CS 320: Principles of Programming Languages

Mark P Jones, Jingke Li, and Andrew Tolmach

Fall 2019

Week 8: Functional Languages

Language “Paradigms”

- Procedural
- Object-oriented
- Functional
- Logic
- Scripting
- Concurrent
- …

What makes a paradigm?

style
organization
expressiveness
purpose

many languages support multiple paradigms

What is functional programming?

- Using functions as **first-class values**
  - Can be stored in data structures
  - Can be passed to/returned from other functions
  - Can be defined anonymously using lambda expressions
  - Can be defined locally within other functions
- Keeping expressions and functions **pure** (no side-effects)
  - Use immutable variables and data structures
  - Support "computing by calculation"
  - Contrasts to conventional **imperative** programming

Programming with first-class/higher-order functions

First-class functions

- In addition to being called with arguments, a function can also be treated as an ordinary "first-class" value, e.g.
  - **passed as a parameter** to another function

```
list = [('a',1),('b',3),('c',2)]

def fst(e): return e[0]
def snd(e): return e[1]

sorted(list,key=fst) ⤳
    [('a', 1), ('b', 3), ('c', 2)]
sorted(list,key=snd) ⤳
    [('a', 1), ('c', 2), ('b', 3)]
```

function to apply to each element to extract sort key

(Python)

First-class functions

- In addition to being called with arguments, a function can also be treated as an ordinary "first-class" value, e.g.
  - **stored** into a variable

```
list = [('a',1),('b',3),('c',2)]

def fst(e): return e[0]
def snd(e): return e[1]

if use_char_sort():
  mykeyfn = fst
else:
  mykeyfn = snd

sorted(list,key=mykeyfn) ⇝
```

some function returning a boolean

a function-valued variable

result depends on which key function was picked

(Python)

First-class functions

- In addition to being called with arguments, a function can also be treated as an ordinary "first-class" value, e.g.
  - **returned** from a function

```
list = [('a',1),('b',3),('c',2)]

def fst(e): return e[0]
def snd(e): return e[1]

def sort_key_to_use():
  if ...:
    return fst
  else:
    return snd

mykeyfn = sort_key_to_use()
sorted(list,key=mykeyfn) ↝
```

some boolean condition

returns a function!

holds function value that is returned

result depends on which key function was returned

Anonymous functions at work

- It is tedious to give names to small functions that are used only once (e.g. as arguments to other functions)
- This is where **lambda** expressions come in handy
- Written thus in Python: **`lambda`** *`args`*`:` *`return-expr`*

```
list = [('a',1),('b',3),('c',2)]

sorted(list,key=lambda e: e[0]) ⇝
    [('a', 1), ('b', 3), ('c', 2)]
sorted(list,key=lambda e: e[1]) ⇝
    [('a', 1), ('c', 2), ('b', 3)]
```

(Python)

Higher-order functions

- Functions that take other functions as arguments or return other functions as results are sometimes called higher-order functions:

```
def adder(x:Int): Int=>Int = {
  def g(z:Int):Int = {return x+z;};
  return g;
}
val add3 = adder(3)
val x = add3(5) // evaluates to 8
```

an type for functions that take an `int` argument and return an `int` result

free variable

(Scala)

- In this case, `g` (alias `add3`) is called after `adder` has returned
- ... so `g` must access `adder`'s `x` parameter after `adder` has returned
- The value that `adder` returns must be a combination of the code for `g` and the value for `x`

Using a lambda expression

- The previous example:

```
def adder(x:Int): Int=>Int = {
  def g(z:Int):Int = {return x+z;};
  return g;
}
```

- Rewritten using a lambda expression:

```
def adder(x:Int): Int=>Int = {
  return z => x+z;
}
```

free variable

bound variable

(Scala)

Using function values

- A general purpose (mutating) "mapping" primitive:

```
def mapArray(f:Int=>Int,a:Array[Int]) = {
  for (i <- 0 until a.length)
    a(i) = f(a(i))
}
```

(Scala)

Note that this `f` is a parameter of `mapArray`, not a known function

- To increment every element in an array, `arr`:

```
mapArray(adder(1), arr)
```

- To double every element in an array, `arr`:

```
mapArray(x => x * 2, arr)
```

- Etc...

