

Logic and predicates

- **Church Booleans**
 - $\text{true} = \lambda x y . x$
 - $\text{false} = \lambda x y . y$
- **Can now define logic operators**
 - $\text{and} = (\lambda p q . p q \text{ false})$
 - Example:
 - $\text{and true false} =$
 - $(\lambda p q . p q \text{ false}) \text{ true false} =$
 - $\text{true false false} =$
 - $(\lambda x y . x) \text{ false false} =$
 - false
 - $\text{or} = \lambda p q . p \text{ true } q$
 - $\text{not} = \lambda p . p \text{ false true}$
 - $\text{ifthenelse} = \lambda p x y . p x y$
 - Example:
 - $\text{ifthenelse true } 3 \ 4 =$
 - $(\lambda p x y . p x y) \text{ true } 3 \ 4 =$
 - $\text{true } 3 \ 4 =$
 - $(\lambda x y . x) 3 \ 4 =$
 - 3
- **Can also define integers (Church numerals), but we'll skip that**

Data structures

- **Define a pair datatype in terms of true and false**
 - $\text{cons} = \lambda f . (\lambda s . (\lambda b . b f s))$ \leftarrow pair
 - $\text{car} = \lambda p . (p \text{ true})$ \leftarrow first
 - $\text{cdr} = \lambda p . (p \text{ false})$ \leftarrow second
 - Example:
 - $\text{cons } 1 \ 2 =$
 - $(\lambda f . (\lambda s . (\lambda b . b f s))) 1 \ 2 =$
 - $(\lambda s . (\lambda b . b 1 s)) 2 =$
 - $(\lambda b . b 1 2)$
 - $\text{car} (\text{cons } 1 \ 2) =$
 - $(\lambda p . (p \text{ true})) (\text{cons } 1 \ 2) =$
 - $(\text{cons } 1 \ 2) \text{ true} =$
 - $(\lambda b . b 1 2) \text{ true} =$
 - $\text{true } 1 \ 2 =$
 - $(\lambda x y . x) 1 \ 2 =$
 - 1

Recursion and the Y combinator

- **Why?**
 - We have if-then-else and data structures, but how can we get loops?

- If we can get recursion, we can use that instead of loops.
- But it appears that λ -calculus doesn't allow functions to call themselves.
- The Y combinator (also called the paradoxical operator or fixed-point operator) is a neat trick that makes recursion possible.
- $Y = \lambda g . (\lambda x . g (x x)) (\lambda x . g (x x))$
 - With parentheses: $Y = \lambda g . ((\lambda x . (g (x x))) (\lambda x . (g (x x))))$
 - Now apply it to g. What does (Y g) expand to?
 - $Y g =$
 - $(\lambda x . (g (x x))) (\lambda x . (g (x x))) =$
 - $g ((\lambda x . (g (x x))) (\lambda x . (g (x x)))) =$
 - $g (Y g) =$
 - $g (g (Y g)) = g (g (g (Y g)))$
 - ... can go on forever, it never stops
- **How to use it?**
 - Example: factorial
 - Define $g = \lambda f n . \{ 1 \text{ if } n = 0; \text{ and } n * f(n - 1) \text{ if } n > 0 \}$
 - Formal notation: $g = (\lambda f . (\lambda n . (\text{ifthenelse } (\text{iszero } n) 1 (* n (f (- n 1))))))$
 - Compute $g (Y g) n$, where n is the number we're calculating the factorial of
 - $g (Y g) 3 =$
 - $(\text{ifthenelse } (\text{iszero } 3) 1 (* 3 ((Y g) (- 3 1)))) =$
 - $(* 3 ((Y g) (- 3 1))) =$
 - $(* 3 ((Y g) 2)) =$
 - $(* 3 ((g (Y g)) 2)) =$
 - $(* 3 (\text{ifthenelse } (\text{iszero } 2) 1 (* 2 ((Y g) (- 2 1)))) =$
 - $(* 3 (* 2 ((Y g) (- 2 1)))) =$
 - $(* 3 (* 2 ((Y g) 1))) =$
 - $(* 3 (* 2 ((g (Y g)) 1))) =$
 - $(* 3 (* 2 (\text{ifthenelse } (\text{iszero } 1) 1 (* 1 ((Y g) (- 1 1)))) =$
 - $(* 3 (* 2 (* 1 ((Y g) (- 1 1)))) =$
 - $(* 3 (* 2 (* 1 ((g (Y g)) 0)))) =$
 - $(* 3 (* 2 (* 1 (\text{ifthenelse } (\text{iszero } 0) 1 (* 0 ((Y g) (- 0 1)))))) =$
 - $(* 3 (* 2 (* 1 1)))$
- **Note: Lazy evaluation**
 - We pass an infinite expression (Y g) as an argument to a function!
 - In Java, the argument would have to be evaluated completely.
 - Why?
 - Because λ -calculus is purely functional, evaluation order doesn't matter, so expressions can be evaluated when needed.

Programming techniques learned from λ -calculus

- **Loop with recursion**
 - Natural way to express many mathematical functions (see slides)
 - Example: fac

- Natural way of operating on data structures (see slides)
 - Example: nnodes
 - **First-order functions**
 - Remember curring: A function can return another function
 - Example: maplist (same as Haskell's built-in map)
 - `maplist add1 [3,7,12]`
 - Compare to the cumbersome Visitor pattern used by Java programmers
 - **Anonymous functions (λ)**
 - `map (\a -> a + 1) [3,7,12]`
 - **Function composition**
 - See Hughes paper
 - **Lazy evaluation**
- Assignment for next time**
- Ghezzi and Jazayeri 3.1-3.5