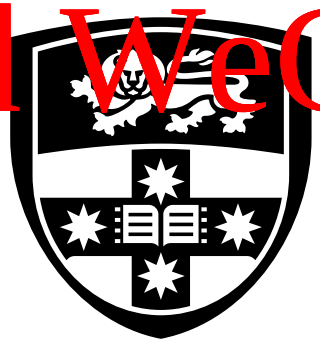
COMP2022: Formal Languages and Logic

2018, Semester 2, Week 3



THE UNIVERSITY OF SYDNEY

OUTLINE

- **Revision - Lambda Calculus**
-
-
  - numbers (a different way)
  - pairs
  - lists

- Functional Programming

When are α-reductions required?

If they never change the meaning, why bother?

-
-
  - ... except when it does!
    - ... even worse – it's not enough to only loo being reduced

WRONG

$$\lambda x.(\lambda yx.y)x$$
$$= \lambda x.(\lambda x.x)$$

$$= FALSE$$

- Where is the error?

WRONG

$$\lambda x.(\lambda yx.y)x$$
$$= \lambda x.(\lambda x.x) \quad \text{(mistake!)}$$

$$= FALSE$$

- Where is the error?
- Why is it a mistake?

WRONG

$$\begin{aligned} & \lambda x.(\lambda yx.y)x \\ = \ & \lambda x.(\lambda x.x) && \text{(mistake!)} \end{aligned}$$

$$= FALSE$$

- Where is the error?
- Why is it a mistake?
- x was bound to the *first* λ, but on line 2 it is not! Free variables in N should not become bound in $M[x := N]$

CORRECT

$$\lambda x.(\lambda yx.y)x$$

(α)

(β)

$$= TRUE$$

Rule of thumb: always perform α redu s.

- sometimes it's necessary
- usually makes the formula easier to read too

η-reduction (Eta)

If x is not free in M, then we can write:

$$\lambda x.Mx = M$$

Idea: a

η-reduction (Eta)

If x is not free in M, then we can write:

$$\lambda x.Mx = M$$

Idea: a

- MN

η-reduction (Eta)

If x is not free in M, then we can write:

$$\lambda x.Mx = M$$

Idea: a MN

- It can simplify some arguments a little
  - e.g. $\lambda x.(\lambda y.y)x = \lambda y.y$
- It can help to convert expressions to 'point free' form (where they do not label their variables).
  - Point-free programs can be easier to reason about, but are often difficult to read.

OUTLINE

- Revision - Lambda Calculus
-
-
  - numbers (a different way)
  - pairs
  - lists

- Functional Programming

Necessary Logical Notation: Quantifiers

- "for all possible values of X ..." denoted $\forall X$
- "there exists a value of X such that ..." denoted $\exists X$

Exa

- "$\forall x\ (x + 1 = 4)$" is false (e.g. false on
- "$\exists x \forall y\ (xy = 0)$" is true (choose
- "$\exists x \forall y\ (xy = 1)$" is false (whatever we c
  able to find a y that doesn't work)
- "$\forall x \exists y\ (xy = 1)$" is true (for any x, we can choose $y = \frac{1}{x}$)

Combinators

A *combinator* is any expression M which contains no free variables.

Combinators

A *combinator* is any expression M which contains no free variables.

Exa

-

Combinators

A *combinator* is any expression M which contains no free variables.

Exa

-
- $\lambda xy.xyy$ is a combinator (all variables bo

Combinators

A *combinator* is any expression M which contains no free variables.

Exa

-
- $\lambda xy.xyy$ is a combinator (all variables bo

Combinators combine values into expressions
quantifiers or explicitly defining variables.

