2  likes/2`
- sentences are statements about the world

```
male(dave)    parent(dave, jack)    ¬happy(chris)
parent(dave, jack) ∧ parent(dave, charlie)
happy(chris) ∨ ¬happy(chris)
healthy(kelly) ⇒ happy(kelly)
∀X (healthy(X) ⇒ happy(X))
∃X parent(dave, X)
```

Predicate calculus: semantics

the *semantics* of a language defines the meaning of statements

- must focus on a particular *domain* (universe of objects)
  terms are assigned values from the domain
  predicate symbols are assigned mappings from arguments to true/false
- an *interpretation* assigns true/false value to sentences based on domain mappings

e.g. DOMAIN: students in this class

mike, sean, pengynan, ken, cedric, richard : mapped to actual people

undergrad/1 & grad/1: map a student to true if right classification, else false
male/1 & female/1: map a student to true if right gender, else false

undergrad(sean)	male(ken)	;;; true if predicate maps objects to true
¬grad(mike)	¬female(cedric)	;;; ¬X is true if X is false
undergrad(pengynan) ∧ undergrad(richard)		;;; X ∧ Y is true if X is true and Y is true
undergrad(mike) ∨ grad(sean)		;;; X ∨ Y is true if X is true or Y is true
∀X (male(X) ∨ female(X))		;;; ∀X S is true if S is true for all X
∃G undergrad(G)		;;; ∀X S is true if S is true for all X
∀X (undergrad(X) ⇒ ¬ grad(X))		;;; X ⇒ Y is true if

Predicate calculus: logical deduction

the semantics of the predicate calculus provides a basis for a formal theory of logical inference

- S is a *logical consequence* of a $\{S_1, \ldots, S_n\}$ if
  every interpretation that makes $\{S_1, \ldots, S_n\}$ true also makes S true

a proof procedure can automate the derivation of logical consequences

- a proof procedure is a combination of inference rules and an algorithm for applying the rules to generate logical consequences
- example inference rules:

Modus Ponens: if S_1 and $S_1 \Rightarrow S_2$ are true, then infer S_2

And Elimination: if $S_1 \wedge S_2$ is true, then infer S_1 and infer S_2

And Introduction: if S_1 and S_2 are true, then infer $S_1 \wedge S_2$

Universal Instantiation: if $\forall X\ p(X)$ is true, then infer p(a) for any a

Logic programming

the Prolog programming language is based on logical deduction

- programs are collections of facts and rules of the form: $P_1 \wedge \ldots \wedge P_n \Rightarrow C$
- a simple but efficient proof procedure is built into the Prolog interpreter
  repeatedly use rules to reduce goals to subgoals, eventually to facts

logic programming: computation = logical deduction from program statements

```
FACTS:   itRains
         isCold
RULES:   itSnows ⇒ isCold
         itRains ⇒ getWet
         isCold ∧ getWet ⇒ getSick

GOAL:    getSick
           ⇓      according to first rule, can reduce getSick
         isCold ∧ getWet
           ⇓      isCold is a fact
         getWet
           ⇓      according to second rule, can reduce get_wet
         itRains
           ⇓      itRains is a fact
   DONE (getSick is a logical consequence)
```

```
FACTS:   human(socrates)
RULES:   ∀P (human(P) ⇒ mortal(P))

GOAL:    mortal(socrates)
           ⇓      according to first rule (when
                  P = socrates) , can reduce
                  mortal(socrates)
         human(socrates)
           ⇓      is a fact

   DONE (human(socrates) is a
   logical consequence)
```


Prolog deduction in Scheme

we can implement a simplified version of Prolog's deduction in Scheme

- represent facts and rules as lists – SURPRISE!
- collect all knowledge about a situation (facts & rules) in a single list

```
(define KNOWLEDGE '((itRains <-- )
                    (isCold <-- )
                    (isCold <-- itSnows)
                    (getSick <-- isCold getWet)
                    (getWet <-- itRains)))
```

- note: facts are represented as rules with no premises
- for simplicity: we will not consider variables (but not too difficult to extend)

- want to be able to deduce new knowledge from existing knowledge

```
> (deduce 'itRains KNOWLEDGE)
#t

> (deduce 'getWet KNOWLEDGE)
#t

> (deduce 'getSick KNOWLEDGE)
#t

> (deduce '(getSick itSnows) KNOWLEDGE)
#f
```

Prolog deduction in Scheme (cont.)

our program will follow the approach of Prolog

- start with a goal or goals to be derived (i.e., shown to follow from knowledge)
- pick one of the goals (e.g., leftmost), find a rule that reduces it to subgoals, and replace that goal with the new subgoals

```
(getSick) → (isCold getWet)          ;;; via (getSick <-- isCold getWet)
          → (getWet)                 ;;; via (isCold <-- )
          → (itRains)                ;;; via (getWet <-- itRains)
          →                          ;;; via (itRains <-- )
```

- finding a rule that matches is not hard, but there may be more than one
  QUESTION: how do you choose?

  ANSWER: you don't! you have to be able to try every possibility

- must maintain a list of goal lists, any of which is sufficient to derive original goal

```
((getSick)) → ((isCold getWet))           ;;; reduces only goal list
            → ((itSnows getWet)(getWet))  ;;; introduces 2 goal lists
                                          ;;;   since 2 matches
            → ((getWet))                  ;;; first goal list is dead end
            → ((itRains))                 ;;;   so go to second goal list
            → (())                        ;;; no remaining goals in list
```


Deduction in Scheme

`deduce-any` takes a list of goal lists, returns #t if any goal list is derivable, else #f

```
(deduce-any '((itRains))) → #t
(deduce-any '((getSick itSnows)(itRains))) → #t
(deduce-any '((getSick itSnows)(itSnows))) → #f
```

```
(define (deduce goal known)

  (define (deduce-any goal-lists)
    (cond ((null? goal-lists) #f)
          ((null? (car goal-lists)) #t)
          (else (deduce-any (append (extend (car goal-lists) known)
                                    (cdr goal-lists))))))
```

`exend` reduces a goal list every possible way

```
(extend '(isCold getWet) KNOWLEDGE) →
   ((getWet) (itSnows getWet))
```

```
  (define (extend anded-goals known-step)
    (cond ((null? known-step) '())
          ((equal? (car anded-goals) (caar known-step))
           (cons (append (cddar known-step) (cdr anded-goals))
                 (extend anded-goals (cdr known-step))))
          (else (extend anded-goals (cdr known-step)))))
```

```
  (if (list? goal)
      (deduce-any (list goal))
      (deduce-any (list (list goal)))))
```

`deduce` first turns user's goal into a list of goal lists,

```
(deduce 'itRains X)  calls
   (deduce-any '((itRains)) X)
(deduce '(getSick itSnows) X)  calls
   (deduce-any '((getSick itSnows)) X)
```

Next week…

AI as search

- problem states, state spaces
- uninformed search strategies
  - depth first search
  - depth first search with cycle checking
  - breadth first search
  - iterative deepening

Read Chapter 3

Be prepared for a quiz on

- this week's lecture (moderately thorough)
- the reading (superficial)