Project 3b

In this project you will explore functional programming and extend a generator of graph input files.

Also, for bonus marks, your can extend the current LISP code base.

Important note

1. In the following, if I say "and test files etc showing *it* working", I expect to find "eg/it.li"and "eg/it.lisp.want" (for the expected output). There may also be, where appropriate, a "eg/it.dat" file containing a sample input file. That is, if you submit right, I should be able to test that part of the code using:

   ```
   make X=it
   ```

2. The following assignment defines 13 deliverables for a standard assignment plus several tasks that could earn bonus marks. Since not everyone may be doing all the bonus points, I need to know what to look for in your code. So, include in your directory "expect.txt" which just lists the deliverables I should look for. This is a very simple file containing one number per line:

   ```
   1
   2
   3
   4
   5
   6
   7
   ```

   Note that everyone has to do "1 2 3 4" but after that, your numbers may change.

Setting up

Downloads

```
cd $HOME
mkdir -p cs310
svn export http://unbox.org/wisp/var/timm/09/310/lib/proj3 proj3b
cd proj3b
chmod +x lisps
./lisps eg/random.lisp
```

Tests

If the above works, you should see something like this:

```
(0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4
 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 7 7 7
 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9 9 9 9 9)
```

How to submit

make submit

Background

The following grammar is a partial description of one of the graph input files:

```
(defparameter *graph-spec*
  '((?spec    -> (?labels ?widths ?heights ?datas))
    (?labels  -> (label ?words nl))
    (?words   -> ?word (?word ?words))
    (?word    -> foo bar blah slug)
    (?widths  -> (width ?num nl))
    (?num     -> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
    (?heights -> (height ?num nl))
    (?datas   -> ?datums2 (?datums2 ?datas))
    (?datums2 -> (?datums1 ?datums1))
    (?datums1 -> (?num ?num ?symbol nl))
    (?symbol  -> \* \# \+)
))
```

In this notation:

- Non-terminal are marked with a question mark; e.g. ?spec
- Terminals are things that are not non-terminals.
- Rules have left-hand sides and right-hand sides, separated by "->".
- Conjunctions are shown as right-hand-side brackets; e.g.

  ```
  ?spec -> (?labels ?widths ?heights ?datas)
  ```

  (so this spec must have labels *and* widths *and* heights *and* data.)

- Disjunctions are shown as right-hand side contents, not bracket Ted; eg.

  ```
  (?word    -> foo bar blah slug)
  ```

  (so word can be either foo *or* bar *or* blah *or* slug.)

By randomly selecting from the disjunctions, it is possible to generate any number of graph input files; e.g. after doing the above set up, you should get:

```

---| grammar.lisp |----------------------------------------



label blah
width 15
height 14
6 7 + 11 16 + 14 8 # 6 17 # 15 10 + 15 3 # 1 19 # 5 1 + 8 7 + 9 9 *

label blah
width 11
height 18
16 20 + 15 9 * 19 8 * 8 3 #

label foo
width 17
height 3
15 14 # 20 16 # 19 10 + 9 6 # 20 1 + 5 3 +

label bar
width 10
height 3
17 7 # 11 17 # 20 16 + 3 20 #

label bar foo slug bar blah
width 15
height 2
19 5 + 4 17 #

label slug
width 11
height 2
```

```
19 3 + 20 3 #

label bar blah
width 19
height 14
18 4 # 17 8 * 18 10 * 8 15 * 9 1 + 17 9 *

label slug
width 20
height 15
2 3 * 19 11 #

label foo slug
width 17
height 16
19 3 * 3 7 #

label foo
width 9
height 18
15 3 + 2 14 +
```

Running the code

As before, there is a file "lisps" that loads all the .lisp files (listed in alphabetically order) into a lisp shell.

If you add arguments to "lisps", you can customize the behavior:

- *./lisps* Loads all the .lisp files, runs what those files want runs, then quits.
- *./lisps -* As above, but then drops the user into an interactive LISP shell.
- *./lisps eg/random.lisp* Runs the example file in eg/random.lisp.

