CMSC 330: Organization of Programming Languages

Lambda Calculus Introduction

Introduction

- We've seen that several language conveniences aren't strictly necessary
  - Multi-argument functions: use currying or tuples
  - Loops: use recursion
  - Side-effects: we don't need them either
- Goal: come up with a "core" language that's as small as possible and still Turing complete
  - This will give a way of illustrating important language features and algorithms

Revised Rule for Application

$$\frac{A;\ E_1 \rightarrow (A',\ \lambda x.E) \qquad A;\ E_2 \rightarrow v \qquad A,\ A',\ x{:}v;\ E \rightarrow v'}{A;\ (E_1\ \ E_2) \rightarrow v'}$$

- To apply something to an argument:
  - Evaluate it to produce a closure
  - Evaluate the argument (call-by-value)
  - Evaluate the body of the closure, in
    - The current environment, extended with the closure's environment, extended with the binding for the parameter

Example

$$\frac{\bullet;\ (\text{fun } x = (\text{fun } y = +\ x\ y)) \rightarrow (\bullet,\ \lambda x.(\text{fun } y = +\ x\ y)) \qquad \bullet;\ 3 \rightarrow 3 \qquad x{:}3;\ (\text{fun } y = +\ x\ y) \rightarrow (x{:}3,\ \lambda y.(+\ x\ y))}{\bullet;\ (\text{fun } x = (\text{fun } y = +\ x\ y))\ \ 3 \rightarrow (x{:}3,\ \lambda y.(+\ x\ y))}$$

Lambda Calculus

- A lambda calculus expression is defined as

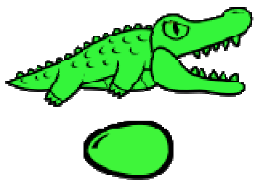
e ::= x	variable
| λx.e	function
| e e	function application

- λx.e is like `(fun x -> e)` in OCaml

- That's it! Only higher-order functions

Intuitive Understanding

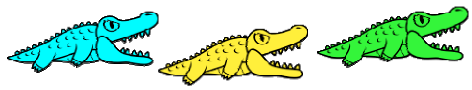
- Before we work more with the mathematical notation of lambda calculus, we're going to play a puzzle game!



- From: http://worrydream.com/AlligatorEggs/

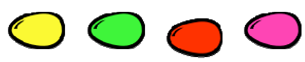
Puzzle Pieces

- Hungry alligators: eat and guard family

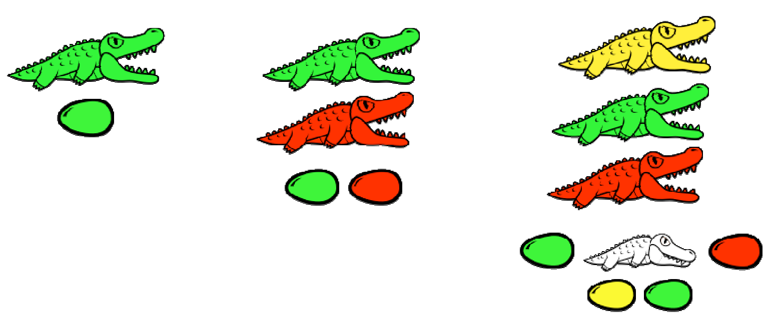


- Old alligators: guard family

- Eggs: hatch into new family



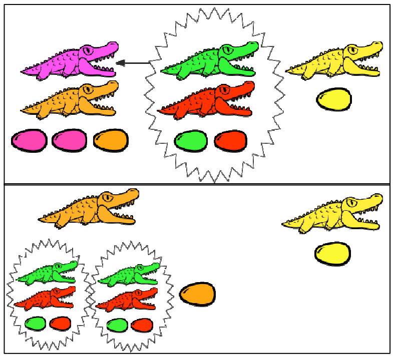
Example Families



- Families are shown in columns
- Alligators guard families *below* them

Puzzle Rule 1: The Eating Rule

- If families are side-by-side the top left alligator eats the entire family to her right
- The top left alligator dies
- Any eggs she was guarding of the same color hatch into what she just ate

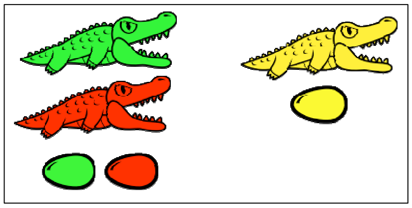


CMSC 330 9

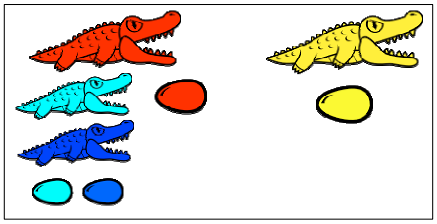
Eating Rule Practice

- What happens to these alligators?

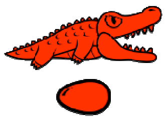
Puzzle 1:



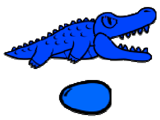
Puzzle 2:



Answer 1:



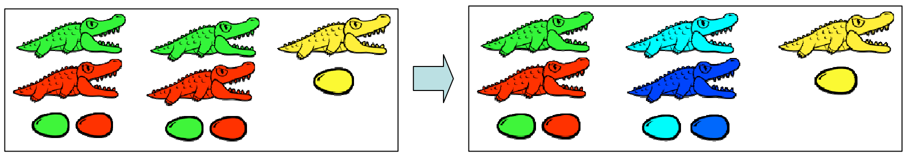
Answer 2:



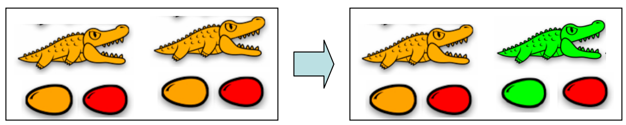
CMSC 330 10

Puzzle Rule 2: The Color Rule

- If an alligator is about to eat a family and a color appears in *both families* then we need to change that color in one of the families.