Combinator examples

Standard combinators:

- $I = \lambda x.x$ (identity)
- $K = \lambda xy.x$ (true)
-
-

Combinator examples

Standard combinators:

- $I = \lambda x.x$ (identity)
- $K = \lambda xy.x$ (true)
-
-

We ca

- $IM = M$
- $KMN = M$
- $K_*MN = N$
- $SMNL = ML(NL)$

Combinator examples

Standard combinators:

- $I = \lambda x.x$ (identity)
- $K = \lambda xy.x$ (true)
-
-

We ca

- $IM = M$
- $KMN = M$
- $K_*MN = N$
- $SMNL = ML(NL)$

Interestingly, these λ-free combinators are sufficient to make expressions equal to any λ term. We will not talk about that further today though.

Solving simple equations

$$\exists G. \forall X. \; G\,X = X\,X\,X$$

(Where X, F are expressions in the lambda calculus)

Solving simple equations

$\exists G \forall X \; GX = XXX$

(Where X, F are expressions in the lambda calculus)

"The GC

Proof:

Solving simple equations

$\exists G \forall X \; GX = XXX$

(Where X, F are expressions in the lambda calculus)

"The GC

Proof:

- Let $G = \lambda x.xxx$

Solving simple equations

$$\exists G \forall X \; GX = XXX$$

(Where X, F are expressions in the lambda calculus)

"The GC

Proof:

- Let $G = \lambda x.xxx$
- Then $GX = (\lambda x.xxx)X = XXX$

Solving simple equations

$$\exists G \forall X \; GX = XXX$$

(Where X, F are expressions in the lambda calculus)

"The GC

Proof:

- Let $G = \lambda x.xxx$
- Then $GX = (\lambda x.xxx)X = XXX$

That was easy... But what if we need to reason about a recursive function?

Fixed Point Combinators

A *fixed point combinator* is a combinator which has a fixed point.

We sa

i.e. some input X exists which, when appli
again.

Fixed point theorem (i)

Theorem:

$$\forall F\ \exists X\ FX = X$$

Fixed point theorem (i)

Theorem:

$$\forall F\ \exists X\ FX = X$$

"For all F, there exists some X such that $FX = X$"

-

Fixed point theorem (i)

Theorem:

$$\forall F\ \exists X\ FX = X$$

"For all F, there exists some X such that $FX = X$"

- ▶

Proo

Let

$$X =$$

Fixed point theorem (i)

Theorem:

$$\forall F \exists X.\ FX = X$$

"For all F, there exists some X such that $FX = X$"

- ▶

Proo
Let

$$X = WW$$

Fixed point theorem (i)

Theorem:

$$\forall F \exists X.\ FX = X$$

"For all F, there exists some X such that $FX = X$"

- ▶

Proo
Let

$$\begin{aligned} X &= WW \\ &= (\lambda x.F(xx))W \\ &= \end{aligned}$$

Fixed point theorem (i)

Theorem:

$$\forall F \exists X.\ FX = X$$

"For all F, there exists some X such that $FX = X$"

- ▶

Proo

Let

$$\begin{aligned} X &= WW \\ &= (\lambda x.F(xx))W \\ &= F(WW) \qquad \beta \qquad \text{ion)} \\ &= \end{aligned}$$

Fixed point theorem (i)

Theorem:

$$\forall F \exists X.\ FX = X$$

"For all F, there exists some X such that $FX = X$"

- ▶

Proo
Let

$$\begin{aligned} X &= WW \\ &= (\lambda x.F(xx))W \\ &= F(WW) && \beta \text{ ion)} \\ &= FX && \text{(def. of X)} \end{aligned}$$

Fixed point theorem (II)

There is a fixed point combinator (the "Y Combinator")

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

Proo

$$YF =$$

Fixed point theorem (II)

There is a fixed point combinator (the "Y Combinator")

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

Proo

$$YF = \big(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad Y\big)$$

=

Fixed point theorem (II)

There is a fixed point combinator (the "Y Combinator")

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

Proo

$$\begin{aligned} YF &= \big(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)\big) \qquad Y) \\ &= (\lambda x.F(xx))(\lambda x.F(xx)) \qquad \text{(on)} \\ &= \end{aligned}$$

