

CSC 550: Introduction to Artificial Intelligence

Fall 2004

Scheme programming

- S-expressions: atoms, lists, functional expressions, evaluation
- primitive functions: arithmetic, predicate, symbolic, equality, high-level
- defining functions: define
- special forms: if, cond
- recursion: tail vs. full
- let expressions, I/O

AI applications

- Eliza
- logical deduction

1

Functional programming

1957: FORTRAN was first high-level programming language

- mathematical in nature, efficient due to connection with low-level machine
- not well suited to AI research, which dealt with symbols & dynamic knowledge

1959: McCarthy at MIT developed LISP (List Processing Language)

- symbolic, list-oriented, transparent memory management
- instantly popular as the language for AI
- separation from the underlying architecture tended to make it less efficient (and usually interpreted)

1975: Scheme was developed at MIT

- clean, simple subset of LISP
- static scoping, first-class functions, efficient tail-recursion, ...

2

Obtaining a Scheme interpreter

many free Scheme interpreters/environments exist

- Dr. Scheme is an development environment developed at Rice University
- contains an integrated editor, syntax checker, debugger, interpreter
- Windows, Mac, and UNIX versions exist

- can download a personal copy from

`http://download.plt-scheme.org/drscheme/`

be sure to set Language to "Textual (MzScheme, includes R5RS)"

3

LISP/Scheme in a nutshell

LISP/Scheme is very simple

- only 2 kinds of data objects
 1. atoms (identifiers/constants) `robot green 12.5`
 2. lists (of atoms and sublists) `(1 2 3.14)`
`(robot (color green) (weight 100))`

Note: lists can store different types, not contiguous, not random access

- functions and function calls are represented as lists (i.e., program = data)

```
(define (square x) (* x x))
```

- all computation is performed by applying functions to arguments, also as lists

```
(+ 2 3)           evaluates to 5  
(square 5)       evaluates to 25  
(car (reverse '(a b c))) evaluates to c
```

4

Functional expressions

computation in a functional language is via function calls (also S-exprs)

```
(FUNC ARG1 ARG2 . . . ARGn)
```

```
(+ 3 (* 4 2))
```

```
(car '(a b c))
```

quote specifies data, not to be evaluated further
(numbers are implicitly quoted)

evaluating a functional expression:

- function/operator name & arguments are evaluated in unspecified order
note: if argument is a functional expression, evaluate recursively
- the resulting function is applied to the resulting values

```
(car '(a b c))
```

evaluates to primitive function

evaluates to list (a b c) : ' terminates recursive evaluation

so, primitive car function is called with argument (a b c)

7

Arithmetic primitives

predefined functions:

```
+ - * /  
quotient remainder modulo  
max min abs gcd lcm expt  
floor ceiling truncate round  
= < > <= >=
```

- many of these take a variable number of inputs

```
(+ 3 6 8 4)           → 21  
(max 3 6 8 4)        → 8  
(= 1 (-3 2) (* 1 1)) → #t  
(< 1 2 3 4)           → #t
```

- functions that return a true/false value are called *predicate functions*
zero? positive? negative? odd? even?

```
(odd? 5)              → #t  
(positive? (- 4 5))  → #f
```

8

Data types in LISP/Scheme

LISP/Scheme is loosely typed

- types are associated with values rather than variables, bound dynamically

numbers can be described as a hierarchy of types



integers and rationals are *exact* values, others can be *inexact*

- arithmetic operators preserve exactness, can explicitly convert

```
(+ 3 1/2)      → 7/2
(+ 3 0.5)     → 3.5

(inexact->exact 4.5) → 9/2
(exact->inexact 9/2) → 4.5
```

9

Symbolic primitives

predefined functions:

```
car  cdr  cons
list list-ref length member
reverse append equal?
```

```
(list 'a 'b 'c)      → (a b c)
(list-ref '(a b c) 1) → b
(member 'b '(a b c)) → (b c)
(member 'd '(a b c)) → #f
(equal? 'a (car '(a b c))) → #t
```

