Lab 12: Code Generation and Template Metaprogramming

Learning objectives:

- Gain experience writing code generators
- Understand how template substitution works in C++
- Understand how function overloading works in combination with templates in C++

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.

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. *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
   ```

2. *Function templates.* In Python, a string can be multiplied by a non-negative integer *N*, which evaluates to a new string with the original string repeated *N* times:

   ```
   >>> 'abc' * 3
   'abcabcabc'
   >>> 3 * 'z'
   'zzz'
   ```

   Write a set of overloads in `mult.hpp` for a `mult()` function in C++ that performs this string repetition when the one argument is a string and the other a non-negative integer. On the other hand, if the two arguments both have numerical type, then `mult()` multiplies the two arguments and returns the result. The following are examples of calling `mult()`:

   ```
   std::cout << mult(3, 4.1) << std::endl;   // prints 12.3
   std::cout << mult("abc", 3) << std::endl; // prints abcabcabc
   std::cout << mult(3, "z") << std::endl;   // prints zzz
   ```

   Run the test cases as follows:

   ```
   $ g++ --std=c++20 mult_test.cpp -o mult_test.exe
   $ ./mult_test.exe
   ```