Anonymous Functions and Problem Decomposition

September 14, 2001

Review

- In the last class you saw several rex *predicates*
  - Functions returning true or false (1 or 0)
  - Examples: `null`, `even`, `is_prime`, `member`, …
- You also saw several *higher-order* functions
  - Functions that take functions as arguments, or return functions as results.
  - Examples: `map`, `map`, `reduce`, `keep`, `drop`, `all, some`

`iterate`

- Keeps applying a function to an argument as many times as is necessary until it fails to satisfy a given predicate.
  - The call `iterate(action, continue, state)` looks at the sequence `state`, `action(state)`, `action(action(state))`, … and returns the first of these values which does not satisfy the `continue` predicate

```
successor(n) = n+1;
next_highest_composite(n) =
  iterate(successor, is_prime, n+1);
```

Anonymous Functions

- Functions have a meaning independent of the names we give them.
- We want a way to refer to a function without giving it a name.
- Notation:

  `(X) => … some expression …`

  means "the function that, with argument `X`, returns the value of … some expression ….

Example

- The function `is_zero`, defined by:

```
is_zero(X) = (X == 0);
```

can also be written anonymously:

```
(X) => (X == 0)
```

"the function that, with argument `X`, returns the value of `X == 0`".

- So we could define

```
drop_zeros(L) = drop(is_zero, L)
```

if `is_zero` has been previously defined, and otherwise

```
drop_zeros(L) = drop((X) => (X == 0), L)
```

Precedent

- This notation for talking about a function goes back to (at least) Bourbaki, where the symbol

$\mapsto$

was used instead of

`=>`

- Church used the idea extensively, but with a different symbol λ as a *prefix*.

More Anonymous Functions

`(X) => X+5`	The function that adds 5
`(X) => X*5`	The function that multiplies by 5
`(X) => X*X`	The function that squares
`(X, Y) => Y/X`	The function that divides its second argument by its first.

Anonymous Functions with "Imported" Values

```
drop_multiples(X, L) =
    drop((Y) => (Y%X == 0),  L)
```

The predicate that tests divisibility by X.

- The book refers to X as being *imported* by the anonymous function
  - It is used by, but not an argument to, the anonymous function.

Exercises

- Give an equation defining `scale` using `map`

- Define `pairWith`, such that `pairWith(X, L)` creates a list in which each element of `L` is paired with `X`:

```
pairWith(3, [1,2,3]) ==> [[3,1], [3,2], [3,3]]
```

Exercises

- Define a function pairs that given two list returns their Cartesian product
  - List of all pairs of an element from the first list with an element of the second list:

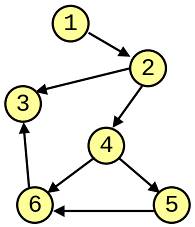
```
pairs([1, 2, 3], [4, 5, 6]) ==>
        [[1, 4], [1, 5], [1, 6], [2, 4], [2, 5],
        [2, 6], [3, 4], [3, 5], [3, 6]]
```

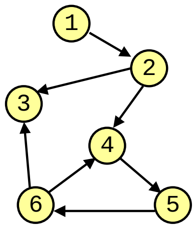
Function Decomposition

- To solve problems using functions, we typically:
  - Express the problem informally as a function, with input and output.
  - Break down the function as a composition of simpler functions.
  - Repeat this process, until we are using only functions that are built-in.

Decomposition Example

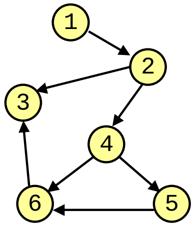
- Problem: Given a graph determine whether it contains a cycle or not.





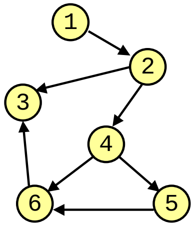
Key Observation 1

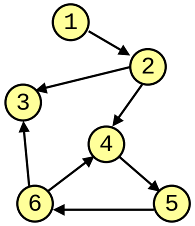
- If a finite graph is acyclic, it must have a leaf
  - That is, a node with no outgoing edges.
- Why?



Key Observation 2

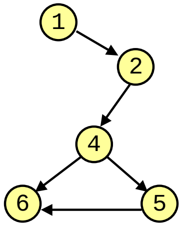
- If we remove a leaf and any attached arcs from a graph, it can't change it from cyclic to noncyclic or vice-versa.

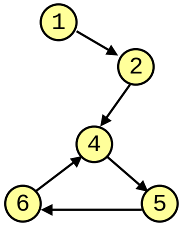




Key Observation 2

- If we remove a leaf and any attached arcs from a graph, it can't change it from cyclic to noncyclic or vice-versa. Why?





So…

- How could we solve the problem using these two key ideas?