Fixed point theorem (II)

There is a fixed point combinator (the "Y Combinator")

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

Proo

$$\begin{aligned} YF &= \Big(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)\Big) \qquad Y) \\ &= (\lambda x.F(xx))(\lambda x.F(xx)) \qquad \text{tion)} \\ &= F\Big((\lambda x.F(xx))(\lambda x.F(xx)\Big) \\ &= \end{aligned}$$

Fixed point theorem (II)

There is a fixed point combinator (the "Y Combinator")

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

Proo

$$\begin{aligned} YF &= \Big(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)\Big) && Y) \\ &= (\lambda x.F(xx))(\lambda x.F(xx)) && \text{con}) \\ &= F\Big((\lambda x.F(xx))(\lambda x.F(xx)\Big) \\ &= F(YF) && \text{(by equality above)} \end{aligned}$$

Y Combinator

So, uh... Why is this interesting?

Y Combinator

So, uh... Why is this interesting?

1.

Y Combinator

So, uh... Why is this interesting?

1.

2. Bad news: it leads to Curry's Paradox, and th
incompleteness of lambda calculus
   - not all valid expressions can be proved / co

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

We'

-

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

We'

-
  - This notation, indicating n repe
  - dangerous (but convenient)
  - Be aware that Church numerals have the f

$$\lambda fz.f(f(f(f(fz)))) \neq \lambda fz.fffffz$$

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n \quad f(n \quad 1)$$

We'l

- ISZERO $:= \lambda n.n\ (\lambda x.FALSE)\ T$

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n \quad f(n \quad 1)$$

We'l

- ISZERO $:= \lambda n.n\ (\lambda x.FALSE)\ T$
  - Returns TRUE if the argument is a Churc any other Church numeral

ISZERO ZERO

ISZERO ZERO

ISZERO ZERO

ISZERO ZERO

$$ISZERO\ ZERO$$

$$= (\lambda fz.z)\ (\lambda x.FALSE)\ TRUE$$

$$=$$

ISZERO ZERO

$$ISZERO\ ZERO$$

$$= (\lambda fz.z)\ (\lambda x.FALSE)\ TRUE \qquad (\beta)$$

$$= (\lambda z.z)\ TRUE \qquad (\beta)$$

ISZERO ZERO

$$ISZERO\ ZERO$$

$$= (\lambda fz.z)\ (\lambda x.FALSE)\ TRUE \qquad (\beta)$$

$$= (\lambda z.z)\ TRUE \qquad (\beta)$$

$$= TRUE \qquad (\beta)$$

ISZERO ONE

ISZERO ONE

ISZERO ONE

ISZERO ONE

$ISZERO\ ONE$

ERO)

(β)

$f.ONE)$

$=$

$=$

$=$

ISZERO ONE

$$ISZERO\ ONE$$

ERO)

(β)

$f.ONE)$

$$= (\lambda z.(\lambda x.FALSE)z)\ TRUE \qquad (\beta)$$

$$=$$

$$=$$

ISZERO ONE

$$ISZERO\ ONE$$

ERO)

$$= (\lambda z.(\lambda x.FALSE)z)\ TRUE \qquad (\beta)$$

$f.ONE)$

$$= (\lambda x.FALSE)TRUE \qquad (\beta)$$

$$= \qquad (\beta)$$

ISZERO ONE

$$ISZERO\ ONE$$

ERO)

$$= (\lambda z.(\lambda x.FALSE)z)\ TRUE \quad (\beta)$$

$f.ONE)$

$$= (\lambda z.(\lambda x.FALSE)z)\ TRUE \quad (\beta)$$

$$= (\lambda x.FALSE)TRUE \quad (\beta)$$

$$= FALSE \quad (\beta)$$

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

We'l

-
-
- MULT := $\lambda xyz.x(yz)$ (seen previo

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

We'l

-
-
- MULT := $\lambda xyz.x(yz)$ (seen previo
- PRED := $\lambda nfx.n(\lambda gh.h(gf))(\lambda y.$
  - This gives the predecessor of a number
  - PRED 1 = 0, PRED 2 = 1, ..., PRED n = (n-1)
  - The derivation of this is *much* longer than for the operations which increase numbers

