**Massachusetts Institute of Technology**

# 6.034 Artificial Intelligence

Solutions # **7** 6034 Item #

## Problem 1

### Part A

The system accepts the sentence and outputs the answer as if the query were simply `Count the blocks`, so it will simply respond, "There are 8". In fact, any number of `the`s will be accepted by the grammar. The problem lies with the procedure `object-description`:

```
(define object-description
  ;; Consumes, for example, A BIG BLACK HEAVY BLOCK
  (make-tree
    '(branch ((determiner) (object-description --> o) rtn o)
             ((attribute-value-list --> a-v-l)
              (object-description --> o)
              rtn `(db-select ,o '(with ,@a-v-l)))
             ((object --> o) rtn o))))
```

The very first branch of the tree consumes a determiner like *the*, but since it is the very first branch taken, it will also consume an endless succession of determiners.

### Part B

Moving the `(determiner)` subtree to the `main` procedure will only make things worse: now the first branch of the `object-description` procedure is `object-description` itself. This creates a recursive loop that will execute endlessly (until the stack depth for Scheme is exceeded).

(More generally, any top-down parser like this one must either avoid such *left-recursive* forms, where the recursion takes place at the immediate left of a tree expansion, or else use some kind of check to see that the exact same call has not already been made.)

### Part C

The problem here lies in the `identify` procedure and in the control structure the parser uses. Because `identify` places `show me` before `show me all`, and because the parser does not backtrack, `identify` will always consume *show me all* via the `show me` path, gobbling *show me*. Since the next word, *all*, can never match the remaining patterns in `main` that start with identify, queries starting with *show me all* will not be successfully parsed. The fix is to simply interchange the `show me` and `show me all` lines in `identify`. (Note that we have in effect placed a more general pattern before a more specific pattern, so the more specific pattern will never get a chance to match.)

# Problem 2

## Part A

The required changes to the grammar file are shown below. The changes required to handle the first set of questions are minimal. We don't need to change the relational database stub, or any other files except `grammar.scm`. We simply need to add new transition trees to incorporate new attributes like the *author* field.

There are essentially two ways to handle the *sci-fi*, *type*, and *topics* questions. The one displayed below simply makes words like *sci-fi* or *type* synonyms of existing database entries like *fiction* or *category*. A more general, but more radical approach could rewrite the books database as a frame system, with *sci-fi* books being a subclass of *fiction* books (instead of being synonyms, "sci-fi" and "fiction" would bear a subset-superset relation to each other). Then, in order to answer a question like "What are the sci-fi books?", the system could look up the inheritance hierarchy, stopping at sci-fi (alternatively, it could start at the top of an inheritance hierarchy and look down).

```
(define attribute
  ;; Consumes, for example, COLOR
  (make-tree
    '(branch (color        rtn 'color)
             (colors       rtn 'color)
             (size         rtn 'size)
             (sizes        rtn 'size)
             (weight       rtn 'weight)
             (weights      rtn 'weight)
             ;; additions to handle books database
             (author       rtn 'author)
             (authors      rtn 'authors)
             (type         rtn 'category)
             (category     rtn 'category)
             (topics       rtn 'category)
             (publisher    rtn 'publisher)
             (author       rtn 'author)
             (price        rtn 'price)
             (prices       rtn 'price)
             (cost         rtn 'price)
             (title        rtn 'title)
```

```
            (titles       rtn 'title)
            (isbn         rtn 'isbn))))

(define attribute
  ;; Consumes, for example, COLOR
  (make-tree
    '(branch (color        rtn 'color)
            (colors       rtn 'color)
            (size         rtn 'size)
            (sizes        rtn 'size)
            (weight       rtn 'weight)
            (weights      rtn 'weight)
            ;; ADDITIONS TO HANDLE BOOKS DATABASE
            (author       rtn 'author)
            (authors      rtn 'authors)
            (type         rtn 'category)
            (category     rtn 'category)
            (topics       rtn 'category)
            (publisher    rtn 'publisher)
            (author       rtn 'author)
            (price        rtn 'price)
            (prices       rtn 'price)
            (cost         rtn 'price)
            (title        rtn 'title)
            (titles       rtn 'title)
            (isbn         rtn 'isbn))))

(define attribute-value
  ;; Consumes, for example, LARGE
  (make-tree
    '(branch ((size-value --> v) rtn v)
            ((color-value --> v) rtn v)
            ((weight-value --> v) rtn v)
            ;;; ADDITIONS FOR BOOKS DATABASE
            ((material-value --> v) rtn v)
            ((category-value --> v) rtn v)
            ((author-value --> v) rtn v)
            ((publisher-value --> v) rtn v))))

;; ADDITIONS TO HANDLE BOOKS DATABASE

(define material-value
  (make-tree
    '(branch (paperback  rtn '(format eq? paper))
            (hardcover  rtn '(format eq? hardcvr)))))

(define author-value
  (make-tree
```

```
    '(branch (winston  rtn '(author eq? winston))
             (winstons rtn '(author eq? winston))
             (patricks rtn '(author eq? winston))
             (gould    rtn '(author eq? gould))
             (goulds   rtn '(author eq? gould))
             (gibson   rtn '(author eq? gibson))
             (gibsons  rtn '(author eq? gibson)))))

(define publisher-value
  (make-tree
   '(branch (ace            rtn '(publisher eq? ace))
            (addison-wesley rtn '(publisher eq? addison-wesley))
            (aw             rtn '(publisher eq? addison-wesley))
            (bantam         rtn '(publisher eq? bantam))
            (norton         rtn '(publisher eq? norton))
            (putnam         rtn '(publisher eq? putnam)))))

(define category-value
  (make-tree
    '(branch (science  rtn '(category eq? science))
             (sci-fi   rtn '(category eq? fiction))
             (fiction  rtn '(category eq? fiction))
             (computer rtn '(category eq? computers))
             (cs       rtn '(category eq? computers)))))
```

