Lambda Calculus: Datatype encodings

4/9/2021

Agenda

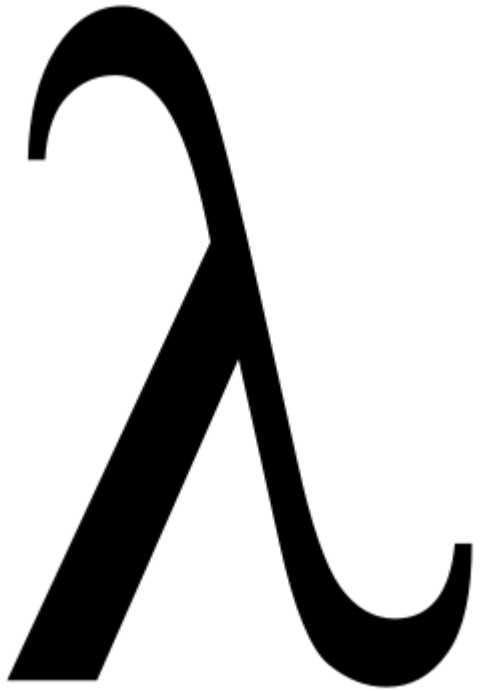
Alpha-renaming

Booleans

Pairs / List

Numbers

Q & A



Alpha-renaming

- Renaming a formal argument
  - `\a -> a  =a>  \b -> b  =a>  \q -> q`

- Only rename the *free* variables

- Rename to make expressions clearer!

Poll

What is NOT a valid alpha renaming of:

```
\f x -> ((\f -> f) x)
```

A. `\g x -> ((\f -> f) x)`

B. `\f y -> ((\f -> f) y)`

C. `\g x -> ((\f -> g) x)`

D. `\f x -> ((\g -> g) x)`

E. No clue `¯\_(ツ)_/¯`

Poll

What is NOT a valid alpha renaming of:

```
\f x -> ((\f -> f) x)
```

A. `\g x -> ((\f -> f) x)`

B. `\f y -> ((\f -> f) y)`

C. `\g x -> ((\f -> g) x)`

D. `\f x -> ((\g -> g) x)`

E. No clue `¯\_(ツ)_/¯`

Working with the Lambda Calculus

- Use alpha / beta reductions to simplify as much as possible

- No more redexes
  - `(\x -> E1) E2`

No more redexes aka "Normal form"

- Use Elsa statements / definitions

=:>

Booleans

```
let TRUE  = \x y -> x        -- Returns its first argument
let FALSE = \x y -> y        -- Returns its second argument
let ITE   = \b x y -> b x y  -- Applies condition to branches
                             -- (redundant, but improves readability)
```

Thinking about booleans

```
let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
```

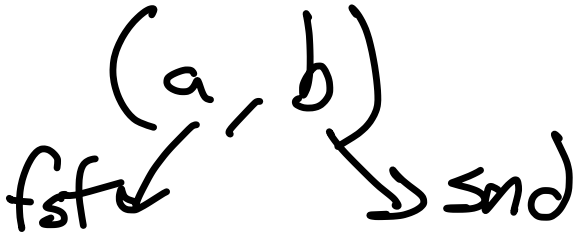
```
let NOT = \b     -> ITE b FALSE TRUE
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR  = \b1 b2 -> ITE b1 TRUE b2
```

```
(ITE (NOT TRUE) (TRUE) (FALSE))
```

- Expand *as necessary*, not all at once!

Stepping through AND (NOT TRUE) TRUE, live!

Pairs

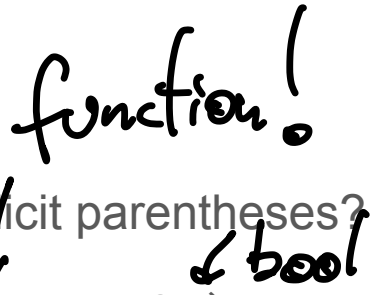


Store structure in a function!

```
let MKPAIR = \x y -> (\b -> ITE b x y)
let FST    = \p -> p TRUE    -- call w/ TRUE, get first value
let SND    = \p -> p FALSE   -- call w/ FALSE, get second value
```

