COM316 Artificial Intelligence

Gary Parker
New London Hall 216
parker@conncoll.edu

Definitions of AI

- Systems that act like humans
- Systems that think like humans
- Systems that think rationally
- Systems that act rationally

Act Like Humans

- Turing Test
  - Natural language processing
  - Knowledge representation
  - Reasoning
  - Learning
- Total Turing test
  - Perception
  - Robotics

Think Like Humans

- Hard to determine
  - Introspection
  - Psychological experiments
- Cognitive Science
  - Computer models to match psych experiments

Think Rationally

- What does that mean?
- Logical reasoning – formal
  - Represent informal knowledge as formal
  - Solving in principle vs in practice
  - Imperfect world

Act Rationally

- Agent
  - Perceives and acts
- Measureable and achievable
- Limited rationality
  - Time constraints

Areas of AI

- Problem solving
- Knowledge representation
- Reasoning
- Planning
- Learning
- Communication
- Perception
- Acting

Symbolic vs Connectionist

- Symbolic
  - Store information as symbols
  - Reason by symbol manipulation
  - Can we think without a language?
- Connectionist
  - Artificial neural networks
  - Simulated neurons

COM316

- Ontogeny recapitulates phylogeny
- AI learning recapitulates AI history
- Solve progressively harder problems
- The same problems solved in the past
- Implement every solution
- Why?

Typical AI Courses

- Wide survey of topics
- Lack continuity and structure
- Confusing -- several different problems
- 70% of time learning detailed AI solutions
- 30% of time implementing a few of these
- Retained: the few that were implemented

Learn the Main Topics

- Search
- Game Playing
- Logic
- Representation
- Production Systems
- Planning
- Learning
- Neural Networks
- Genetic Algorithms

One Problem Per Week

- Tuesday
  - Hand in program and three questions
  - Discuss last week's program
  - Introduce a new problem
- Thursday
  - Hand in and discuss problem solutions
  - Explain standard solution
  - Start implementation program

Scheme

- functional programming language
  - program is expressed as function calls.
- Structured data
  - strings, lists & vectors.
- garbage collection
- procedures are first-class data objects
- processing symbolic information

Identifiers

- Identifiers must be delimited by whitespace, parentheses, double quote or the comment character ';'.
- No length limit.

List Syntax

- Structured forms and list constants are enclosed within parentheses:

```
(a b c)
(* (- x 2) y)
```

- The empty list is written `()`

Boolean values

- Boolean value for true is written **`#t`**
- Boolean value for false is written **`#f`**

- Scheme conditional expressions treat **`#f`** as false and everything else as true.
- Some Scheme implementations treat **`()`** as false.

Whitespace and Comments

- Whitespace includes spaces, tabs and newlines.
- Whitespace length is not important (one space is the same as 100).
- Scheme expressions can span several lines.
- Comments appear between a '**`;`**' and the end of a line.

Some Naming Conventions

- Predicate names end in a **`?`**
  - examples: **`eq?`** **`zero?`** **`string=?`**
  - exceptions: **`=`** **`<`** **`>`** **`<=`** **`>=`**
- Procedures that cause side effects end with **`!`**
  - example: **`set!`**
  - exceptions: **`write`** **`display`** **`read`** **`load`**

Scheme Expressions

- An expression can be a constant data object
  - string, number, symbol, list
  - examples: **`17.5 3/5 "Hello World"`**
- Or an expression can be a *procedure application:*

`(foo 1 2)`

application of the procedure foo.

Procedure Application

- Any procedure application is written in prefix form:
  - **`(`*`procedure_name arg1 arg2`* …`)`**
- Arithmetic operations are not special, they are written in prefix form:

```
(+ 10 20)
(* (* (* 52 7) 24) 60)
```

Precedence

- Using prefix notation means that there are no rules for operator precedence!
- In a nested expression, innermost expressions are computed first (so the programmer determines the order of evaluation, not a bunch of rules).