Composing function values

- A general purpose "composition" primitive:

```
def compose(f:Int=>Int,g:Int=>Int):Int=>Int = {
    return x => f(g(x))
}
```

(Scala)

- Using `compose`, we can combine two separate mapping operations:

```
mapArray(g, arr)
mapArray(f, arr)
```

- `into` a single iteration across the array:

```
mapArray(compose(f, g), arr)
```

- e.g., to convert every element to a corresponding odd number

```
mapArray(compose(adder(1),x=>x*2), arr)
```

Representing function values

- How should we represent values of type `int=>int`?
  - There are many different values, including: `(z => x+z)`, `(x => x+1)`, `(x => x*2)`, `(x => f(g(x)))`, ...
  - ... any of which could be passed as arguments to functions like `mapArray` or `compose`, ...
  - ... so we need a uniform, but flexible way to represent them

- A common answer is to represent functions like these by a pointer to a "closure", which is an object that contains:
  - a code pointer (i.e., the code for the function)
  - the values of its free variables

Closures

- *Every* function of type `int=>int` will be represented using the same basic structure:

`codeptr`	`...`

- The code pointer and list of variables vary from one function value to the next:

`(z => x+z)`	$\texttt{codeptr}_1$	`x`	
`(x => x+1)`	$\texttt{codeptr}_2$		
`(x => x*2)`	$\texttt{codeptr}_3$		
`(x => f(g(x)))`	$\texttt{codeptr}_4$	`f`	`g`

- To make a closure, allocate a suitably sized block of memory and save the required code pointer and variable values

Closures vs. objects

- Invoking an unknown function through a closure is very similar to invoking a method of an object …
  - Method invocations pass the object itself as an implicit argument (just as we are pass the closure pointer here)
  - Closures are like objects with a single method
  - Free variables correspond to object fields
- In Java 8, lambda expressions are "just" a convenient way to write (local, anonymous) definitions of single-method classes
  - Very useful for GUI call-backs, aggregate operations, concurrency libraries, etc…

Simulating closures in Java

We can simulate the use of closures using Java classes:

```
interface IntToInt {                          //   int => int type
  abstract int apply(int arg);
}

class PlusOne implements IntToInt {   //   x => x + 1
  public int apply(int arg) { return arg + 1; }
}

class TimesTwo implements IntToInt {     //   x => x * 2
  public int apply(int arg) { return arg * 2; }
}

class PlusX implements IntToInt {     //   z => x + z
  private int x;
  PlusX(int x)     { this.x = x; }
  public int apply(int arg) { return x + arg; }
}
```

(Java)

Mapping over an array

- A general purpose (mutating) “mapping” primitive:

```
static void mapArray(IntToInt f, int[] arr) {
    for (int i=0; i<arr.length; i++) {
        arr[i] = f.apply(arr[i]);
    }
}
```

(Java)

- To increment every element in an array :

```
mapArray(new PlusOne(), arr);
```

- Using an “inner” class:

```
mapArray(new IntToInt() {
            int apply(int arg) { return arg+1; }
         }, arr);
```

- Using a lambda expression:

works because `IntToInt` has exactly one method

```
mapArray(arg -> arg+1, arr);
```

Composing functions

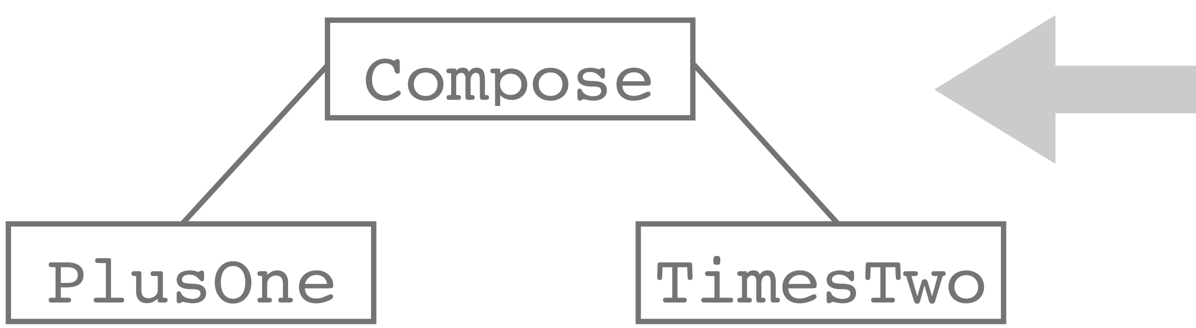
- A general purpose "composition" primitive:

```
class Compose implements IntToInt {
    private IntToInt f;
    private IntToInt g;
    Compose(IntToInt f, IntToInt g)
      { this.f = f; this.g = g; }
    int apply(int arg) { return f.apply(g.apply(arg)); }
}
```

