ECS 140A Programming Languages
Homework 3

About This Assignment

- This assignment asks you to complete programming tasks using the Go programming language and the GNU Common Lisp programming language.
- **You are only allowed to use the subset of Go and Lisp that we have discussed in class. No credit will be given in this assignment if any of the problem solutions use material not discussed in class.** Please use Piazza for any clarifications regarding this issue.
- To complete the assignment (i) download `hw3-handout.zip` from Canvas, (ii) modify the `members.txt`, `.go` and `.lisp` files in the `hw3-handout` directory as per the instructions in this document, and (iii) zip the `hw3-handout` directory into `hw3-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 of `hw3-handout`.
- This assignment can be worked on in a group. Even if working in a group, each student has to submit a copy of the work to Canvas to get credit. See the Syllabus for more details.
- List all the names and email addresses of all members of the group in the `members.txt` file in the `hw3-handout` directory, one per line in the format `name <email>`.

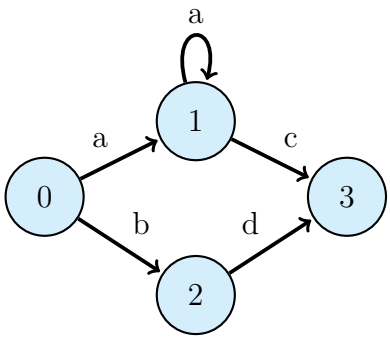
  If you are working individually, then only add your name and email to `members.txt`.
- Refer to **Homework 0** for instructions on installing the correct versions of the programming language as well as using CSIF computers.
- Begin working on the homework early.
- Apart from the description in this document, look at the unit tests provided to understand the requirements for the code you have to write.
- Post questions on Piazza if you require any further clarifications. Use private posts if your question contains part of the solution to the homework.
- Keep your homework solution after you submit it. You may need to use it for later assignments.

1 lgraph (20 points)

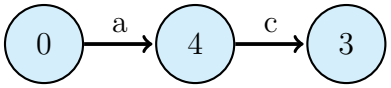
For this part of the assignment, you only need to modify `hw3-handout/lgraph/lgraph.lisp`.

- A *labeled directed graph* G is a directed graph where each edge is labeled with a symbol.

  Figure 1 shows two examples of labeled directed graphs G_1 and G_2.



(a) Labeled directed graph G_1



(b) Labeled directed graph G_2

Figure 1

- $[\ell_1, \ell_2, \ldots, \ell_k]$ is said to be a *sequence* from nodes u to v in G if there exists a path from u to v in G whose edges are labeled with $\ell_1, \ell_2, \ldots, \ell_k$ in order. The number of labels k in the sequence is defined as the *length of the sequence*.

  The sequences $[a, c]$ and $[b, d]$ are both sequences of length 2 from node 0 to node 3 in graph G_1. The sequence $[a, a, c]$ is a sequence of length 3 from node 0 to node 3 in graph G_1.

  The sequence $[a, c]$ is a sequence of length 2 from node 0 to node 3 in graph G_2.

- In `hw3-handout/lgraph/lgraph.lisp`, implement the `find-sequence` function.

  `find-sequence('g1 'g2 u v k)` returns `((S).t)` if there is a sequence S of length `k` from node `u` to node `v` in graph `g1` and S is *not* a sequence from `u` to `v` in graph `g2`; else it returns `nil`.

- Consider the graphs G_1 and G_2 shown in Figure 1:

  `find-sequence('G1 'G2 0 3 2)` should return `((b d).t)`

  `find-sequence('G1 'G2 0 3 3)` should return `((a a c).t)`

  `find-sequence('G1 'G2 3 0 2)` should return `nil`

  `find-sequence('G2 'G1 0 3 2)` should return `nil`

  `find-sequence('G2 'G1 4 4 0)` should return `(nil.t)`

- You can additional helper functions in `hw3-handout/lgraph/lgraph.lisp`, if needed.

