The evolution of a DSL?

general-purpose → domain-specific → general-purpose?

library

- discover nouns & verbs

API

- add fluency

internal

- remove host flavor

external

- tools & environments

features

We are here!
A taste of metaprogramming
Python decorators

@decorator
def f():
...

≈

def f():
...
    f = decorator(f)
Something more useful

def logging(f):
    def wrapper(*args, **kwargs):
        print 'Calling {0} with {1} and {2}'.format(f.func_name, args, kwargs)
        return f(*args, **kwargs)
    return wrapper

@logging
def f(x, y):
    return x+y
from functools import wraps

def logging(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        print 'Calling {0} with {1} and {2}'.format(f.func_name, args, kwargs)
        return f(*args, **kwargs)

    return wrapper

@logging
def f(x, y):
    return x+y
from functools import wraps

class logging(object):
    def __init__(self, handle=sys.stdout):
        self.handle = handle

    def __call__(self, f):
        @wraps(f)
        def wrapper(*args, **kwargs):
            print >> self.handle, "Calling {0} with {1} and {2}".format(f.func_name, args, kwargs)
            return f(*args, **kwargs)

        return wrapper

@logging(file('log.txt', 'a'))
def g(x,y):
    return x**y
Another decorator

from functools import wraps

def memoize(f):
    cache = {}

    @wraps(f)
    def wrapper(*args, **kwargs):
        key = (args, tuple(kwargs.values()))
        if key not in cache:
            cache[key] = f(*args, **kwargs)
        return cache[key]

    return wrapper

@memoize
def g(x, y):
    return x**y
Decorators are composable

```python
@memoize
@logging
def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1) + fib(n-2)
```
make(p):

if potion p is not yet in the pantry:
    foreach potion $p_i$ in $p$’s ingredients:
        make($p_i$)
        add $p$ to the pantry
    proclaim “Made $p$!”

fetch $p$ from the pantry
@ingredients([])
def Atominite(): pass

@ingredients([Atominite])
def Bicarbonite(): pass

@ingredients([Bicarbonite])
def CockroachCocktail(): pass

@ingredients([])
def Diatonic(): pass

@ingredients([Atominite, Diatonic])
def EssenceOfNight(): pass

@ingredients([EssenceOfNight])
def Florabinitne(): pass

@ingredients([Diatonic])
def Grapplegum(): pass
Scala Documentation

- Views
- Iterators
- Creating Collections From Scratch
- Conversions Between Java and Scala Collections
- Migrating from Scala 2.7

- The Architecture of Scala Collections
- String Interpolation
- Implicit Classes
- Value Classes and Universal Traits

Reference / Documentation

- Scaladoc
  - Overview
  - Using Scaladoc Effectively
  - Authoring Scaladoc

- Scala REPL
  - Overview

Parallel and Concurrent Programming

- Futures and Promises
- Scala's Parallel Collections Library
  - Overview
  - Concrete Parallel Collection Classes
  - Parallel Collection Conversions
  - Concurrent Tries
  - Architecture of the Parallel Collections Library
  - Creating Custom Parallel Collections
  - Configuring Parallel Collections
  - Measuring Performance

- The Scala Actors Migration Guide
- The Scala Actors API

Metaprogramming

- Reflection
  - Overview
  - Environment, Universes, and Mirrors
  - Symbols, Trees, and Types
  - Annotations, Names, Scopes, and More
  - TypeTags and Manifests
  - Thread Safety
  - Changes in Scala 2.11

- Macros
  - Use Cases
  - Blackbox Vs Whitebox
  - Def Macros
  - Quasiquotes