

Tzewen Wang
ECE578
Winter 2005

Homework 3c

Source code with description

```
/**
 * Tzewen Wang
 * ECE578
 * Winter 2005
 *
 * Homework 3c
 *
 * This program implements variant c which is a simple language giving
 * the robot imperative commands. The grammar is specified as the following:
 *
 * ACT -> CMD ("AND" | "WHILE" CMD)*
 *
 * CMD -> "TURN" (DIR | "AROUND") |
 *       "WALK" NUM "STEPS" |
 *       "TELL" STR |
 *       "SING" SONG_LIST |
 *       "SMILE" |
 *       "SMILING" |
 *       "CRY" |
 *       "CRYING" |
 *       "DANCE" |
 *       "WAVE" ("YOUR")- DIR PARTS NUM "TIMES" |
 *       "RAISE" ("YOUR")- DIR PARTS |
 *       "LIFT" ("YOUR")- DIR PARTS
 *
 * DIR -> "LEFT" | "RIGHT"
 *
 * NUM -> "ONE" | "TWO" | "THREE" | "FOUR" | "FIVE"
 *
 * SONG_LIST -> "\"A SONG\""
 *
 * PARTS -> "HAND" | "FOOT"
 *
 * The grammar is expressed by regular expression. The minus sign (-) simply
 * means the term can occur at most once.
 *
 * This program parses a command using the language described above. Each term
 * in the grammar has its own function to handle successive input
 * respectively. The functions are invoked exactly in the same order as the
 * grammar. At top level, the command will be parsed and chunked into
 * partial sentences by identifying conjunction keywords such as AND and WHILE.
 * Each partial sentence will be then parsed by a function that follows
 * term CMD recursively. The functions are designed in such a way that NIL
 * is returned if no errors are found. Otherwise, the function will return
 * at whichever word that is not in the dictionary and non-gramatical and
 * the rest of the unparsed sentence. This is particularly useful for
 * debugging as well as produce an error message informing the commander.
 *
 * There is no actual command data structure in this project. It just a
 * simple parser. The semantic actions can be inserted and developed in
 * the future.
 */
```

```

;/**
; * Global variables.
;*/
(setf DICT `(AND
            WHILE
            AROUND
            WALK
            STEPS
            TELL
            TURN
            SING
            SMILE
            SMILING
            CRY
            CRYING
            DANCE
            WAVE
            RAISE
            LIFT
            YOUR
            TIMES
            LEFT
            RIGHT
            ONE
            TWO
            THREE
            FOUR
            FIVE
            LEFT
            RIGHT
            HAND
            FOOT))

(setf SONG_LIST `("A SONG"))

```

```

;/**
; * accept (SENTENCE)
; *
; * Parses a sentence.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept (SENTENCE)
  (accept-cmd (accept-act SENTENCE)))

```

```

;/**
; * accept-act (SENTENCE)
; *
; * Parses term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.

```

```

;*/
(defun accept-act (SENTENCE)
  (cond ((null SENTENCE) NIL)
        ((null (find-word (first SENTENCE) (append DICT SONG_LIST))) SENTENCE)
        ((find-word (first SENTENCE) `(AND WHILE))
         (accept-cmd (accept-act (rest SENTENCE))))
        (T (cons (first SENTENCE) (accept-act (rest SENTENCE))))))

```

```

;/**
; * accept-cmd (SENTENCE)
; *
; * Parses term CMD.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/

```

```

(defun accept-cmd (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `TURN) (accept-turn (rest SENTENCE)))
        ((equal (first SENTENCE) `WALK) (accept-walk (rest SENTENCE)))
        ((equal (first SENTENCE) `TELL) (accept-tell (rest SENTENCE)))
        ((equal (first SENTENCE) `SING) (accept-sing (rest SENTENCE)))
        ((equal (first SENTENCE) `RAISE) (accept-raise (rest SENTENCE)))
        ((equal (first SENTENCE) `LIFT) (accept-lift (rest SENTENCE)))
        ((equal (first SENTENCE) `WAVE) (accept-wave (rest SENTENCE)))
        ((equal (first SENTENCE) `SMILE) (rest SENTENCE))
        ((equal (first SENTENCE) `SMILING) (rest SENTENCE))
        ((equal (first SENTENCE) `CRY) (rest SENTENCE))
        ((equal (first SENTENCE) `CRYING) (rest SENTENCE))
        ((equal (first SENTENCE) `DANCE) (rest SENTENCE))
        (T SENTENCE)))