- `car` and `cdr` can be combined for brevity

```
(cadr '(a b c)) ≡ (car (cdr '(a b c))) → b
```

```
cadr  returns 2nd item in list
caddr returns 3rd item in list
caddr returns 4th item in list (can only go 4 levels deep)
```

10

Defining functions

can define a new function using `define`

- a function is a mapping from some number of inputs to a single output

```
(define (NAME INPUTS) OUTPUT_VALUE)
```

```
(define (square x)
  (* x x))

(define (next-to-last arblast)
  (cadr (reverse arblast)))

(define (add-at-end1 item arblast)
  (reverse (cons item (reverse arblast))))

(define (add-at-end2 item arblast)
  (append arblast (list item)))
```

```
(square 5) → 25

(next-to-last '(a b c d))
→ c

(add-at-end1 'x '(a b c))
→ '(a b c x)

(add-at-end2 'x '(a b c))
→ '(a b c x)
```

11

Examples

```
(define (miles->feet mi)
  IN-CLASS EXERCISE
)
```

```
(miles->feet 1) → 5280
```

```
(miles->feet 1.5) → 7920.0
```

```
(define (replace-front new-item old-list)
  IN-CLASS EXERCISE
)
```

```
(replace-front 'x '(a b c))
→ (x b c)
```

```
(replace-front 12 '(foo))
→ (12)
```

12

Conditional evaluation

can select alternative expressions to evaluate

```
(if TEST TRUE_EXPRESSION FALSE_EXPRESSION)

(define (my-abs num)
  (if (negative? num)
      (- 0 num)
      num))

(define (wind-chill temp wind)
  (if (<= wind 3)
      (exact->inexact temp)
      (+ 35.74 (* 0.6215 temp)
         (* (- (* 0.4275 temp) 35.75) (expt wind 0.16)))))
```

13

Conditional evaluation (cont.)

logical connectives `and`, `or`, `not` can be used

predicates exist for selecting various types

```
symbol?   char?   boolean?  string?   list?    null?
number?   complex? real?     rational? integer?
exact?    inexact?
```

note: an `if`-expression is a *special form*

- is *not* considered a functional expression, doesn't follow standard evaluation rules

```
(if (list? x)
    (car x)
    (list x))
```

test expression is evaluated

- if value is anything but `#f`, first expr evaluated & returned
- if value is `#f`, second expr evaluated & returned

```
(if (and (list? x) (= (length x) 1))
    'singleton
    'not)
```

Boolean expressions are evaluated left-to-right, short-circuited

14

Multi-way conditional

when there are more than two alternatives, can

- nest if-expressions (i.e., cascading if's)
- use the `cond` special form (i.e., a switch)

```
(cond (TEST1 EXPRESSION1)
      (TEST2 EXPRESSION2)
      . . .
      (else EXPRESSIONn))
```

evaluate tests in order

- when reach one that evaluates to "true", evaluate corresponding expression & return

```
(define (compare num1 num2)
  (cond ((= num1 num2) 'equal)
        (> num1 num2) 'greater)
        (else 'less)))

(define (wind-chill temp wind)
  (cond (> temp 50) 'UNDEFINED)
        (<= wind 3) (exact->inexact temp)
        (else (+ 35.74 (* 0.6215 temp)
                  (* (- (* 0.4275 temp) 35.75)
                     (expt wind 0.16))))))
```

15

Examples

```
(define (palindrome? lst)
  IN-CLASS EXERCISE
)
```

```
(palindrome? '(a b b a))
→ #t
```

```
(palindrome? '(a b c a))
→ #f
```

```
(define (safe-replace-front new-item old-list)
  IN-CLASS EXERCISE
)
```

```
(safe-replace-front 'x '(a b c))
→ (x b c)
```

```
(safe-replace-front 'x '())
→ 'ERROR
```