Recursion example

Suppose we want to compute factorials:

$$f(n) = if\ (n == 0)\ then\ 1\ else\ n * f(n-1)$$

We'l

-
-
- MULT := $\lambda xyz.x(yz)$ (seen previo
- PRED := $\lambda nfx.n(\lambda gh.h(gf))(\lambda y.$
  - This gives the predecessor of a number
  - PRED 1 = 0, PRED 2 = 1, ..., PRED n = (n-1)
  - The derivation of this is *much* longer than for the operations which increase numbers
    - Subtraction and division are also difficult!

PRED TWO... is a monster

$$PRED\ TWO = \lambda nfx.n(\lambda gh.h(gf))(\lambda y.x)(\lambda u.u)\ TWO$$
$$= \lambda fx.TWO\ (\lambda gh.h(gf))(\lambda y.x)(\lambda u.u)$$
$$= \lambda fx.(\lambda gh.h(gf))\Big((\lambda$$
$$= \lambda fx.\big(\lambda gh.h(gf)\big)(\lambda h.$$
$$= \lambda fx.\big(\lambda gh.h(gf)\big)(\lambda i.ix)(\lambda u.u)$$
$$= \lambda fx.\big(\lambda h.h((\lambda i.ix)f)\big)(\lambda u.u)$$
$$= \lambda fx.(\lambda h.h(fx))(\lambda u.u)$$
$$= \lambda fx.(\lambda u.u)(fx)) = \lambda fx.fx = ONE$$

$$)$$
$$x)(\lambda u.u)$$
$$.u)$$

Recursion example

Suppose we want to compute factorials:

We'l

- $c_n = \lambda fx.f^n(x)$
- ISZERO $:= \lambda n.n\ (\lambda x.\ FALSE)\ T$
- MULT $:= \lambda xyz.x(yz)$
- PRED $:= \lambda nfx.n(\lambda gh.h(gf))(\lambda y.x)(\lambda u.u)$

Recursion example

We want to write something like:

"FACT n = (ISZERO n) 1 (MULT n (FACT (PRED n)))"

We ca

Com

Recursion example

We want to write something like:

"FACT n = (ISZERO n) 1 (MULT n (FACT (PRED n)))"

We ca

Com

- $H = \lambda fn.(ISZERO\ n)\ 1 \quad MULT\ n\ (f\ (PRED\ n))$
  - H takes a function and a number. If the nu
    returns 1, otherwise it returns the produc

Recursion example

We want to write something like:

"FACT n = (ISZERO n) 1 (MULT n (FACT (PRED n)))"

We ca

Com

- $H = \lambda fn.(ISZERO\ n)\ 1 \quad MULT\ n\ (f\ (PRED\ n))$
  - H takes a function and a number. If the nu
    returns 1, otherwise it returns the produc
- $FACTORIAL = Y\ H$
  - Because $YH = H(YH)$, the Y Combinator helps us to apply the H function to itself

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

$$= H\ (Y\ H)\ 5$$ r!)

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

$$= H\ (Y\ H)\ 5$$ r!)

$$= 5 \times ((Y\ H)$$ <0)

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

$$= H\ (Y\ H)\ 5 \qquad \text{r!)}$$

$$= 5 * ((Y\ H) \qquad < 0)$$

$$= \ldots$$

$$= 120 * ((Y\ H)\ 0)$$

Factorial 5(overview)

- $H = \lambda fn.(ISZERO\ n)\ 1\ \left(MULT\ n\ (f\ (PRED\ n))\right)$
- $FACTORIAL = Y\ H$