```

```

;/**
; * accept-turn (SENTENCE)
; *
; * Parses action TURN in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/

```

```

(defun accept-turn (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `AROUND)
         (rest SENTENCE))
        (T (accept-dir SENTENCE))))

```

```

;/**
; * accept-walk (SENTENCE)
; *
; * Parses action WALK in term ACT.
; *

```

```

; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-walk (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first (accept-num SENTENCE)) `STEPS)
         (rest (rest (accept-num SENTENCE))))
        (T SENTENCE)))

;/**
; * accept-tell (SENTENCE)
; *
; * Parses action TELL in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-tell (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((stringp (first SENTENCE))
         (rest SENTENCE))
        (T SENTENCE)))

;/**
; * accept-sing (SENTENCE)
; *
; * Parses action SING in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-sing (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((find-word (first SENTENCE) SONG_LIST) (rest SENTENCE))
        (T SENTENCE)))

;/**
; * accept-raise (SENTENCE)
; *
; * Parses action RAISE in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-raise (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `YOUR)
         (rest SENTENCE))
        (T SENTENCE)))

```

```

        (accept-parts (accept-dir (rest SENTENCE))))
      (T (accept-parts (accept-dir SENTENCE))))))

;/**
; * accept-lift (SENTENCE)
; *
; * Parses action LIFT in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-lift (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `YOUR)
         (accept-parts (accept-dir (rest SENTENCE))))
        (T (accept-parts (accept-dir SENTENCE)))))

;/**
; * accept-wave (SENTENCE)
; *
; * Parses action WAVE in term ACT.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-wave (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `YOUR)
         (let (PARSED_SENTENCE (accept-num (accept-parts (accept-dir (rest
SENTENCE))))))
          (if (equal (first PARSED_SENTENCE) `TIMES)
              (rest (rest PARSED_SENTENCE))
              PARSED_SENTENCE)))
        (T (let (PARSED_SENTENCE (accept-num (accept-parts (accept-dir
SENTENCE))))
            (if (equal (first PARSED_SENTENCE) `TIMES)
                (rest (rest PARSED_SENTENCE))
                PARSED_SENTENCE))))))

;/**
; * accept-num (SENTENCE)
; *
; * Parses term NUM.
; *
; * Parameters:
; * SENTENCE - the command to be parsed.
; *
; * Returns NIL if success.
;*/
(defun accept-num (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `ONE) (rest SENTENCE))

```

```

      ((equal (first SENTENCE) `TWO) (rest SENTENCE))
      ((equal (first SENTENCE) `THREE) (rest SENTENCE))
      ((equal (first SENTENCE) `FOUR) (rest SENTENCE))
      ((equal (first SENTENCE) `FIVE) (rest SENTENCE))
      (T SENTENCE)))

;/**
; * accept-dir (SENTENCE)
; *
; *   Parses term DIR.
; *
; * Parameters:
; *   SENTENCE - the command to be parsed.
; *
; * Returns everything after DIR.
;*/
(defun accept-dir (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `LEFT) (rest SENTENCE))
        ((equal (first SENTENCE) `RIGHT) (rest SENTENCE))
        (T SENTENCE)))

;/**
; * accept-parts (SENTENCE)
; *
; *   Parses term PARTS.
; *
; * Parameters:
; *   SENTENCE - the command to be parsed.
; *
; * Returns everything after PARTS.
;*/
(defun accept-parts (SENTENCE)
  (cond ((null SENTENCE) `ERROR)
        ((equal (first SENTENCE) `HAND) (rest SENTENCE))
        ((equal (first SENTENCE) `FOOT) (rest SENTENCE))
        (T SENTENCE)))

;/**
; * find-word (WORD KEYWORDS)
; *
; *   Finds a word occurring in a given list of keywords.
; *
; * Parameters:
; *   WORD - the word to be matched with a list of words.
; *   KEYWORDS - a list of words.
; *
; * Returns T if found.
;*/
(defun find-word (WORD KEYWORDS)
  (cond ((null KEYWORDS) NIL)
        ((equal WORD (first KEYWORDS)) T)
        (T (find-word WORD (rest KEYWORDS)))))