(Java)

- To replace each value in an array with a corresponding odd number:

```
mapArray(new Compose(new PlusOne(), new TimesTwo()), arr);
```



abstract syntax, with an interpreter/eval function called "`apply`"!

Summary: First-class functions

- Common patterns of computation can be abstracted out as general purpose, higher-order functions that provide new opportunities for code reuse and modularity
- Function values can be represented by closure objects that pair a code pointer with a list of variable values
- Invoking an unknown function through a closure is very similar to invoking a method of an object ...
- These techniques are key tools in the implementation of functional (and, increasingly, OO) programming languages

Effects vs. Purity:

Computing by Calculating

Reminder: side effects

- A function or expression is said to have a **side effect** if, in addition to producing a value, it also modifies a variable or memory, or performs I/O

```
++i      (a = 3) > b      printf("hi")
```

(C)

- Functions in an imperative language often have side effects:

```
void sum(int *s, int a) { *s += a; }

int c = 0;
int count() { return ++c; }

int safe_divide(int x, int y) {
  if (y == 0) { printf("oops\n"); return 0;}
  return x/y; }
```

(C)

Referential Transparency and Purity

- An expression is **referentially transparent** if it can be replaced by its value without changing the behavior of the program
  - Expressions that have no side-effects and no input from the environment are referentially transparent
- A **pure** function is one whose body is referentially transparent, so its value depends only on its arguments, and it has no side effects
  - Some pure functions in C library: **`atan()`**, **`strlen()`**
  - Some impure functions in C: **`printf()`**, **`rand()`**

    -- performs output, reads/updates global seed value

Benefits of working in a pure language

- We can transform programs freely using **equational reasoning**
  - e.g. **`f(x)+f(x)`** is always equivalent to **`2*f(x)`**
- We can loosen evaluation order:
  - The arguments of $\mathtt{f(e_1,e_2,...e_n)}$ can be evaluated in any order or in **parallel**
  - Argument evaluation can be deferred until the parameter is known to be needed ("**lazy evaluation**")
- We can compute program behavior by **calculation**

Degrees of purity

- Some functional languages are pure: all expressions are referentially transparent
  - Example: Haskell allows side effects only in the sense that the entire program can be an "IO action" that transforms the state of the real world
- Other functional languages are impure: they discourage side effects, but do permit them
  - Examples: Scheme, OCaml
- Many imperative languages have libraries favoring purity
  - Example: Java streams, Guava immutable collections library

Programming in pure languages

- Pure functional languages feel different from imperative ones in ways that go beyond syntax
- Many familiar things are missing:
  - mutable variables and assignment
  - mutable data structures such as arrays
  - iterative loops controlled by an index variable
- Instead, we have:
  - immutable "variables"
  - immutable data structures (especially lists and trees)
  - recursion

Recursion

- In functional languages, recursion replaces loops
  - Index variables are useless if we cannot update them!

```
int fac(int n) {
  int r = 1;
  for (int i = 2; i <= n; i++)
    r = r * i;
  return r;
}
```

(C++)

```
int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

potentially more expensive because we incur the overhead of many calls

(C++)

Tail Recursion

- Certain forms of recursion can be optimized by a compiler to use iteration "under the hood"
  - A function is **tail recursive** if it never does any work after returning from a recursive call

not tail recursive