## Part B

To handle questions such as *What are the books on fiction*, one method is to modify the net that parses objects, `object-description` so that it can analyze phrases like *about fiction* and add them to the attribute list that is used by the database selection procedures. There are many other possible ways to construct such networks. The method that is used below splits up object descriptions into two subnetworks: one that parses objects as before, `ordinary-object-description`, and one that allows for phrases coming after the main object, starting with prepositions such as *on* or "synonyms" like *published by*.

```
;;;; OBJECTS

(define ordinary-object-description
  ;; Consumes, for example, A BIG BLACK HEAVY BLOCK
  (make-tree
    '(branch ((determiner) (ordinary-object-description --> o) rtn o)
             ((attribute-value-list --> a-v-l)
              (ordinary-object-description --> o)
              rtn `(db-select ,o '(with ,@a-v-l)))
```

```
            ((object --> o) rtn o))))

(define object-description
  (make-tree
   '(branch ((ordinary-object-description --> o)
             (after-object-list --> a-o-l)
                     rtn `(db-select ,o '(with ,@a-o-l)))
            ((ordinary-object-description --> o) rtn o)
            )))

(define after-object-list
  (make-tree
   '(branch ((prep) (attribute-value --> a-v) rtn a-v))))

(define prep
  ;; Consumes, for example, ON or PUBLISHED BY
  (make-tree
   '(branch (on) (by) (about) (published by))))
```

## Problem 3

To use `main` to implement an Eliza-like system that can generate both questions and answers, we can dispatch on various keywords that are found in the input, responding with canned answers. An example is shown below. A more sophisticated system might make use of the parse to fill a thematic role frame ("who did what to whom") in order to construct a more "relevant" response, instead of using just keywords. However, the chief point is that no matter how the user's input is "understood," in a *generating* sentences, one must make word and phrase choices, rather than simply be guided by the user's input. In generating (as opposed to parsing) sentences, some mechanism must be introduced to vary the "style" (word choice and sentence structure), if only by the random selection of responses. Further, in a dialog one ought to somehow globally keep track of what has been "said".