H takes a function f and a number n. It returns 1 if the number is 0, oth

$$= H\ (Y\ H)\ 5 \qquad \text{r!)}$$

$$= 5 * ((Y\ H) \qquad < 0)$$

$$= \ldots$$

$$= 120 * ((Y\ H)\ 0)$$

$$= 120 * 1 = 120$$

Factorial 3 (detailed 1)

FACTORIAL 3

=

=

=

=

=

=

=

Factorial 3 (detailed 1)

$$
\begin{aligned}
& \mathit{FACTORIAL}\ 3 \\
= & \ Y\ H\ 3 \\
= & \\
= & \\
= & \\
= & \\
= & \\
= &
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= \mathit{FACTORIAL}\ 3 \\
&= Y\ H\ 3 \\
&= \\
&= \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= \mathit{FACTORIAL}\ 3 \\
&= Y\ H\ 3 \\
&= \\
&= \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= \mathit{FACTORIAL}\ 3 \\
&= Y\ H\ 3 \\
&= \qquad\qquad \text{inator)} \\
&= \qquad\qquad 3 \quad \text{(H)} \\
&= \qquad\qquad 3 \quad (\beta) \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= FACTORIAL\ 3 \\
&= Y\ H\ 3 \\
&= \qquad\qquad\qquad\qquad\qquad\qquad \text{inator)} \\
&= \qquad\qquad\qquad\qquad\qquad\qquad 3 \quad \text{(H)} \\
&= \qquad\qquad\qquad\qquad\qquad\qquad 3 \quad (\beta) \\
&= (ISZERO\ 3)\ 1\ (MULT\ 3\ (Y\ H \qquad\qquad (\beta) \\
&= \\
&= \\
&=
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
& \mathit{FACTORIAL}\ 3 \\
= {} & Y\ H\ 3 \\
= {} & \ldots \text{inator)} \\
= {} & \ldots 3 \quad \text{(H)} \\
= {} & \ldots 3 \quad (\beta) \\
= {} & (\mathit{ISZERO}\ 3)\ 1\ \big(\mathit{MULT}\ 3\ (Y\ H \ldots \quad (\beta) \\
= {} & \ldots = \mathit{FALSE}\ 1\ \big(\mathit{MULT}\ 3\ (Y\ H \ldots \\
= {} & \\
= {} &
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= FACTORIAL\ 3 \\
&= Y\ H\ 3 && \text{(def. } Y\text{ combinator)} \\
&= \ldots\ 3 && \text{(H)} \\
&= \ldots\ 3 && (\beta) \\
&= (ISZERO\ 3)\ 1\ (MULT\ 3\ (Y\ H\ \ldots)) && (\beta) \\
&= \ldots = FALSE\ 1\ \left(MULT\ 3\ \left(Y\ H\ \ldots\right)\right) && / \ 0) \\
&= \ldots = MULT\ 3\ \left(Y\ H\ (PRED\ 3)\right) && \text{(def. } FALSE\text{)} \\
&=
\end{aligned}
$$

Factorial 3 (detailed 1)

$$
\begin{aligned}
&= FACTORIAL\ 3 && \\
&= Y\ H\ 3 && \\
&= && \text{inator)} \\
&= && \text{)3 (H)} \\
&= && 3 \quad (\beta) \\
&= (ISZERO\ 3)\ 1\ \left(MULT\ 3\ (Y\ H\right. && (\beta) \\
&= \ldots = FALSE\ 1\ \left(MULT\ 3\ \left(Y\ H\right.\right. && /\ 0) \\
&= \ldots = MULT\ 3\ \left(Y\ H\ (PRED\ 3)\right) && (\text{def. } FALSE) \\
&= \ldots = MULT\ 3\ (Y\ H\ 2) && (PRED\ 3 = 2)
\end{aligned}
$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= \ldots = MULT\ 3\ (Y\ H\ 2)$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= \ldots = MULT\ 3\ (Y\ H\ 2)$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= \ldots = MULT\ 3\ (Y\ H\ 2)$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= \ldots = MULT\ 3\ (Y\ H\ 2)$$