```python
def fact(n):
  if n > 1:
    return n*fact(n-1)
  else:
    return 1
```

Python

tail recursive

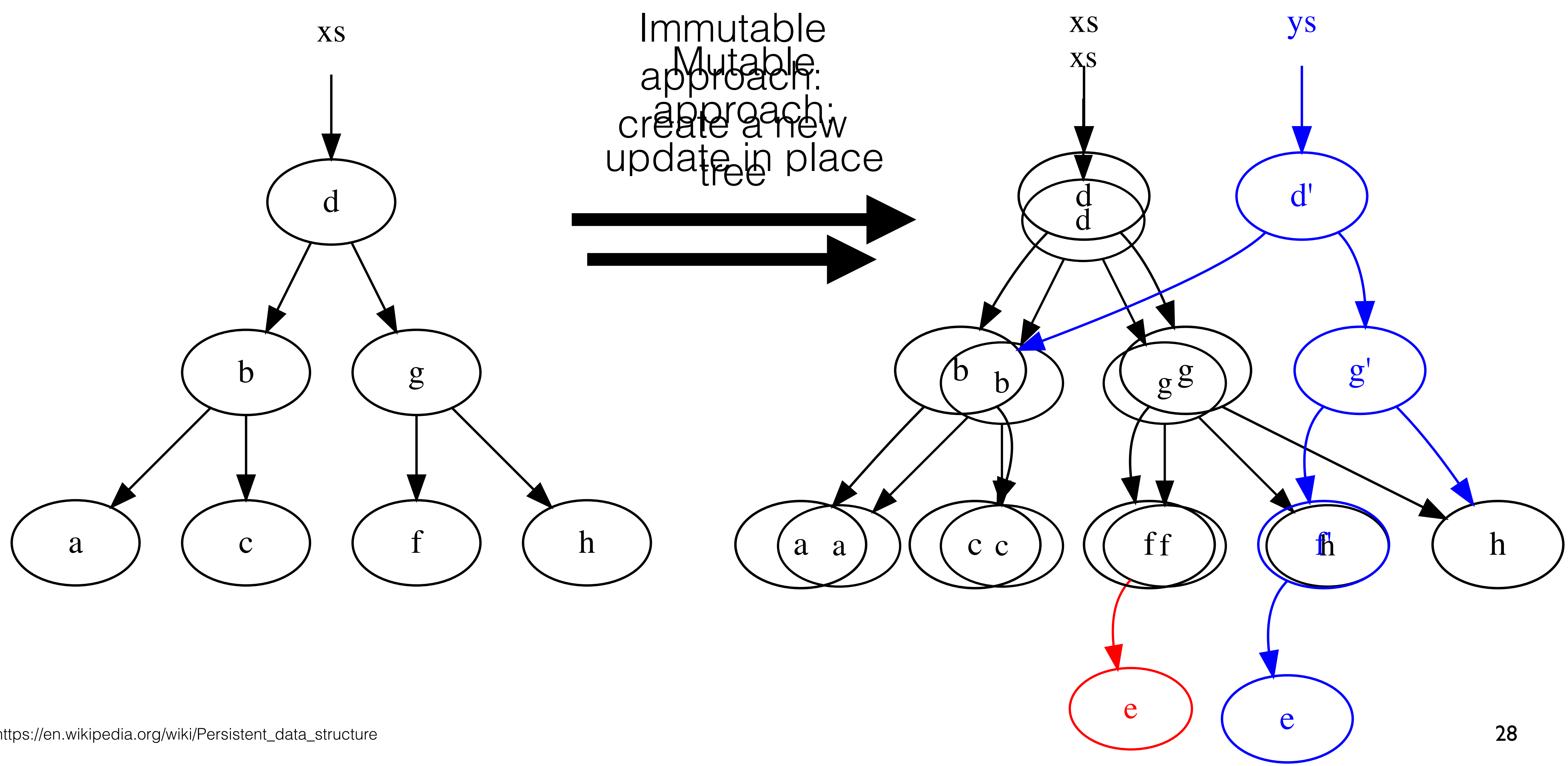
```python
def fact(n):
  def f (n,a):
    if n > 1:
      return f(n-1,n*a)
    else:
      return a
  return f(n,1)
```

Python

may compile to faster code

Immutable data structures

- Instead of modifying data structures in place, pure programs must build new data structures, leaving the old ones in place
- Requires more copying, but sharing can help a lot
  - Example: inserting an element 'e' into tree xs



https://en.wikipedia.org/wiki/Persistent_data_structure

A simple calculation example

(Python)