16

Repetition via recursion

pure LISP/Scheme does not have loops

- repetition is performed via recursive functions

```
(define (sum-1-to-N N)
  (if (< N 1)
      0
      (+ N (sum-1-to-N (- N 1)))))
```

```
(define (my-member item lst)
  (cond ((null? lst) #f)
        ((equal? item (car lst)) lst)
        (else (my-member item (cdr lst)))))
```

17

Examples

```
(define (sum-list numlist)
  IN-CLASS EXERCISE
  )
```

```
(sum-list '()) → 0
```

```
(sum-list '(10 4 19 8)) → 41
```

```
(define (my-length lst)
  IN-CLASS EXERCISE
  )
```

```
(my-length '()) → 0
```

```
(my-length '(10 4 19 8)) → 4
```

18

Tail-recursion vs. full-recursion

a tail-recursive function is one in which the recursive call occurs last

```
(define (my-member item lst)
  (cond ((null? lst) #f)
        ((equal? item (car lst)) lst)
        (else (my-member item (cdr lst)))))
```

a full-recursive function is one in which further evaluation is required

```
(define (sum-1-to-N N)
  (if (< N 1)
      0
      (+ N (sum-1-to-N (- N 1)))))
```

full-recursive call requires memory proportional to number of calls

→ limit to recursion depth

tail-recursive function can reuse same memory for each recursive call

→ no limit on recursion

19

Tail-recursion vs. full-recursion (cont.)

any full-recursive function can be rewritten using tail-recursion

- often accomplished using a help function with an accumulator
- since Scheme is statically scoped, can hide help function by nesting

```
(define (factorial N)
  (if (zero? N)
      1
      (* N (factorial (- N 1)))))
```

value is computed "on the way up"

```
(factorial 2)
  ↑
(* 2 (factorial 1))
  ↑
(* 1 (factorial 0))
  ↑
1
```

```
(define (factorial N)
  (define (factorial-help N value-so-far)
    (if (zero? N)
        value-so-far
        (factorial-help (- N 1)
                        (* N value-so-far))))
  (factorial-help N 1))
```

value is computed "on the way down"

```
(factorial-help 2 1)
  ↓
(factorial-help 1 (* 2 1))
  ↓
(factorial-help 0 (* 1 2))
  ↓
2
```

20

Structuring data

an association list is a list of "records"

- each record is a list of related information, keyed by the first field

```
(define NAMES '((Smith Pat Q)
                (Jones Chris J)
                (Walker Kelly T)
                (Thompson Shelly P)))
```

note: can use `define` to
create "global constants"
(for convenience)

- can access the record (sublist) for a particular entry using `assoc`

```
> (assoc 'Smith NAMES)      > (assoc 'Walker NAMES)
(Smith Pat Q)                (Walker Kelly T)
```

- `assoc` traverses the association list, checks the `car` of each sublist

```
(define (my-assoc key assoc-list)
  (cond ((null? assoc-list) #f)
        ((equal? key (caar assoc-list)) (car assoc-list))
        (else (my-assoc key (cdr assoc-list)))))
```

21

Association lists

to access structured data,

- store in an association list with search key first
- access via the search key (using `assoc`)
- use `car/cdr` to select the desired information from the returned record

```
(define MENU '((bean-burger 2.99)
               (tofu-dog 2.49)
               (fries 0.99)
               (medium-soda 0.79)
               (large-soda 0.99)))
```

```
> (cadr (assoc 'fries MENU))
0.99

> (cadr (assoc 'tofu-dog MENU))
2.49
```

```
(define (price item)
  (cadr (assoc item MENU)))
```

22

assoc example

consider a more general problem: determine price for an entire meal

- represent the meal order as a list of items,
e.g., (tofu-dog fries large-soda)
- use recursion to traverse the meal list, add up price of each item

```
(define (meal-price meal)
  (if (null? meal)
      0.0
      (+ (price (car meal)) (meal-price (cdr meal)))))
```

```
> (meal-price '())
0.0

> (meal-price '(large-soda))
0.99

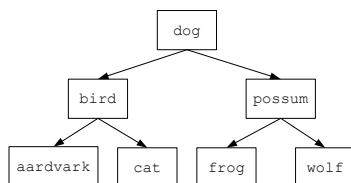
> (meal-price '(tofu-dog fries large-soda))
4.47
```

