ECS 140A Programming Languages

Spring 2024

Homework 2 due Wednesday May 8th at 5pm

About This Assignment

- This assignment asks you to complete programming tasks using the GNU Common Lisp programming language.
- **You are only allowed to use the subset of Lisp that we have discussed in class.** For example, using loop constructs or other built-in functions is not allowed. **You will get no credit in this assignment if any of the problem solutions use constructs or built-in functions not discussed in class.** Please use Piazza for any clarifications regarding this issue.
- Note that the problems can be solved without using `setq`, and we would advise against its use. However, you are allowed to use `setq`, and no points will be deducted if you use it in your solution.
- This assignment has to be worked on individually.
- You are not allowed to search the internet for code to solve the homework. All submitted code must be your own.
- To complete the assignment (i) download `hw2-handout.zip` from Canvas, (ii) modify the `.lisp` files in the `hw2-handout` directory as per the instructions in this document, and (iii) zip the `hw2-handout` directory into `hw2-handout.zip` and upload this `zip` file to Canvas by the due date.

  Do not change the file names, create new files, or change the directory structure in `hw2-handout`.
- We will be using the GNU CLISP implementation of Common Lisp, version 2.49, which can be installed from https://clisp.sourceforge.io/.

  Use the command `clisp --version` to verify that you have the correct version installed:

  ```
  $ clisp --version
  GNU CLISP 2.49<other output>
  ```

- CLISP 2.49 is also installed on *all* CSIF machines. For instance,

  ```
  $ ssh <kerberos-id>@pc2.cs.ucdavis.edu
  <ssh output>
  $ clisp --version
  GNU CLISP 2.49<other output>
  ```

- Information about using CSIF computers, such as how to remotely login to CSIF computers from home and how to copy files to/from the CSIF computers using your personal computer, can be found at http://csifdocs.cs.ucdavis.edu/about-us/csif-general-faq.
- Begin working on the homework early.
- Apart from the description in this document, look at the unit tests provided to understand the requirements for the code you have to write.

  We are using the `lisp-unit` test framework.[1]

  You do NOT *need* to write new unit tests: code coverage is not going to be a factor in the grade for this assignment. However, you are encouraged to add new tests to understand and test your own code.
- Post questions on Piazza if you require any further clarifications. Use private posts if your question contains part of the solution to the homework.
- This content is protected and may not be shared, uploaded, or distributed.

General Tips

- When developing your program, you might find it easier to first test your functions interactively before using the test program. You might find trace, step, print functions useful in debugging your functions.
- The command `clisp myFile.lisp` runs the lisp interpreter on the file `myFile.lisp`.
- You can start clisp interactively using:

  ```
  $ clisp
  ```

- To load function definitions from/run `myFile.lisp` in the current directory:

  ```
  [1]> (load "myFile.lisp")
  ```

- To exit error mode, choose the command for ABORT (in this case, it's `:R3`):

  ```
  [1]> asjfkasf
  <error output>
  ABORT :R3 Abort main loop
  <error output>
  [2]> :R3
  [3]>
  ```

- You can exit the interactive clisp interpreter using:

  ```
  [1]> (bye)
  ```

[1] https://github.com/OdonataResearchLLC/lisp-unit

1 mmm (10 points)

- Complete the definition of the function `min-mean-max` in `hw2-handout/mmm/mmm.lisp`, which takes a list of numbers (with at least one element) and returns a list of length 3 that consists of the smallest number, the mean (reduced to the simplest fraction) of all numbers and the largest number.

```
> (min-mean-max '(2 5 11 15 7 1 8))
(1 7 15)

> (min-mean-max '(6 6 5 -4 3 2 1 1))
(-4 5/2 6)
```

- Use the following commands to run the unit tests provided in `hw2-handout/mmm/mmm_test.lisp`:

```
$ cd hw2-handout/mmm/
$ clisp mmm_test.lisp
```

2 qsort (10 points)

- Complete the definition of the function `pivot` in `hw2-handout/qsort/qsort.lisp`, which takes a list `xs` and a number `n` and splits it into two lists, one containing all the numbers in `xs` less than `n` and the other containing all numbers in `xs` greater than or equal to `n`. The function should preserve the relative order of elements inside the list.

```
> (pivot 3 '(3 2 5 1 4))
((2 1) (3 5 4))

> (pivot 3 nil)
(NIL NIL)
```

- Complete the definition of the function `quicksort` in `hw2-handout/qsort/qsort.lisp`, which sorts a list.

  Review of the quicksort algorithm: First pick an element and call it the pivot. The head of the list is an easy option for pivot. Partition the rest of the list into two sublists, one with all the elements less than the pivot and the other with all the elements not less than the pivot. Recursively sort the sublists. Combine the two sublists and the pivot into a final sorted list.

```
> (quicksort '(2 9 5 3 8))
(2 3 5 8 9)
```

- Use the following commands to run the unit tests provided in `hw2-handout/qsort/qsort_test.lisp`:

```
$ cd hw2-handout/qsort/
$ clisp qsort_test.lisp
```

3 match (15 points)

- An *assertion* represents a fact in the form of a list. For instance, the following are three different assertions:

  ```
  (this is an assertion)
  (color apple red)
  (supports table block1)
  ```

- The set of assertions can be maintained in a database by representing them in a list.