```

```

;/**
; * replace-word (SENTENCE KEY NEWWORD)
; *
; * Replaces a word in the sentence with a new word.
; *
; * Parameters:
; * SENTENCE - a list of words to be processed.
; * KEY - the old word to be substituted.
; * NEWWORD - the new word.
; *
; * Returns substituted list of words.
;*/
(defun replace-word (SENTENCE KEY NEWWORD)
  (cond ((null SENTENCE) SENTENCE)
        ((equal (first SENTENCE) KEY)
         (cons NEWWORD (replace-word (rest SENTENCE) KEY NEWWORD)))
        (T (cons (first SENTENCE) (replace-word (rest SENTENCE) KEY NEWWORD)))))

;/**
; * The following code are borrowed from Peter Norvig, the author of
; * "Paradigms of Artificial Intelligence Programming: Case studies in LIST"
; * The code deals with user interface.
;*/
(defun read-line-no-punct ()
  "Read an input line, ignoring punctuation."
  (read-from-string
   (concatenate 'string "(" (substitute-if #\space #'punctuation-p
                                             (read-line))
                ")")))
(defun punctuation-p (char) (find char ".,:;!#-()\`\"))
(defun print-with-spaces (list)
  (mapc #'(lambda (x) (prinl x) (princ " ")) list))
(defun print-with-spaces (list)
  (format t "~{~a ~}" list))

;/**
; * Implements user interface and driver.
;*/
(defun main ()
  (print-with-spaces `(WHAT CAN I DO FOR YOU ?))
  (loop
   (print `ELIZA-B>)
   (let ((COMMAND (read-line-no-punct)))
     (cond ((null (accept COMMAND))
            (print-with-spaces
             (replace-word (append (append `(YOU WANT ME TO) COMMAND) `(?)
                                     `YOUR `MY))
                           (print `ELIZA-B>)
                           (if (equal (read-line-no-punct) `(YES))
                               (print-with-spaces `(NOT A PROBLEM. WHAT IS NEXT ?))
                               (print-with-spaces `(ALRIGHT. LET'S NOT DO THAT.))))
            (T (print-with-spaces `(I DO NOT UNDERSTAND WHAT YOUR ARE
SAYING)))))))

```

Program output

[255]> (main)
WHAT CAN I DO FOR YOU ?
ELIZA-B> turn down
I DO NOT UNDERSTAND WHAT YOUR ARE SAYING
ELIZA-B> turn right
YOU WANT ME TO TURN RIGHT ?
ELIZA-B> yes
NOT A PROBLEM. WHAT IS NEXT ?
ELIZA-B> dance and sing "A SONG" while smiling
YOU WANT ME TO DANCE AND SING A SONG WHILE SMILING ?
ELIZA-B> yes
NOT A PROBLEM. WHAT IS NEXT ?
ELIZA-B> wave left hand three times and walk five steps
YOU WANT ME TO WAVE LEFT HAND THREE TIMES AND WALK FIVE STEPS ?
ELIZA-B> no
ALRIGHT. LET 'S NOT DO THAT.
ELIZA-B> raise your left foot
YOU WANT ME TO RAISE MY LEFT FOOT ?
ELIZA-B> yes
NOT A PROBLEM. WHAT IS NEXT ?
ELIZA-B> cry
YOU WANT ME TO CRY ?
ELIZA-B> yes
NOT A PROBLEM. WHAT IS NEXT ?
ELIZA-B> lift your left hand
YOU WANT ME TO LIFT MY LEFT HAND ?
ELIZA-B> yes
NOT A PROBLEM. WHAT IS NEXT ?
ELIZA-B>