$$= \ldots = \ldots(ISZERO\ 0)\ 1\ldots$$

$$=$$

$$=$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= \ldots = MULT\ 3\ (Y\ H\ 2)$$

$$= \ldots = \ldots(ISZERO\ 0)\ 1\ldots$$

$$= \ldots = MULT\ 3\ (MULT\ 2\ ($$

$$=$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$= ... = MULT\ 3\ (Y\ H\ 2)$$

$$= ... = ...(ISZERO\ 0)\ 1...$$

$$= ... = MULT\ 3\ (MULT\ 2\ ($$

$$= ... = MULT\ 3\ (MULT\ 2\ 1)$$

$$=$$

$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$
$$= ... = MULT\ 3\ (Y\ H\ 2)$$

$$= ... = ...(ISZERO\ 0)\ 1...$$
$$= ... = MULT\ 3\ (MULT\ 2\ ($$
$$= ... = MULT\ 3\ (MULT\ 2\ 1)$$

$$= ... = MULT\ 3\ 2$$
$$=$$

Factorial 3 (detailed 2)

$$FACTORIAL\ 3$$

$$= ... = MULT\ 3\ (Y\ H\ 2)$$

$$= ... = ...(ISZERO\ 0)\ 1...$$

$$= ... = MULT\ 3\ (MULT\ 2\ ($$

$$= ... = MULT\ 3\ (MULT\ 2\ 1)$$

$$= ... = MULT\ 3\ 2$$

$$= ... = 6$$

Y Combinator reminder

This worked because the Y Combinator

Has th

Important:

- When performing the reductions, use that
- *Don't* β-reduce the Y Combinator directly.

OUTLINE

- Revision - Lambda Calculus
-
-
  - numbers (a different way)
  - pairs
  - lists

- Functional Programming

Booleans

Reminder: our definition of Church Booleans lets us write:

simp

where B is some boolean. i.e. anything which re

- $TRUE = \lambda xy.x$, or
- $FALSE = \lambda xy.y$

Pairs (Barendregt style)

Let P, Q be expressions in the lambda calculus.

If we write:

Then:

- $[M, N]\ TRUE = M$
- $[M, N]\ FALSE = N$

We can use $[M, N]$ to denote an ordered pair.

Pairs (Barendregt)

$$[M, N]\ TRUE = (\lambda z.z\ M\ N)\ TRUE$$
$$= TRUE\ M\ N$$

Pairs (Barendregt)

$$[M, N]\ TRUE = (\lambda z.z\ M\ N)\ TRUE$$
$$= TRUE\ M\ N$$

$$[M, N]\ FALSE = (\lambda z.z\ M\ N)$$
$$= FALS$$
$$= (\lambda xy.y)\ M\ N$$
$$= (\lambda y.y)\ N$$
$$= N$$

Numbers (Barendregt style)

Recall that predecessor, subtraction, division were difficult in the Church encoding.

Numbers (Barendregt style)

Recall that predecessor, subtraction, division were difficult in the Church encoding.

We ca

- $n + 1 = [FALSE, n]$

Numbers (Barendregt style)

Recall that predecessor, subtraction, division were difficult in the Church encoding.

We ca

- $n + 1 = [FALSE, n]$

For example:

- $1 = [FALSE, 0] = [FALSE, I] =$

Numbers (Barendregt style)

Recall that predecessor, subtraction, division were difficult in the Church encoding.

We ca

- $n + 1 = [FALSE, n]$

For example:

- $1 = [FALSE, 0] = [FALSE, I] =$
- $2 = [FALSE, 1] = [FALSE, [FALSE, I]]$

Numbers (Barendregt style)

Recall that predecessor, subtraction, division were difficult in the Church encoding.

