CS 520: Introduction to Artificial Intelligence

Prof. Louis Steinberg

Lecture 8:
Knowledge-based Agents
Propositional Logic

Review - Lisp

Symbol value vs meaning as function

• Value:
  – Assign with (setf var <expression>)
  – Create local binding with let or let*

    (let* ((var <expr>)
          (var <expr>) ...
          <expr> ...
          )

• Meaning as function
  – Set global meaning with defun
  – Create local scope with flet or labels
Review

Functions as arguments

• Value of #’ foo is function meaning of foo

• (funcall fn-expr argx1 ... argxn) means treat value of fn-expr as a function, call it with values of argx1 ... argxn as arguments

Review - Lexical Scope

• Scope of a variable is syntactic scope of let, defun, etc.

• If a function is created, it carries bindings with it
  – Closure: represents function, points to binding & code
Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
)(let((c1 (counter 4))
     (c2 (counter 0)))
  (list
    (funcall c1 10)
    (funcall c2 3)
    (funcall c2 2)
    (funcall c1 60)))
=> (14 3 5 74)

Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
)(let((c1 (counter 4))
     (c2 (counter 0)))
  (list
    (funcall c1 10)
    (funcall c2 3)
    (funcall c2 2)
    (funcall c1 60)))
=> (14 3 5 74)
Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
)(let((c1 (counter 4))
     (c2 (counter 0)))
  (list
   (funcall c1 10)
   (funcall c2 3)
   (funcall c2 2)
   (funcall c1 60)))
=> (14 3 5 74)

Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
)(let((c1 (counter 4))
     (c2 (counter 0)))
  (list
   (funcall c1 10)
   (funcall c2 3)
   (funcall c2 2)
   (funcall c1 60)))
=> (14 3 5 74)
Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
(let ((c1 (counter 4))
     (c2 (counter 0)))
  (list
    (funcall c1 10)
    (funcall c2 3)
    (funcall c2 2)
    (funcall c1 60)))
=> (14 3 5 74)

stack

Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
(let ((c1 (counter 4))
     (c2 (counter 0)))
  (list
    (funcall c1 10)
    (funcall c2 3)
    (funcall c2 2)
    (funcall c1 60)))
=> (14 3 5 74)

stack
Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
(let((c1 (counter 4))
    (c2 (counter 0)))
  (list
   (funcall c1 10)
   (funcall c2 3)
   (funcall c2 2)
   (funcall c1 60)))
=> (14 3 5 74)
Closures

(defun counter (n)
  #'(lambda (i)
      (setf n (+ n i)))
)(let((c1 (counter 4))
     (c2 (counter 0)))
  (list
    (funcall c1 10)
    (funcall c2 3)
    (funcall c2 2)
    (funcall c1 60)))
=> (14 3 5 74)
Closures

(defun counter (n)
    #'(lambda (i)
        (setf n (+ n i)))
)(let((c1 (counter 4))
    (c2 (counter 0)))
    (list
        (funcall c1 10)
        (funcall c2 3)
        (funcall c2 2)
        (funcall c1 60)))
=> (14 3 5 74)

Programming Techniques

• Data-driven programming
• To write a program for X, extend Lisp to be a good language for X
Quiz

• Write code for some.

(some fn list) calls fn on each element of list until a call returns non-nil, then returns that value.
If all calls return nil, some returns nil.

(some #'oddp '(2 4 5)) => t
(some #'oddp '(2 4 6)) => nil

Mechanisms for Intelligence

Given a problem, formalize it as an instance of a general class of problems, which you know how to solve

• Search
  – Water jugs => graph search
  – N Queens => constraint satisfaction
  – Chess => minimax tree

• Inference
Knowledge-Based Agents

- A knowledge-based agent has knowledge of the world (its environment) and knowledge of its actions.
- Using logical reasoning, it maintains a description of the world, and will try to determine an action that will meet its goals.

Knowledge-based Agent Architecture
Assertions and Queries

- Executive asserts (tells KB) “I sense ....”
- Executive queries (asks KB) “What should I do?”
- Internal assertions and queries of KB
  - Sense and Do
  - State of the world
- E.g.: (sense obstacle +45 degrees)
  (sense obstacle -45 degrees)
  (sense no obstacle 0 degrees)
  => (facing door)
  => (do move-forward)

Why Inference?

- Ideal:
  - Tell agent what is true
    - Fact1, fact2, ...
  - Give agent a goal
  - Agent decides for itself how to use the facts to attain the goal
- Better than giving agent a program
  - More autonomous
  - For given effort we get agent with more general competence.
Logic

• A logic consists of
  – Formal language: sentences
    – Represent statements about the world
  – Inference mechanism:
    \{\text{sentence}_1, \ldots, \text{sentence}_n\} \Rightarrow \text{sentence}_{n+1}
    – Creates new sentences

• Eg, propositional logic

Propositional Logic

• Language:
  – Symbols: a, b, …
  – Operators: ¬, ^, v, =>, ( )

• Semantics:
  – Symbol represents a statement about world that is either true or false
  – Operators combine truth values

a = “it is raining”, b = “the sidewalk is wet”
\text{a} \land \text{b} = “it is raining and the sidewalk is wet”
Models

• Model:
  – An assignment of real-world meaning to the symbols of a logic
• A sentence in a logic may be true or false with respect to a model
• A sentence may be such that it is true in every model: “tautology”
• A sentence may be such that it is true in some model: “satisfiable”
• A sentence may be such that it is false in every model: “contradiction”

Validity of Inference

• An inference rule
  \{old sentences\} \rightarrow new-sentence
  is valid if for every model it is true that
  – If meaning(old-sentences) is true then meaning(new-sentence) must be true
• IE, (old-sentences \rightarrow new-sentence) is a tautology
  – Old-sentences entail new sentence
Soundness and Completeness

- An inference mechanism is **sound** or truth-preserving if it generates from the KB only sentences that are entailed by the KB.
- Tracing through the steps of deriving a sentence from a KB using a sound inference mechanism is called a **proof**.
- If the inference mechanism can derive any sentence that is entailed by the KB, the mechanism is said to be **complete**.

Propositional Inference

- We can test entailment in propositional logic with a truth table

\[(A \land B) \text{ entails } \neg(\neg A \lor \neg B)\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(A ∧ B)</th>
<th>¬(¬A v ¬B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Inference by Rules

• Problem: truth table size is exponential in number of variables
• Solution: test entailment by searching for a sequence of inference rule applications

Rules of Inference for Propositional Logic

Modus Ponens : \( \alpha; \alpha \Rightarrow \beta / \beta \)
And-elimination : \( \alpha_1 \land \alpha_2 \land \ldots \land \alpha_n / \alpha_1 \)
And-introduction …
Or-elimination …
Or-introduction …
Double-negation-elimination
Unit resolution
Resolution
…
Resolution

- Problem: too many rules -> high branching factor for search
- Want single complete rule
  - Resolution