Lists

- Lists are written as sequences of objects surrounded by parentheses.
- Examples:

```
(1 2 3 4)
("Hi" "Dave")
("One" 1)
(a b c (d e f))
```

List vs. Procedure Application

- A procedure application looks just like a list, so how does scheme known the difference?
- We must tell scheme to treat a list as data rather than as a procedure application.
- The **`quote`** procedure forces a list to be treated as data: **`(quote (+ 3 4))`**
- This is so common we also can use the shorthand notation: **`'(+ 3 4)`**

List manipulation procedures

- **`car`** returns the first element of the list
- **`cdr`** returns the remainder of the list

```
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
```

- **`cons`** constructs lists by adding a new element to the beginning of a list

```
(cons 'a '(b c)) => (a b c)
```

Proper and Improper Lists

- a *list* is a sequence of *pairs*.
- Each *pair's* **`cdr`** is the next *pair*.
- The **`cdr`** of the last pair in a proper list is the empty list.
- If the **`cdr`** of the last pair is not **`()`**, the list is an *improper list*.

Dotted Pair

- Improper lists are printed as a *dotted-pair*.

```
(cons 'a 'b) => (a . b)
(cdr '(a . b)) => b
(cons 'a '(b . c)) => (a b . c)
```

The procedure `list`

- **`list`** takes any number of arguments and builds a proper list:

```
(list 'a 'b 'c) => (a b c)
(list 'a) => (a)
(list ) => ()
(list '(a . b)) => ((a . b))
```

Expression Evaluation

- Procedure application:

  `(procedure_name arg1 arg2 …)`

  find the value of *procedure_name*

  find the value of *arg1, arg2, …*

  In any order!

  apply the value of *procedure_name* to the values of *arg1, arg2, …*

`quote` evaluation

- The rules for evaluation don't work for **`quote`** - why not?
- **`quote`** does not evaluate the subexpression at all.

Variables

- during procedure application the value of variables is determined - so how do we set the value of a variable?
- The **`let`** syntactic form:

```
(let ((x 1))
  (+ x 3))  => 4

(let ((x 1) (y 2))
  (+ x y))  => 3
```

Let Expressions

- Let expressions include a list of variable-value pairs and a sequence of expressions (called the body).
- Let expressions are often used to simplify an expression that contains multiple copies of the same subexpression:

```
(let ((x (+ 2 2))) (* x x))=>16
```

Let Expressions

- Procedure names are no different than any other list element - so we can do this:

```
(let ((f +)) (f 1 2))  => 3
(let ((+ *)) (+ 2 3)) => 6
```

- The values bound by a **`let`** are only *visible* within the body of the **`let`**.

Let can be nested

```
(let ((x 1))
   (let ((x (+ x 1)))
      (+ x x))) => 4
```

The scope of each variable can be determined by the placement within the text of the program == lexical scoping.

Quiz

- What is the value of:

```
(let ((x 9))
  (* x
     (let ((x (/ x 3)))
        (+ x x))))
```

54

Lambda Expressions

- The syntactic form **`lambda`** creates a new procedure:

  (lambda (var …) exp_1 exp_2 …)

- The list of variables are the formal parameters and the expressions are the body.
- The variables in the var list are *bound* and all other variable are called *free*.

Lambda Expression Example

```
((lambda (x) (+ x x))
    (* 3 4))
```

The expression **`(lambda (x) (+ x x))`** defines an unnamed procedure which is then applied to the value **`(* 3 4).`**

The result of this expression is 24.

Lambda Expression Quiz

```
(let ((foo (lambda (x) (+ x 1))))
   (let ((y 3))
       (foo y)))
```

Hint: In this case we are assigning a name to the procedure defined by the lambda expression.

4

define

```
(define x 5)

(* x 3)
```

mult

```
(define multPlus1
  (lambda (x y)
     (+ (* x y) 1)))

(multPlus1 3 4)
```