```
def sz(s):
  if s:
    return 1 + sz(s[1:])
  else:
    return 0
```

(Python)

```
def sz(s):
  c = 0
  while s:
    c += 1
    s = s[1:]
  return c
```

sz("abc")
= 1 + sz("bc")
= 1 + (1 + sz("c"))
= 1 + (1 + (1 + sz("")))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
= 1 + 2
= 3

this approach to calculation doesn't work in imperative style!

A bigger calculation example

(Haskell)

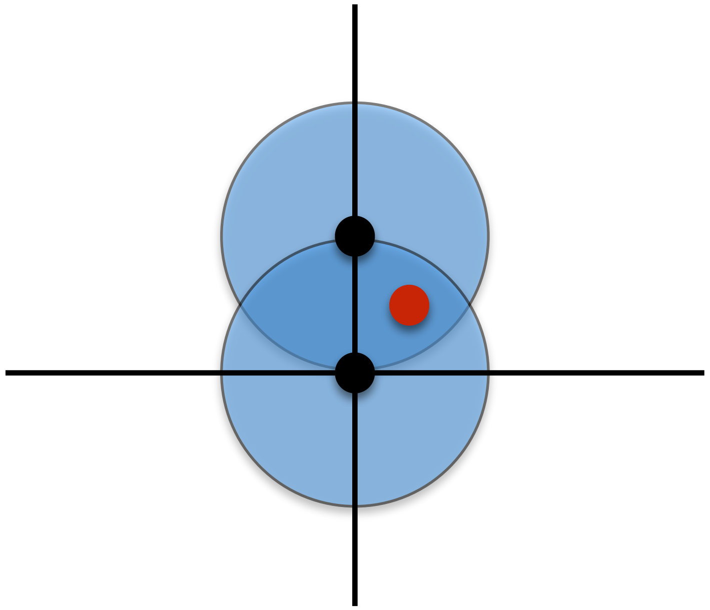
```
type Point = (Double,Double)
distance :: Point -> Point -> Double
distance (x1,y1) (x2,y2) =
               sqrt((x1 - x2)^2 + (y1 - y2)^2)

type PSet = Point -> Bool

in :: Point -> Pset -> Bool
p `in` ps = ps p

intersect :: PSet -> PSet -> PSet
intersect ps1 ps2 =
     \point -> ps1 point && ps2 point

disk :: Point -> Double -> PSet
disk center radius =
     \point -> distance center point <= radius
```

A bigger calculation example (2)

(Haskell)

```
myregion :: PSet
myregion = disk (0,0) 2 `intersect` disk (0,2) 2
```

(1,1) \`in\` myregion
= myregion (1,1)
= (disk (0,0) 2 \`intersect\` disk (0,2) 2) (1,1)
= disk (0,0) 2 (1,1) && disk (0,2) 2 (1,1)
= distance (0,0) (1,1) <= 2
&& distance (0,1) (1,1) <= 2
= … = sqrt(2) <= 2 && sqrt(2) <= 2
= True && True
= True



Eager vs. Lazy evaluation

- In a pure language, order of evaluation cannot affect the result of a computation (except for whether it terminates)
- Most languages use eager evaluation: evaluate arguments before invoking function call

(Haskell syntax)

```
g :: Int -> Int -> Int -> Int
g i x y = if i > 0 then x + x else y + y

a = g 1 42 (fact 1000000)
b = g 0 0  (fact 1000000)
```

- Under eager evaluation, evaluating either **`a`** or **`b`** will cause **`(fact 1000000)`** to be evaluated before entering the body of **`g`**

Eager vs. Lazy evaluation

(Haskell)

```
g :: Int -> Int -> Int -> Int
g i x y = if i > 0 then x + x else y + y

a = g 1 42 (fact 1000000)
b = g 0 0  (fact 1000000)
```

- Under lazy evaluation (e.g. in Haskell):
  - evaluating **`a`** doesn't require **`(fact 1000000)`** to be evaluated at all, because **`y`** is not needed to compute the return value of **`g`**
  - evaluating **`b`** causes **`(fact 1000000)`** to be evaluated (but just once) because **`y`** is needed to compute the return value of **`g`**