Laziness and accumulators
(leftover from last week)

Object-Oriented Programming

Principles and implementation

“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)

“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

(aka “the most important punctuation mark in OOP”)

Sending messages

(aka “the most important punctuation mark in OOP”)

An object is a **mapping** of messages to values.

An object responds to a message by looking up the corresponding value.

A **class** is a “template” used to create objects with the same behaviour.

```
(define (Point x y)

  (let* ([__dict__
          (make-immutable-hash
           (list (cons 'x x)
                 (cons 'y y)
                 (cons 'scale
                       (lambda (factor)
                         (Point (* x factor) (* y factor))))))])

    (lambda (msg)
      (hash-ref __dict__ msg
                ; Raise an error if not message not found.
                (attribute-error 'Point msg)))))
```

Problem: too much boilerplate code!

I want `class`.

macro: a transformation on program syntax

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

A simple interpreter pipeline

$$\begin{tikzcd}
\text{source code} \arrow[d, "\textit{parsing}"] \\
\text{abstract syntax tree} \arrow[d, "\textit{evaluation}"] \\
\text{output}
\end{tikzcd}$$

A simple interpreter pipeline

source code

↓ *parsing*

abstract syntax tree

↓ ***macro expansion***

abstract syntax tree

↓ *evaluation*

output

Pattern-based macros

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

We use functions to extract computational boilerplate.
We use macros to extract syntactic boilerplate.

Let’s make `class`.

Macro ellipses

In a pattern, operates like Kleene star: *

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

Macro ellipses

In a template, operates like `map`

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

Syntax error vs. runtime error

Usage of a pattern-based macro can have one of three outcomes:

1. No patterns match the expression
2. A pattern matches, but the resulting expression is not semantically valid
3. A pattern matches, and the resulting expression is semantically valid

Identifier bindings in macros

1. Looking up identifiers obeys lexical scope (i.e., is based on where the macro is defined).

2. Identifiers written and bound in a macro are not visible outside the macro body.

Extending our **class** macro: methods!

Syntax keywords

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

The problem of `self`