We ca

- $n + 1 = [FALSE, n]$

For example:

- $1 = [FALSE, 0] = [FALSE, I] =$
- $2 = [FALSE, 1] = [FALSE, [FALSE, I]]$
- $3 = [FALSE, 2] = [FALSE, [FALSE, [FALSE, I]]]$

Numbers (Barendregt style)

Some of the operators are a *lot* simpler:

- $SUCC = \lambda x.[FALSE, x]$ (the next number)
  - This simply puts another FALSE in front.
  - $SUCC\ \ ONE = (\lambda x.[FALSE, x])ONE = [FALSE, ONE] =$

-

$$[FALSE, I]FALSE = I$$

- $ISZERO = \lambda x.x\ TRUE$

Numbers (Barendregt style)

Some of the operators are a *lot* simpler:

- $SUCC = \lambda x.[FALSE, x]$ (the next number)
  - This simply puts another FALSE in front.
  - $SUCC\ \ ONE = (\lambda x.[FALSE, x])ONE = [FALSE, ONE] =$

$$[FALSE, I]FALSE = I$$

- $ISZERO = \lambda x.x\ TRUE$

... recall that PRED for the Church numerals was

$$\lambda nfx.n(\lambda gh.h(gf))(\lambda y.x)(\lambda u.u)$$

Numbers (Barendregt style)

- Addition is more complex but quite intuitive
  - base case: $ADD(0, y) = y$
  -

Numbers (Barendregt style)

- Addition is more complex but quite intuitive
  - base case: $ADD(0, y) = y$
  -

To im

$$ADD = Y\left(\lambda f x y.(ISZERO\ x)\ y\ (\quad)\right)$$

  - i.e. Y "if x is 0 then y else $(1 + f(x - 1,\ y))$"

Generalised recursion

We can generalise this idea of recursion to support an arbitrary num

See section 3.11 in the reference text (Barendreg interested in the fine details of this.

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
&= \\
&= \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
&= \\
&= \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
&= \\
&= \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
 & = (\lambda xyz. \\
 & = \\
 & = \\
 & = \\
 & = \\
 & =
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
 & = (\lambda xyz. \\
 & = (\lambda yz.z \\
 & = \\
 & = \\
 & = \\
 & = \\
 & =
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
 & (\\
= & (\lambda xyz.\\
= & (\lambda yz.z\\
= & (\lambda z.za\\
= & \\
= & \\
= & \\
= &
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
&= (\lambda xyz. \\
&= (\lambda yz.z \\
&= (\lambda z.za \\
&= TRUE\ a\ b \\
&= \\
&= \\
&=
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
 & = (\lambda xyz. \\
 & = (\lambda yz.z \\
 & = (\lambda z.za \\
 & = TRUE\ a\ b \\
 & = (\lambda xy.x)ab \\
 & = \\
 & =
\end{aligned}
$$

Pairs (Church)

Similar idea, but not identical to the encoding Barendregt uses.

- $PAIR = \lambda xyz.zxy$
- $FIRST = \lambda p.p\ TRUE$
- $SECOND = \lambda p.p\ FALSE$

e.g.

$$
\begin{aligned}
 & (\\
=\ & (\lambda xyz.\\
=\ & (\lambda yz.z\\
=\ & (\lambda z.za\\
=\ & TRUE\ a\ b\\
=\ & (\lambda xy.x)ab\\
=\ & (\lambda y.a)b\\
=\ & a
\end{aligned}
$$

List (Church)

Idea: lists are pairs of (head, tail)

-
-

I will denote lists as $\{a, b, c, d, ...\}$

List (Church)

We need a way to signal if the list is empty.

List (Church)

We need a way to signal if the list is empty.

Idea: each list entry is a nested pair (isempty, (head, tail))

List (Church)

We need a way to signal if the list is empty.

Idea: each list entry is a nested pair (isempty, (head, tail))

- Empty list $= NIL = PAIR\ TRUE\ TRUE$
-
-

List (Church)

We need a way to signal if the list is empty.