- Use the following commands to run the unit tests provided in `hw3-handout/lgraph/lgraph_test.lisp`:

```
$ cd hw3-handout/lgraph/
$ clisp lgraph_test.lisp
```

2 match (10 points)

For this part of the assignment, you only need to modify `hw3-handout/match/match.lisp`.

- 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 `hw3-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 `hw3-handout/match/match_test.lisp`:

```
$ cd hw3-handout/match/
$ clisp match_test.lisp
```

3 MiniLisp (40 points)

In this assignment, you will implement a parser and interpreter for a small subset of lisp (MiniLisp) in the Go programming language.

For this part of the assignment, you only need to modify `hw3-handout/sexpr/parser.go` and `hw3-handout/sexpr/parser_test.go`, which implements the parser for MiniLisp, as well as `hw3-handout/sexpr/eval.go` and `hw3-handout/sexpr/eval_test.go`, which implements the interpreter for MiniLisp.

- You need to define a `struct` type that implements the `Parser` interface defined in `hw3-handout/sexpr/parser.go`. Do not modify this interface.

  Specifically, this type needs to implement the method `Parse(string) (*SExpr, error)` that takes a `string` and parses it to a `*SExpr` if the input string is in the language of grammar G defined below, else it returns an error.

  The grammar G, whose start symbol is `<sexpr>`, is:

```
<sexpr>       ::= <atom> | <pars> | QUOTE <sexpr>
<atom>        ::= NUMBER | SYMBOL
<pars>        ::= LPAR <dotted_list> RPAR | LPAR <proper_list> RPAR
<dotted_list> ::= <proper_list> <sexpr> DOT <sexpr>
<proper_list> ::= <sexpr> <proper_list> | \epsilon
```

- The type `SExpr` is defined in `hw3-handout/sexpr/sexpr.go`. You will also find helpful comments and helper functions in this file.
- See the tests in `hw3-handout/sexpr/parser_test.go` to understand the behavior of `Parse`.
- You need to modify the `NewParser` function in `hw3-handout/sexpr/parser.go` to create an instance of the type that satisfies the `Parser` interface.
- We have provided an implementation of a *lexer* in `hw3-handout/sexpr/lexer.go` for the grammar G.

  The lexer performs lexical analysis, converting an input string into a sequence of lexical *tokens*, which correspond to terminals in the grammar G (e.g., `NUMBER`, `SYMBOL`, `QUOTE`, `)`) or the end-of-file (EoF) symbol.

  For example, the lexer turns the input string "`'(a 1)`" into tokens `"QUOTE", "(", "a", "1", ")"`.
- We have provided unit tests the lexer in `hw3-handout/sexpr/lexer_test.go`. These tests should help you understand how to use the lexer in your implementation of the parser.
- Then you need to implement the `Eval` function in `hw3-handout/sexpr/eval.go`, which evaluates an S-expression.
- You are required to implement the evaluation of
  - numbers;
  - Quotations `QUOTE`;
  - `CAR`, `CDR`, `CONS` and `LENGTH`;
  - Unary predicates `ATOM`, `LISTP` and `ZEROP`;
  - Arithmetic operations `+` and `*`. To support arbitrary-precision arithmetic for integers you should use the package big.
- See the tests in `hw3-handout/sexpr/eval_test.go` to understand the behavior of `Eval`. The semantics of MiniLisp CLISP are mostly the same. In some cases, the behavior of MiniLisp might deviate from that in CLISP; this is primarily to simplify the implementation. Please use Piazza if you require further clarification.

- If needed, write new tests in `hw3-handout/sexpr/eval_test.go` and `hw3-handout/sexpr/parser_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw3-handout/sexpr` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw3-handout/sexpr` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

General Tips on Lisp

- 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]> some)nonsense
  <error output>
  ABORT :R3 Abort main loop
  <error output>
  [2]> :R3
  [3]>
  ```

- You can exit the interactive clisp interpreter using:

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