Project 3

Due November 17, 2020 at 9:00 PM

Overview

This project specification is subject to change at any time for clarification. For this programming assignment you will be programming in LISP. This entire project must be purely functional; you are not allowed to use `setq`, `loop` or similar constructs. Points may be docked for use of such constructs.

Fibonacci Sequence Less Than N

Write a function `fibo-lt` to return a list as the Fibonacci sequence such that all numbers in the sequence are less than *n* (**NOT** the first *n* numbers of the sequence). For example:

```
> (fibo-lt '50)
(0 1 1 2 3 5 8 13 21 34)
```

Your function must run in sublinear time and may not hardcode results except for those of the beginning of the sequence such as `'()`, `'(0)` and `'(0 1)`.

Pattern Matching Program

Before we start building the pattern matching function, let us first build a set of routines that will allow us to represent facts, called *assertions*. For instance, we can define the following assertions:

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

Patterns are like assertions, except that they may contain certain special atoms not allowed in assertions, the single `!` character, for instance, or may have strings containing the `*` character.

```
(color ! red)
(su*ts table block1)
```

Write a function match, 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 symbol '!' expands the capability of match by matching zero or more atoms.

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

Here, the first symbol '!' matches `this`, `table` matches `table`, and 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 list (`green red brown`), whereas the `brown` matches the last element.

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

In this example, the '!' matches the empty list. The * character matches zero or more characters inside a string.

```
> (match '(red gr*n blue) '(red green blue))
T
> (match '(t* table is *n) '(this table is blue))
NIL
```

In the first example the *, matches `ee`. In the second example the first * matches `his`, but the second one fails to match because of the `n`. The lone * will match 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.

Erasure Code Reconstruction

Erasure codes are used when a value can be detected as incorrect (or erased). They are commonly used in storage (like in RAID-5) as well as in communication. Write a function

to return a reconstructed message. Each message will be a set of words each with a parity bit followed by a parity word. Each word is represented as a list of `0`, `1` or `NIL` values. The parity bit will be the final value in each word, the parity word will be the final word in the list of words. An example output should be something like:

```
> (parity-correction '((0 1 1 NIL 0) (0 0 0 NIL 1) (NIL 1 1 1 0) (1 0 1
NIL 0) (0 NIL 1 0 1)))
(T ((0 1 1 0 0) (0 0 0 1 1) (1 1 1 1 0) (1 0 1 0 0) (0 0 1 0 1)))
```

This would be equivalent to the following message.

	D_0	D_1	D_2	D_3	P
W_0	**0**	**1**	**1**	0	**0**
W_1	**0**	**0**	**0**	1	**1**
W_2	1	**1**	**1**	**1**	**0**
W_3	**1**	**0**	**1**	0	**0**
W_P	**0**	0	**1**	**0**	**1**

If the message is incorrect or can't be solved NIL should be returned with the original message. For example:

```
> (parity-correction '((0 1 1 NIL 0) (0 0 0 NIL 1) (NIL 1 1 1 0) (1 0 1
NIL 0) (0 NIL 1 0 0)))
(NIL ((0 1 1 NIL 0) (0 0 0 NIL 1) (NIL 1 1 1 0) (1 0 1 NIL 0) (0 NIL 1
0 0)))
```

The `parity-correction` function should return results for modest size messages within a second or two. Submissions that take significant amount of time may lose points.

Scripts of working solutions are in `/home/cjnitta/ecs140a/proj3` on the CSIF. Passing the arguments may require care on bash as a single quote may need to be passed in to the test script.

You **must** submit the source file(s), and README.txt file, in a tgz archive. Do a `make clean` prior to zipping up your files so the size will be smaller. You will want to be in the parent directory of the project directory when creating the tgz archive. You can tar gzip a directory with the command:

```
tar -zcvf archive-name.tgz directory-name
```

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. You must also provide the URL any code sources in comments of your source code. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Helpful Hints

- The command to use Common LISP is clisp.

- Appendix A of LISPcraft summarizes LISP's built-in functions. Each function is explained briefly. You will find this a very useful reference as you write and debug your programs. Also, you can get help about clisp by typing: man clisp

- You may define additional helper functions that your main functions use. Be sure, though, to name the main functions as specified since the test program uses those names.

- If you place a init.lsp file in the directory in which you execute LISP (or your home directory), LISP will load that file automatically when it starts execution. Such a file is useful to define your own environment. For instance, you will probably want to put the following command in that fiile:

  (setq *print-case* :downcase)

- When developing your program, you might find it easier to test your functions first interactively before using the test program. You might find trace, step, print, etc. functions useful in debugging your functions.

- A few points to help the novice LISP programmer:

  - Watch your use of `(`, `)`, `"`, and `'`. Be sure to quote things that need to be quoted.

  - To see how lisp reads your function, use pretty printing. For example,

    ```
    > (pprint (symbol-function 'foo))
    ```

    will print out the definition of function foo, using indentation to show nesting. This is useful to locate logically incorrect nesting due to, e.g., wrong parenthesizing.

  - If you cause an error, Common Lisp places you into a mode in which debugging can be performed (LISPcraft section 11.2). To exit any level, except the top level, type :q. To exit the top level, type

    ```
    > (bye)
    ```