accessor

Poll



Where are the implicit parentheses?

```
SND (MKPAIR ITE FALSE)
```

A. `SND ((MKPAIR ITE) FALSE)`
B. `SND (MKPAIR (ITE FALSE))`

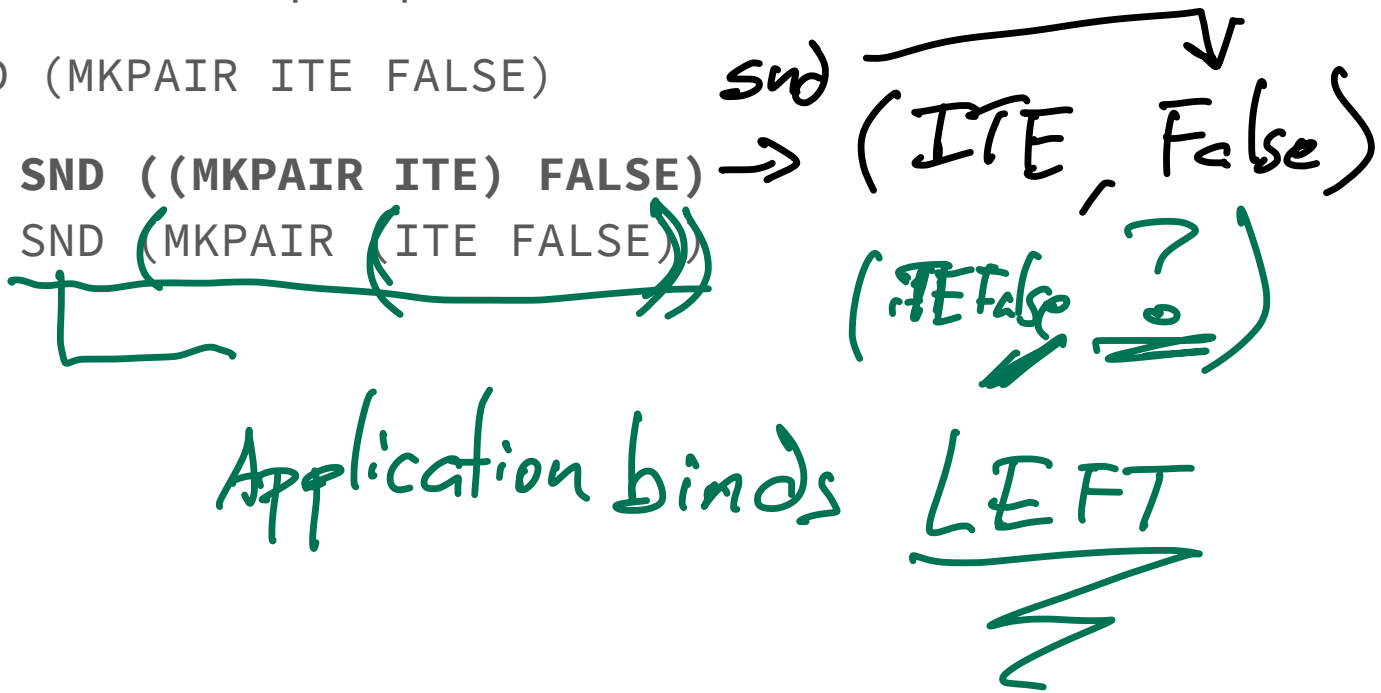
Poll

Where are the implicit parentheses?

```
SND (MKPAIR ITE FALSE)
```

A. SND ((MKPAIR ITE) FALSE)

B. SND (MKPAIR (ITE FALSE))

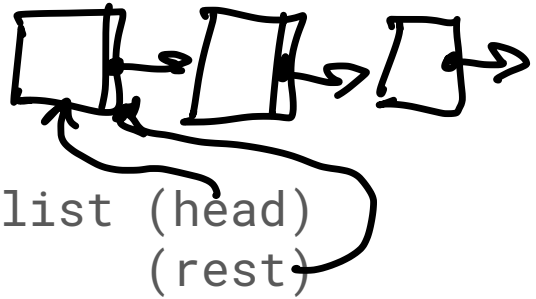


Live

Where are the implicit parentheses?

```
SND (MKPAIR ITE FALSE)
```

Example: List



API:

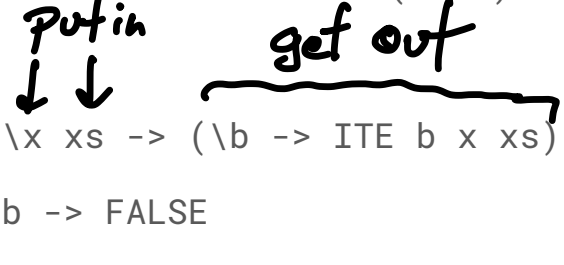
- Get head element of list (head)
- Get rest of list (rest)

```
Let cons = \x xs -> (\b -> ITE b x xs)

Let nil = \b -> FALSE
```

Example: List -- Poll

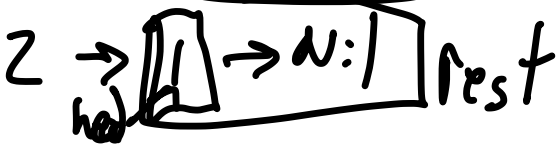
API:

- Get head element of list (head)
- Get rest of list (rest)




```
Let cons = \x xs -> (\b -> ITE b x xs)

Let nil = \b -> FALSE
```

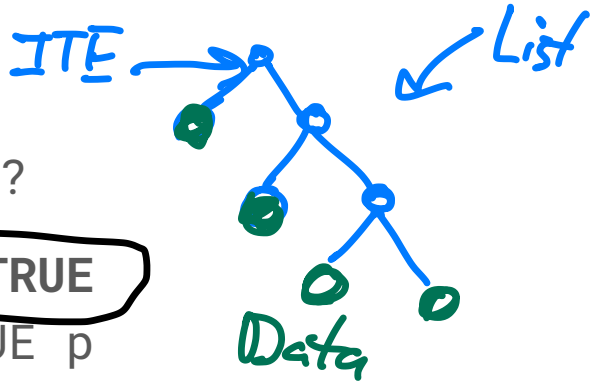
```
HEAD (CONS ONE NIL) == ONE

HEAD (REST (CONS TWO (CONS ONE NIL))) ==
ONE
```



What is HEAD?

