Lab 12: Macros and Code Generation

Learning objectives:

- Gain experience writing code generators
- Gain experience writing macros in Scheme

Project 5 mainly focuses on code generation for a nontrivial AST, and this lab aims to help you prepare for the work you will do for that project.

For all Scheme coding questions, you must use recursion and you may not use mutation (`set!`, `set-car!`, `set-cdr!`, etc.).

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab12/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Query generation.* In this problem, we will write Python code to generate a query to the `match` predicate from homework 3:

   ```
   % match(Tasks, People, Assignments).
   ```

   a) Implement `prefs_to_tasks()`, which converts a mapping of people to their preferences into a mapping of tasks to person/preference pairs. See the docstring for details.

   Write your implementation in `gen_query.py`. To test your implementation, run the included doctests:

   ```
   $ python3 -m doctest gen_query.py
   ```

   b) Now complete the `gen_query()` function, which prints a generated query to standard out. You must use `prefs_to_tasks()`. You may also use the `list_to_string()` function, which converts a nested list structure into a single string, as in the following example:

   ```
   >>> list_to_string([['fe', 1], ['gwyn', 1]])
   '[[fe, 1], [gwyn, 1]]'
   ```

   Since the output may vary, you will need to check your results by hand. The following invokes `gen_query()` on a simple test case:

   ```
   $ python3 gen_query.py
   ```

2. *Scheme macros.* Implement the `my-or` macro in Scheme, which should behave exactly like the or library syntax:

   ```
   > (define x 0)
   > (my-or (= x 0) (/ 3 x))
   #t
   > (define x 1)
   > (my-or (= x 0) (/ 3 x))
   3
   ```

   You may use the built-in `if` form in your solution.

   Write your implementation in `my-or.scm`. To test your implementation, run the included tests:

   ```
   $ plt-r5rs my-or.scm
   ```

3. *More code generation.* Recall that in Project 3, we implemented type checking on primitive procedures. For instance, the procedures `car` and `cdr` both require their argument to satisfy the `pair?` predicate.

   Suppose we want to also support combinations such as `cddr` and `cddar`, which are composed versions of `car` and `cdr`. To be valid, the argument of such a combination must have a nested-pair structure. For example, the argument to `cddar` must satisfy the following predicate:

```
(lambda (x) (and (pair? x) (pair? (car x)) (pair? (cdar x))))
```

In this problem, we will write Python code to generate the appropriate predicate for a `c*r` combination. Implement the `gen_predicate()` function, which takes a string composed of `a`'s and `d`'s. For example, we would invoke `gen_predicate('dda')` to obtain the predicate for `cddar`, which should be returned as a string (`gen_predicate()` should not print anything to standard out):

```
>>> gen_predicate('dda')
'lambda (x) (and (pair? x) (pair? (car x)) (pair? (cdar x))))'
```

Your implementation for `gen_predicate()` should work on any string of length at least two that consists of a combination of `a`'s and `d`'s.

Write your implementation in `gen_predicate.py`. To test your implementation, run the included doctests:

```
$ python3 -m doctest gen_predicate.py
```