CS262 Artificial Intelligence Concepts

Lisp Coding Guidelines

1 Introduction

This document describes the coding guidelines which should be followed for submission of Lisp programs in CS26210 and CS36210.

2 Lisp dialect, interpreter and file names

All Lisp programs should be written in Common Lisp, and should be read by the GNU Common Lisp interpreter (Version 1.1) on the Departmental Sun workstations. If you use any other Lisp dialect or interpreter for the development of the program, make sure you test the program so that it works properly under the environment described above.

The extension `.lisp` should be used in the file names.

3 Coding style

3.1 Headings

This should be followed by filename, purpose, author, date, and information concerning files and packages the program depends upon. This should all be there in the form of comments, i.e., each line beginning with a semi-colon (;), so that the file can be loaded without raising any error.

3.2 Function format

Function definitions should follow the format below:

```
(defun function_name (arg1 arg2 ...)
    function_body)
```

That is, the first line of the definition should contain `defun`, the function name and the list of arguments, and the body of the function should begin on the next line.

3.3 Brackets

The position of closing brackets should be clear and consistent. One possible (and recommended) style is to place all the closing brackets at the end, e.g.,

```
(defun myfirst (list)
   (cond ((not (listp list)) (error "MYFIRST: the argument must be a list"))
         (t (first list))))
```

Another possibility is to align the corresponding closing brackets with the opening brackets for larger expressions, e.g.,

```
(defun myfirst (list)
   (cond ((not (listp list)) (error "MYFIRST: the argument must be a list"))
         (t (first list))
   )
)
```

The latter makes clear the pairing between corresponding brackets, but might end up using many lines just containing closing brackets. The former accumulates closing brackets at the end of the definition, but with a proper indentation scheme, this won't be a problem. Choose whichever style you like, but make sure it's consistent and, more importantly, readable.

3.4 Commenting and Indentation

Your program should be commented *heavily*. Especially, for each function defined, you should provide a function header which contains 1) function name, 2) list of arguments (and their datatypes if crucial), 3) results expected to be returned, 4) side effects (if any), and 5) relevant remarks.

Make sure the indentation is consistent and improves clarity of the program. Each indentation should be between 3 and 6 spaces.

4 Software Engineering

Remember that Lisp (unlike Ada but like C) has no design features that help you to write good code. This means you must take special care in the design and testing of your functions. You *can* write impenetrable code that works but which no one can understand (very like C), or you can use modular and hierarchical design, structured testing, etc and produce clean effective code. Needless to say, the latter will get far more marks! There are a few points that will help:

- Use local rather than global variables
- Emphasise all global variables by using * around names
- Remember, there is no type checking - any misspelling will be accepted!
- Keep functions small (if a function gets large, redesign it)
- Use comments to remind you of the design
- Test functions on their own before combining
- Add code to print key variables during testing and debugging
- Be systematic!

5 Checklist

- Is there adequate information in the header?
- Is the commenting clear and appropriate?
- Are the functions small and concise?
- Are the variables properly initialised?
- Are the variables reasonably mnemonic?
- Are the global variables used appropriately?
- Are the algorithms easy to follow?

6 Example

```
;;;;   File:    rules.lisp
;;;;   Author:  Keiichi Nakata
;;;;   Date:    6 March 1996
;;;;   Purpose: CS26210 Assignment 4
;;;;
;;;;   Top level functions: parse, generate
;;;;     Usage: (parse <sentence>)
;;;;            (generate <grammar>)
;;;;
;;;;   Other files/packages required:  auxfuns.lisp, lexicon.lisp

;;;;   Loading necessary files

(load "auxfns.lisp")
(load "lexicon.lisp")

;;;;   Initialising global variables

(setq *wordlist* nil)
(setq *sentence-parsed* nil)
(setq *english* t)

;;;;   Function:      parse
;;;;   Arguments:     (1) a list representing a sentence
;;;;   Results:       Returns the parse tree of the sentence provided
;;;;   Side effects:  - Prints out the sentence type (see SENT-TYPE).
;;;;                  - Sets global variable *sentence-parsed* to t.
;;;;   Remark:        This function implements parsing using the
;;;;                  standard depth-first search strategy.
(defun parse (sent)
   ;;; First check the sentence type
   (cond ((equal (stype sent) 'question)
          (princ "This is a question.") (terpri))
         (t (princ "This is a normal sentence.") (terpri)))
   ;;; The parsing function.  English sentence by default (*english*)
   (parsing 'depth-first (struct sent) *english*)
   ;;; Set the parsed flag to T
   (setq *sentence-parsed* t))

;;;;   Function:      memberp
;;;;   Arguments:     (1) any
;;;;                  (2) a list
;;;;   Results:       Returns T if (1) is a member of (2); NIL otherwise.
;;;;   Side effects:  None.
;;;;   Remark:        Uses EQUAL to check the equality.
(defun memberp (item list)
    (cond ((null list) nil)               ;;; NIL if the empty list is given
          ((equal item (first list)) t)   ;;; Uses EQUAL for comparison
          (t (memberp item (rest list)))))
```