A. `\p -> p TRUE`
B. `\p -> TRUE p`
C. `\p -> p FALSE`
D. `\p -> \f x -> f x`
E. ????

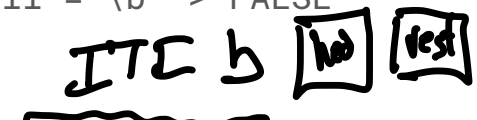
Example: List -- Poll



API:

- Get head element of list (head)
- Get rest of list (rest)


```
Let cons = \x xs -> (\b -> ITE b x xs)

Let nil = \b -> FALSE
```




```
HEAD (CONS ONE NIL) == ONE

HEAD (REST (CONS TWO (CONS ONE NIL))) ==
ONE
```

What is HEAD?

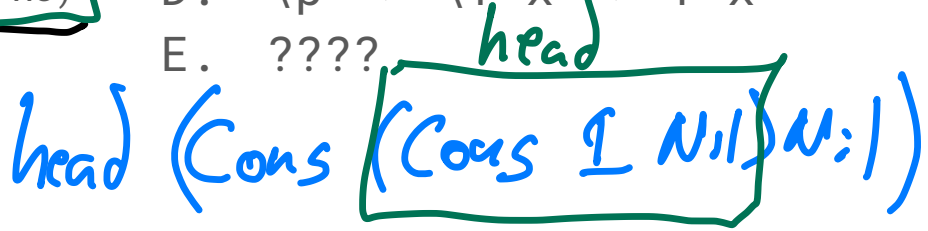
A. \p -> p TRUE

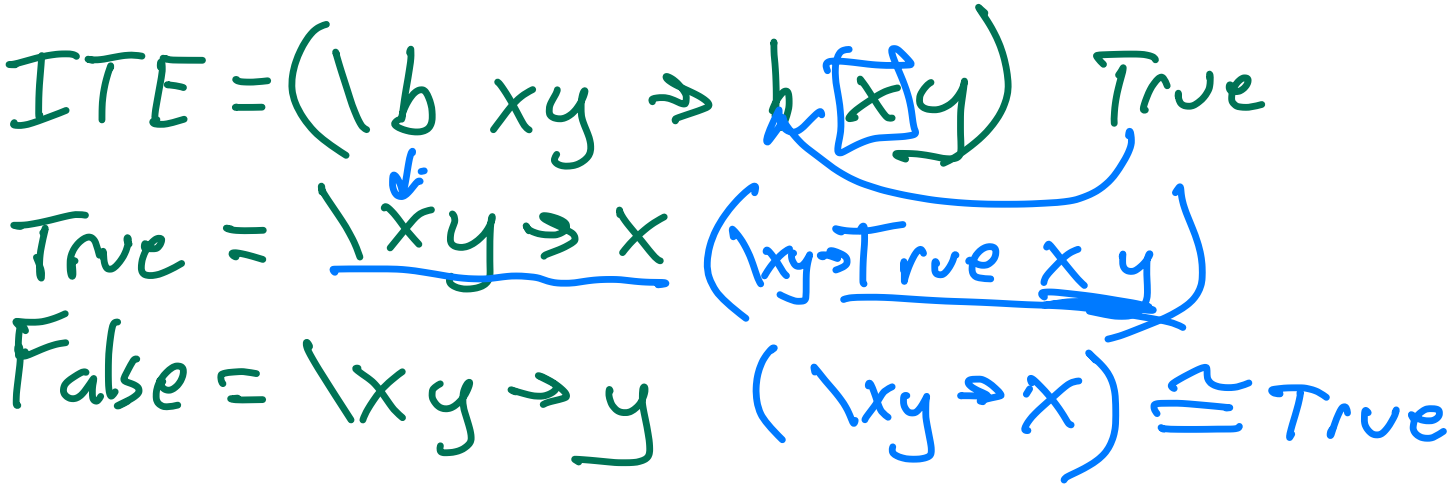
B. \p -> TRUE p

C. \p -> p FALSE

D. \p -> \f x -> f x

E. ????





List Live

Numbers

- Implemented for a purpose: to count or do something X times

- Church Numerals

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x)))))
```

Using Add

```
let INC  = \n f x -> f (n f x)

let ADD  = \n m -> n INC m
```

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
```

ONE represents both:

- The idea of one (1), AND,
- iterating a function 1 time!

Add, Live!

Q & A