Note that, as before, the standard Makefile commands still run such as

```
make                       --> runs one test
make X=grammar.lisp run    --> runs one test
make X=grammar.lisp cache --> runs one test, caches the results
make tests                 --> runs all tests
make score                 --> scores the tests.
```

If your run "make score" you'll see that there is something wrong with the code:

```
 make score
*** - no handler for label
*** - no handler for bottom
*** - no handler for label
*** - no handler for label
*** - no handler for label
*** - no handler for label
*** - no handler for label
*** - no handler for label
*** - no handler for label
  12 FAILED
   6 PASSED
```

What is going on here is that the LISP code I have supplied can not (yet) draw graphs. For bonus marks, you might choose to fix that.

What to do

There are several things wrong with the above that you must fix:

Easy Stuff (5 marks)

1. The grammar does not describe all the possible graph data inputs (e.g. bottom ticks, amongst other things). Fix that. Note: the grammar can be found around line 100 of grammar.lisp.
   - WHAT TO SUBMIT: (1) a better version of grammar.lisp with a longer grammar for the graph inputs; (2) test files eg/graphgrammer.lisp (and other) demonstrating that you can run some full graph in eg/graphgramer.dat.
2. The grammar does not handle data lines properly
   - There is no new line after each data item.
   - There should be at least two points for each symbol and this is not the case. E.g at line 29 of the above, there is a single "+" and "#".
     - WHAT TO SUBMIT: (3) same file, grammar.lisp, with the grammar changed so it does have the above two issues; (4) test files eg/graphlines.lisp, etc.

Harder stuff (5 marks)

Both the following require problems require a recursive walk through a nested list structure. In LISP, such a recursive descent looks like this:

```
(defun countr (x l)
  (cond ((null l)  0)                    ; case 1) empty list
        ((atom l)  (if (eql x l) 1 0)) ; case 2) not a list
        (t                               ; case 3) isa list so ...
         (let ((n 0))
           (dolist (one l n)             ; we recurse
             (incf n
                   (countr x one)))))))
```

1. The grammar needs a syntax checker. The following checks can be implemented as a recursive walk through the grammar:
   - Flag an error if a RHS non-terminal never appears as the LHS of a rule.
   - Flag an error if a LHS non-terminal never appears in a RHS6
     - WHAT TO SUBMIT: (5) a file checker.lisp, that implements the checker; (6) test files eg/checker.lisp , etc.
2. The grammar output needs a sanity check. For example, when you code "bottom ticks" using "?num" it is possible to get crazy stuff like xmin values that are greater than the xmax values. This can be implemented as a recursive descent through the output of "(generate '?spec)", coded as follows:

   ```
   (dotimes (i 20)
      (let ((spec (generate '?spec)))
         (if (sane spec)
             (spec->lines spec))))
   ```

   Note that not all the specs will be printed. So if your call this with "(dotimes (i 20) .." it is well possible you will only see, say, 18 specs. <6l>

3. WHAT TO SUBMIT: (7) a file sanity.lisp, that implements the checker; (8) test files eg/sanity.lisp , etc.

Bonus Stuff (20 marks)

1. (5 marks) Adjust the grammar to generate science fiction stories of the form described in http://www.yorku.ca/srowley/g_wilson_computer.jpg . To demonstrate this:
   - WHAT TO SUBMIT: (9) a file, scifi.lisp; (10) test files "eg/scifi.lisp" that generates 350 randomly generated files.
2. (5 marks) Add probabilities to each rule. Right now, disjunctions are chosen at random. This is not approrpirate for some applications- sometimes you want to run some things more often than others. Add a number at the rhs of the production showing the probability of that production firing. Skew the firing according to those numbers. The cheat's way of doing this is to input the productions+probabilities and output a new grammar with certain production copied multiple times. Then the above grammar will pick according to your probabilities.

   - WHAT TO SUBMIT: (11) a file, prob.lisp that does the above expansions (12) test files "eg/prob.lisp" etc

3. (5 marks) Take the output to test your Smalltalk code. Here, you succeed if you can (a) do the systems work of automatically passing the LISP output to the Smalltalk and (b) show that you can break the Smalltalk code (some stack dump).
   - WHAT TO SUBMIT:
     - (13) A file "testOut.txt" describing how you did this task and show some of the output (including a crashed Smalltalk stack dump).

   Your file "testOut.txt" should include enough information so I can run your code. If you ha to include your st code, do so using a sub-directory of proj3b.

4. (5 marks) The code I've given you implements a skeleton of the graph generator in LISP. Code it up and show that you can get "18 PASSED" from "make scores".
   - WHAT TO SUBMIT: (14) test files eg/1.lisp ...eg/10.lisp etc.