  For instance, the following list represents an assertion database containing the above assertions:
  `((this is an assertion) (color apple red) (supports table block1))`

- *Patterns* are like assertions, except that they may contain certain special atoms `?` and `!`, which are not allowed in assertions. Two examples of patterns are:

  ```
  (this ! assertion)
  (color ? red)
  ```

- Complete the definition of the function `match` in `hw2-handout/match/match.lisp`, which compares a pattern and an assertion.

  When a pattern containing no special atoms is compared to an assertion, the two match only if they are exactly the same, with each corresponding position occupied by the same atom.

  ```
  > (match '(color apple red) '(color apple red))
  T

  > (match '(color apple red) '(color apple green))
  NIL
  ```

  The special atom `?` matches any single atom.

  ```
  > (match '(color apple ?) '(color apple red))
  T
  > (match '(color ? red) '(color apple red))
  T
  > (match '(color ? red) '(color apple green))
  NIL
  ```

  In the last example, `(color ? red)` and `(color apple green)` do not match because red and green do not match.

The special atom `!` expands the capability of `match` by matching any one or more atoms.

```
> (match '(! table !) '(this table supports a block))
T
```

Here, the first `!` symbol matches this, table matches table, and the second `!` symbol matches supports a block.

```
> (match '(this table !) '(this table supports a block))
T
> (match '(! brown) '(green red brown yellow))
NIL
```

In the last example, the special symbol `!` matches `green red`. However, the match fails because `yellow` occurs in the assertion after `brown`, whereas it does not occur in the assertion. However, the following example succeeds:

```
> (match '(! brown) '(green red brown brown))
T
```

In this example, `!` matches the list `(green red brown)`, whereas `brown` matches the last element.

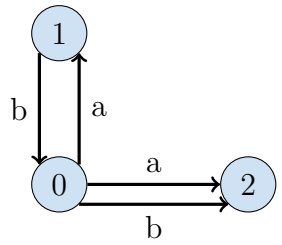
- Use the following commands to run the unit tests provided in `hw2-handout/match/match_test.lisp`:

```
$ cd hw2-handout/match/
$ clisp match_test.lisp
```

4 nfa (15 points)

- A non-deterministic finite automaton (NFA) is defined by a set of states, symbols in an alphabet, and a transition function. A state is represented by an integer. A symbol is represented by a rune, i.e., a character. Given a state and a symbol, a transition function returns the set of states that the NFA can transition to after reading the given symbol. This set of next states could be empty.

  A graphical representation of an NFA is shown below:



- In this example, $\{0, 1, 2\}$ are the set of states, $\{a, b\}$ are the set of symbols, and the transition function is represented by labelled arrows between states.
  - If the NFA is in state 0 and it reads the symbol a, then it can transition to either state 1 or to state 2.
  - If the NFA is in state 0 and it reads the symbol b, then it can only transition to state 2.
  - If the NFA is in state 1 and it reads the symbol b, then it can only transition to state 0.
  - If the NFA is in state 1 and it reads the symbol a, it cannot make any transitions.
  - If the NFA is in state 2 and it reads the symbol a or b, it cannot make any transitions.
- A given final state is said to be *reachable* from a given start state via a given input sequence of symbols if there exists a sequence of transitions such that if the NFA starts at the start state it would reach the final state after reading the entire sequence of input symbols.
- In the example NFA above:
  - The state 1 is reachable from the state 0 via the input sequence *abababa*.
  - The state 1 is *not* reachable from the state 0 via the input sequence *ababab*.
  - The state 2 is reachable from state 0 via the input sequence *abababa*.
- Complete the definition of the function `reachable` in `hw2-handout/nfa/nfa.lisp`, which returns `true` if a final state is reachable from the start state after reading an input sequence of symbols, and `nil`, otherwise.

  The transition function for the NFA described above is represented by the `expTransitions` function in `hw2-handout/nfa/nfa_test.lisp`.

```
> (reachable 'expTransitions 0 0 '(A B))
T

> (reachable 'expTransitions 0 0 '(A A))
nil
```

- Use the following commands to run the unit tests provided in `hw2-handout/nfa/nfa_test.lisp`:

```
$ cd hw2-handout/nfa/
$ clisp nfa_test.lisp
```

- You may need to use `funcall` or `apply` to call the transition function to get the next states.

5 matrix (30 points)

- Suppose we represent a matrix in LISP as a list of lists. For example, `((a b) (c d))` would represent a 2*2 matrix whose first row contains the elements `a` and `b`, and whose second row contains the elements `c` and `d`. You may assume that the matrices are well-formed, compatible, and not empty.

- Complete the definition of the function `matrix-add` in `hw2-handout/matrix/matrix.lisp`, which takes two matrices as input and outputs the sum of the two matrices.

```
> (matrix-add '((1 2) (2 1)) '((1 2) (3 4)))
((2 4) (5 5))
```

- Complete the definition of the function `matrix-transpose` in `hw2-handout/matrix/matrix.lisp`, which takes a matrix as input, and outputs its transpose. You may assume that the matrix is well-formed, and not empty.

```
> (matrix-transpose '((1 2 3) (4 5 6)))
((1 4) (2 5) (3 6))
```

- Complete the definition of the function `matrix-multiply` in `hw2-handout/matrix/matrix.lisp`, which takes two matrices as input and multiplies them and outputs the resultant. You may assume that the matrices are well-formed, compatible, and not empty.

```
> (matrix-multiply ((1 2) (2 1)) '((3 1) (1 3)))
((5 7) (7 5))
```

- Use the following commands to run the unit tests provided in `hw2-handout/matrix/matrix_test.lisp`:

```
$ cd hw2-handout/matrix/
$ clisp matrix_test.lisp
```