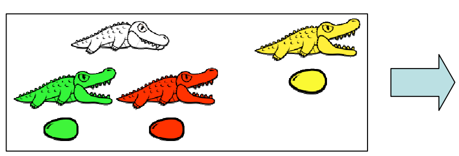


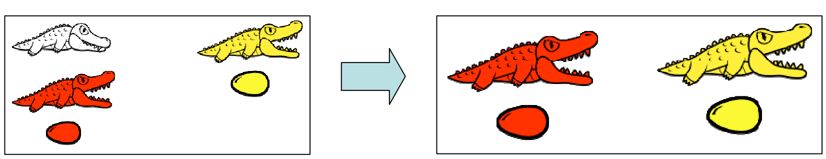
- If a color appears in both families, but *only* as an egg, no color change is made.



Puzzle Rule 3: The Old Alligator Rule

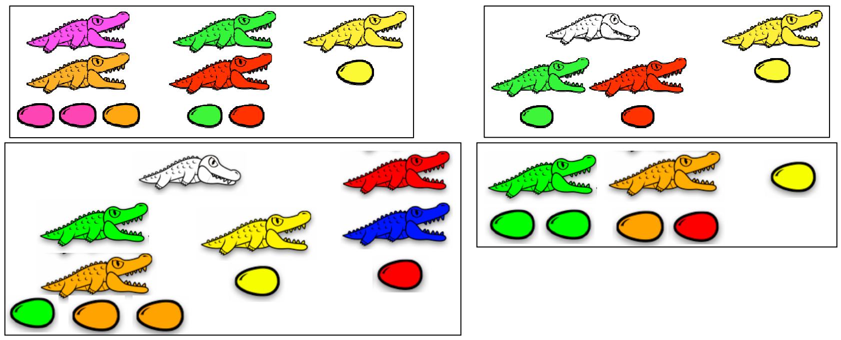
- When an old alligator is only guarding one family it dies.





Challenging Puzzles!

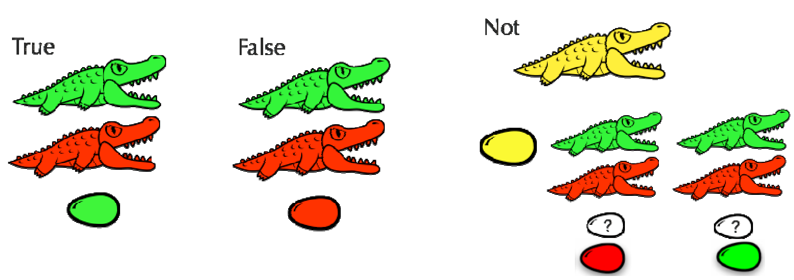
- Try to reduce these groups of alligators as much as possible using the three puzzle rules:



- Challenge your neighbors with puzzles of your own.

More Puzzles

- When Family Not eats Family True it becomes Family False and when Not eats False it becomes True… what color should the white eggs be?



- What do the AND and OR families look like?

Lambda Calculus

- A lambda calculus expression is defined as

e ::= x	variable	(e: egg)
| λx.e	function	(λx: alligator)
| e e	function application	(adjacency of families)

- λx.e is like `(fun x -> e)` in OCaml
- That's it! Only higher-order functions

Three Conveniences

- Syntactic sugar for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1
- The scope of λ extends as far to the right as possible
  - λx. λy.x y is λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

Operational Semantics

- All we've got are functions, so all we can do is call them
- To evaluate (λx.e1) e2
  - Evaluate e1 with x bound to e2
- This application is called "beta-reduction"
  - (λx.e1) e2 → e1[x/e2] (the eating rule)
    - e1[x/e2] is e1 where occurrences of x are replaced by e2
    - Slightly different than the environments we saw for Ocaml
      - Do substitutions to replace formals with actuals, instead of carrying around environment that maps formals to actuals
  - We allow reductions to occur anywhere in a term

Examples (try with alligators too)

- (λx.x) z → z
- (λx.y) z → y
- (λx.x y) z → zy
  - A function that applies its argument to y
- (λx.x y) (λz.z) → (λz.z) y → y
- (λx.λy.x y) z → λy.z y
  - A curried function of two arguments that applies its first argument to its second
- (λx.λy.x y) (λz.zz) x →

  λy.((λz.zz)y)x → (λz.zz)x → xx

Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - $(\lambda x.x\ (\lambda x.x))\ z \rightarrow$?
    - The rightmost "x" refers to the second binding
  - This is a function that takes its argument and applies it to the identity function
- This function is "the same" as $(\lambda x.x\ (\lambda y.y))$
  - Renaming bound variables consistently is allowed
    - This is called *alpha-renaming* or *alpha conversion* (color rule)
  - Ex. $\lambda x.x = \lambda y.y = \lambda z.z$ $\lambda y.\lambda x.y = \lambda z.\lambda x.z$

Static Scoping (cont'd)

- How about the following?
  - $(\lambda x.\lambda y.x\ y)\ y \rightarrow$?
  - When we replace y inside, we don't want it to be "captured" by the inner binding of y
- This function is "the same" as $(\lambda x.\lambda z.x\ z)$