Week 4: Macros and Object-Oriented Programming

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science
“OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.”

–Alan Kay (inventor of Smalltalk)

**Sending messages**

What’s the most important punctuation mark in OOP?
An object is a mapping of messages to values. Program interacts with an object by sending it messages.

A class is a “template” used to create objects with the same behaviour.
Problem: too much boilerplate code!

I want `class`.

Macros
**macro**: a transformation on program syntax

“syntax -> syntax” or “AST -> AST” function

**Interpreter pipeline**

source code

↓ *(parsing)  ↓*

abstract syntax tree

↓ *(evaluation)  ↓*

program value
We use functions to extract computational boilerplate.
We use macros to extract syntatic boilerplate.
Pattern-based macros

```scheme
(define-syntax <syntax-name>
  (syntax-rules ()
    [<pattern> <template>] ...))
```

Let’s make `class`.

Macro ellipses
In a pattern, ... operates like Kleene star *.

```
<attr> ...
(<x> <y>) ...
(<a> ...) ... 
```

In a template, operates like map.

```
(list <attr> ...)
(list (+ <x> <y>) ...)
(list (+ <a> ...) ...) 
```
Macros vs. functions

Warmup

Why can’t `my-class` be defined as a function?

```scheme
(my-class Point (x y))
```
Syntax vs. runtime errors

Using a pattern-based macro can have one of these results:

1. No patterns match the expression.
2. A pattern matches, but the expanded expression is not semantically valid.
3. A pattern matches, and the resulting expression is semantically valid.
AST-based vs. text-based macros

In C/C++, macros operate at the level of source code, rather than the AST. This is very dangerous!
Problem 1: Expression structure is not preserved (by default)

Problem 2: Name capture

Identifiers in C macros obey dynamic scope!
In Racket’s pattern-based macros, identifiers obey *lexical scope*; literal identifiers bound in a macro template do not interact with where the macro is used.

Another name for “lexically-scoped” macros is **hygienic macros**.

---

**Wait…**

```lisp
(my-class Point (x y))
; Now Point is in scope!
(define p (Point 2 3))
```

Identifiers passed as macro arguments *can* be bound by the macro and have scope that extends outside the macro template.
Back to my-class: methods

Code demo!
Literal keywords

```
(define-syntax <syntax-name>
  (syntax-rules (<keyword> ...) 
    [<pattern> <template>] ...))
```

Problem: method that calls another method?
The problem of self

An instance method should be able to refer to the calling object.

(“I should be able to refer to myself.”)
Approach 1: `this/self` as a built-in keyword
(e.g., Java, C++, Ruby, JavaScript)

Approach 2: `this/self` as a method parameter
(e.g., Python, Rust)
Implementation first try: self everywhere!

```lisp
(my-class (Point (x y))
  (method (size self)
    (sqrt (+ (* (self 'x) (self 'x))
            (* (self 'y) (self 'y)))))))
```