23

Non-linear data structures

note: can represent non-linear structures using lists

e.g. trees



```
(dog
 (bird (aardvark () ()) (cat () ()))
 (possum (frog () ()) (wolf () ())))
```

- empty tree is represented by the empty list: ()
- non-empty tree is represented as a list: (ROOT LEFT-SUBTREE RIGHT-SUBTREE)
- can access the the tree efficiently

```
(car TREE)    →  ROOT
(cadr TREE)   →  LEFT-SUBTREE
(caddr TREE)  →  RIGHT-SUBTREE
```

24

Tree routines

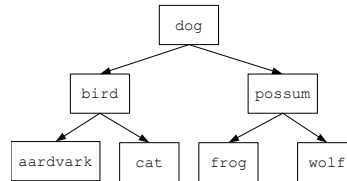
```
(define TREE1
  ' (dog
    (bird (aardvark () ()) (cat () ()))
    (possum (frog () ()) (wolf () ())))
```

```
(define (empty? tree)
  (null? tree))
```

```
(define (root tree)
  (if (empty? tree)
      'ERROR
      (car tree)))
```

```
(define (left-subtree tree)
  (if (empty? tree)
      'ERROR
      (cadr tree)))
```

```
(define (right-subtree tree)
  (if (empty? tree)
      'ERROR
      (caddr tree)))
```



25

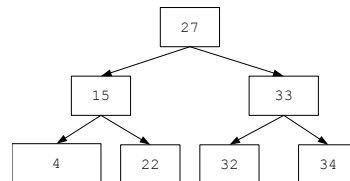
Tree searching

note: can access root & either subtree in constant time

→ can implement binary search trees with $O(\log N)$ access

binary search tree: for each node, all values in left subtree are \leq value at node
all values in right subtree are $>$ value at node

```
(define (bst-contains? bstree sym)
  (cond ((empty? tree) #f)
        ((= (root tree) sym) #t)
        ((> (root tree) sym) (bst-contains? (left-subtree tree) sym))
        (else (bst-contains? (right-subtree tree) sym))))
```



note: recursive nature of trees makes them ideal for recursive traversals

26

Finally, variables!

Scheme does provide for variables and destructive assignments

```
> (define x 4)           define creates and initializes a variable

> x
4

> (set! x (+ x 1))      set! updates a variable

> x
5
```

since Scheme is statically scoped, can have global variables

- destructive assignments destroy the functional model
- for efficiency, Scheme utilizes structure sharing – messed up by set!

27

Let expression

fortunately, Scheme provides a "clean" mechanism for creating variables to store (immutable) values

```
(let ((VAR1 VALUE1)
      (VAR2 VALUE2)
      . . .
      (VARn VALUEn))
  EXPRESSION)
```

let expression introduces a new environment with variables (i.e., a block)

good for naming a value (don't need set!)

same effect could be obtained via help function

game of craps:

- if first roll is 7, then WINNER
- if first roll is 2 or 12, then LOSER
- if neither, then first roll is "point"
 - keep rolling until get 7 (LOSER) or point (WINNER)

```
(define (craps)
  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (cond ((= next-roll 7) 'LOSER)
            ((= next-roll point) 'WINNER)
            (else (roll-until point)))))
  (let ((roll (+ (random 6) (random 6) 2)))
    (cond ((or (= roll 2) (= roll 12)) 'LOSER)
          ((= roll 7) 'WINNER)
          (else (roll-until roll)))))
```

28

Scheme I/O

to see the results of the rolls, could append rolls in a list and return

or, bite the bullet and use non-functional features

- `display` displays S-expr (`newline` yields carriage return)
- `read` reads S-expr from input
- `begin` provides sequencing (for side effects), evaluates to last value

```
(define (craps)

  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (begin (display "Roll: ") (display next-roll) (newline)
             (cond ((= next-roll 7) 'LOSER)
                   ((= next-roll point) 'WINNER)
                   (else (roll-until point))))))

  (let ((roll (+ (random 6) (random 6) 2)))
    (begin (display "Point: ") (display roll) (newline)
           (cond ((or (= roll 2) (= roll 12)) 'LOSER)
                 ((= roll 7) 'WINNER)
                 (else (roll-until roll))))))
```