One modification to `main` might go as follows. Any similarity to existing human conversation and handed-in student code is entirely not coincidental.

```
;;;
;;; additions

(define main
  (make-tree
    '(branch
      ;; COUNT THE BLOCKS
      (count (object-description --> o)
        if-end-rtn (db-call `(db-report-count (db-count-items ,o))))
```

```
      ;; HOW MANY BLOCKS ARE THERE
      (how many (object-description --> o) are there
        if-end-rtn (db-call `(db-report-count (db-count-items ,o))))
      ;; WHAT COLORS ARE THE BLOCKS
      (what (attribute-list --> a-l) are (object-description --> o)
        if-end-rtn (db-call `(db-show (db-project ,o '(over ,@a-l)))))
      ;; WHAT ARE THE BLOCKS COLORS
      (what are (object-description --> o) (attribute-list --> a-l)
        if-end-rtn (db-call `(db-show (db-project ,o '(over ,@a-l)))))
      ((identify)
         branch
         ;; IDENTIFY THE BRICKS
         ((object-description --> o)
          if-end-rtn (db-call `(db-show ,o))
          (format #t "~2%Why are you so interested in ~A" (print-extract o)))
         ;; respond to questions with answers and new, randomized questions
         ((eliza-type-answers))
         ;; IDENTIFY THE COLORS OF THE BRICKS
         (the (attribute-list --> a-l) of (object-description --> o)
          if-end-rtn (db-call `(db-show (db-project ,o '(over ,@a-l))))))
      ;; RANK THE BRICKS BY COLOR
      ((rank) (object-description --> o) by (attribute-list --> a-l)
       if-end-rtn (db-call `(db-show (db-sort ,o '(by ,@a-l)))))
            (stop if-end-rtn 'quit)
      (rtn (format #t "~%Sorry, I do not understand")
           #f))))


(define eliza-type-answers
 (make-tree
  '(branch
    ((conjunction) (answers --> a) rtn a)
    (I (positive-statement --> p) (object-description --> o)
       if-end-rtn
       (talk p o)
       (newline)
       (write-line '(Here they are))
       (db-call `(db-show ,o))
       (newline)
       (question))
    (I (positive-statement --> p) (attribute-value-list --> a-v-l)
       if-end-rtn
       (newline)
       (talk p `(a (a b '()) (with (a. ,@a-v-l))))
       (write-line '(Here you go))
       (db-call `(db-show (db-select books '(with ,@a-v-l))))
       (newline)
       (question))
```

```
    ((price) (object-description --> o)
             rtn
             (newline)
             (write-line (append '(Here are the prices of) (print-extract o)))
             (db-call `(db-show (db-project ,o '(over title author price))))
             (newline)
             (question)))))

;; randomization for dialog
(define (talk p o)
  (let ((choice (random 4)))
    (newline)
    (cond ((= choice 0)
           (write-line (append '(Is that so? I) (list p) (print-extract o) '(also))))
          ((= choice 1)
           (write-line (append '(Good choice! I) (list p) (print-extract o) '(too))))
          ((= choice 2)
           (write-line (append '(What a coincidence. You and I both) (list p)
                               (print-extract o) '())))
          ((= choice 3)
           (write-line (append '(Mois aussi. I ) (list p) (print-extract o)
                              '(without reservation))))
          (else
           (write-line (append '(Your choice. But I don't) (list p) (print-extract o)
                              '(very much)))))))

;; random questions after answers
(define (question)
  (let ((choice (random 4)))
    (cond ((= choice 0)
           (write-line '(Are there any other books that your like?)))
           (= choice 1)
           (write-line '(How about some other books?)))
           (= choice 2)
           (write-line '(You must have some other books you'd like to buy)))
           (= choice 3)
           (write-line '(What else?))))))

;; get values from db in a more printable form
;;
(define (print-extract o)
  (let ((books (second o)))
    (if (pair? books)
        (let ((book (if (null? (second  (third books)))
                        '(books)
                        (get-values (rest (second (third books)))
                                    '() ))))
          (append (reverse (get-values (rest (second (third o))) '())) book))
```

```
        (if (pair? (second (third o)))
            (list (fourth (second (third o))))
            'books))))

(define (get-values o v)
  (if (null? o)
      v
      (get-values (cdddr o) (cons (third o) v))))

;; extra tree-transition constructs

(define positive-statement
  (make-tree
   '(branch (like rtn 'like)
            (love rtn 'love)
            (want rtn 'really like))))

(define conjunction
  (make-tree
   '(branch (because)
            (since)
            (so)
            (really)
            (and))))

(define verb
  (make-tree
   '(branch (is)
            (are)
            (do)
            (does))))

(define price
  (make-tree
   '(branch (how much (verb))
            (how costly)
            (how much)
            (how expensive (verb))
            (what is the price of))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
> (show me the books)

[table of all books]

Why are you interested in books?
```

```
> (because I like winston)

(Good choice!  I like Winston books too)
(Let me show you them)

[table of Winston's books]

(Are there any others books that you like?)

>(I love Gould)

(Is that  so? I love Gould books also.)

etc.
```