29

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

30

Eliza rules in Scheme

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

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

```
(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))))
```

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

31

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 (cдар 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)
```

32

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)
```

33

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

34

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)
```

```
 $\forall X$  (healthy(X) --> happy(X))
```

```
 $\exists X$  parent(dave, X)
```

35

Predicate calculus: semantics and logical consequence

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

- must focus on a particular *domain* (universe of objects)
- an *interpretation* assigns true/false value to sentences within that domain

S is a *logical consequence* of a $\{S_1, \dots, S_n\}$ if
every interpretation that makes $\{S_1, \dots, S_n\}$ 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 $(S_1 \ \&\& \ S_2)$ is true, then infer S_1 and infer S_2

And Introduction: if S_1 and S_2 are true, then infer $(S_1 \ || \ S_2)$

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

36

Logical deduction

```
FACTS: (F1) itRains
        (F2) isCold
RULES: (R1) itSnows --> isCold
        (R2) itRains --> getWet
        (R3) isCold && getWet --> getSick
```

```
↓      Modus Ponens using R2 and F1
getWet
↓      And Introduction using F2 and new fact
isCold && getWet
↓      Modus Ponens using R3 and new fact
getSick
```

∴ getSick is a logical consequence

```
FACTS: (F1) human(socrates)
RULES: (R1) ∀P (human(P) --> mortal(P))
```

```
↓      Universal Instantiation of R1
human(socrates) --> mortal(socrates)
↓      Modus Ponens using F1 and new rule
mortal(socrates)
```

∴ mortal(socrates) is a logical consequence

logic programming is a popular AI approach in Europe & Asia (e.g., Prolog)

- programs are collections of facts and rules of the form: $P_1 \ \&\& \ \dots \ \&\& \ P_n \ \rightarrow \ C$
- an interpreter works backwards from a goal, trying to reach facts

logic programming: computation = logical deduction from program statements

37

Logical deduction in Scheme

we can implement simple logical deduction in Scheme

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

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

- 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
```

38

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 ((isCold getWet) --> getSick)
           → (true getWet)       ;;; via ((true) --> isCold)
           → (getWet)           ;;; true is assumed
           → (itRains)         ;;; via ((itRains) --> getWet)
           → (true)            ;;; via ((true) --> itRains)
           → ()                ;;; true is assumed
```

- 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))      ;;; 1 rule matches getSick
            → ((itSnows getWet) (true getWet)) ;;; 2 rules match isCold
            → ((true getWet))        ;;; itSnows has no match
            → ((getWet))             ;;; assume true
            → ((itRains))           ;;; 1 rule matches getWet
            → ((true))              ;;; 1 rule matches itRains
            → ()                    ;;; assume true
```

39

Deduction in Scheme

```
(define (deduce goal known)
```

```
(define (apply-rules goal-list known)
  (cond ((null? known) '())
        ((equal? (car goal-list) (caddr known))
         (cons (append (cdr goal-list) (caar known))
               (apply-rules goal-list (cdr known))))
        (else (apply-rules goal-list (cdr known)))))
```

```
(define (deduce-any goal-lists known)
  (if (null? goal-lists)
      #f
      (let ((first-goals (car goal-lists)))
        (cond ((null? first-goals) #t)
              ((equal? (car first-goals) 'true)
               (deduce-any (cons (cdr first-goals) (cdr goal-lists)) known))
              (else (deduce-any (append (cdr goal-lists)
                                         (apply-rules first-goals known))
                                known)))))
```

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

apply-rules reduces a goal list every possible way

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

deduce-any returns #t if any goal list succeeds

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

deduce first turns user's goal into a list of lists, then calls deduce-any to process

40

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)