Idea: each list entry is a nested pair (isempty, (head, tail))

- Empty list $= NIL = PAIR\ TRUE\ TRUE$
-
-

A list containing $\{a, b, c, d\}$ would look li

$$(PAIR\ FALSE\ (PAIR\ a$$
$$(PAIR\ FALSE\ (PAIR\ b$$
$$(PAIR\ FALSE\ (PAIR\ c$$
$$(PAIR\ FALSE\ (PAIR\ d\ NIL)))))))$$

List (Church)

To make our lists useful, we want the following functions:

-
-
-
- *TAIL* gets the rest
- *CONS* prepends a given value to the head o

List (Church)

Encoding:

-
-
-
- $TAIL =$
- $CONS =$

List (Church)

Encoding:

-
-
-
- $TAIL =$
- $CONS =$

List (Church)

Encoding:

-
-
-
- $TAIL =$
- $CONS =$

List (Church)

Encoding:

-
-
-
- $TAIL =$
- $CONS =$

List (Church)

Encoding:

-
-
-
- $TAIL = \lambda z.SECOND\ (SECON$
- $CONS =$

List (Church)

Encoding:

-
-
-
- $TAIL = \lambda z.SECOND\ (SECON$
- $CONS = \lambda ht.PAIR\ FALSE\ (P$

List (Church)

Example:

$$\mathit{ISNIL}\ \mathit{NIL}$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$\mathit{ISNIL}\ \mathit{NIL}$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$\mathit{ISNIL}\ \mathit{NIL}$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$\mathit{ISNIL}\ \mathit{NIL}$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ NIL$$

$$= PAIR\ TRUE\ TRUE\ TR$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ NIL$$

$$= PAIR\ TRUE\ TRUE\ TR$$

$$= (\lambda xyz.zxy)\ TRUE\ T$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ NIL$$

$$= PAIR\ TRUE\ TRUE\ TR$$

$$= (\lambda xyz.zxy)\ TRUE\ T$$

$$= \ldots = TRUE\ TRUE\ TR$$

$$=$$

List (Church)

Example:

$$ISNIL\ NIL$$

$$= PAIR\ TRUE\ TRUE\ TR$$

$$= (\lambda xyz.zxy)\ TRUE\ T$$

$$= ... = TRUE\ TRUE\ TR$$

$$= ... = TRUE$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$\mathit{ISNIL}\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$= PAIR\ FALSE\ (PAIR\ a\ \{$$

$$=$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$= PAIR\ FALSE\ (PAIR\ a\ \{$$

$$= (\lambda xyz.zxy)\ FALSE\ (PAI$$

$$=$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$= PAIR\ FALSE\ (PAIR\ a\ \{$$

$$= (\lambda xyz.zxy)\ FALSE\ (PAI$$

$$= \ldots = TRUE\ FALSE\ (PAI$$

$$=$$

List (Church)

Example:

$$ISNIL\ \{a, b, c, d\}$$

$$= PAIR\ FALSE\ (PAIR\ a\ \{$$

$$= (\lambda xyz.zxy)\ FALSE\ (PAI$$

$$= ... = TRUE\ FALSE\ (PAI$$

$$= ... = FALSE$$

List (Church) example

$$\mathit{HEAD}\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example

$$\mathit{HEAD}\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example

$$\mathit{HEAD}\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example

$$\mathit{HEAD}\ \{a, b, c, d\}$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example

$$HEAD\ \{a, b, c, d\}$$

$$= SECOND\ \{a, b, c, d\}\ TRU$$

$$=$$

$$=$$

$$=$$

List (Church) example

$$HEAD\ \{a, b, c, d\}$$

$$= SECOND\ \{a, b, c, d\}\ TRU$$

$$= (\lambda p.\ p\ FALSE)\ \{a, b, c, d\}$$

$$=$$

$$=$$

List (Church) example

$$HEAD\ \{a, b, c, d\}$$

$$= SECOND\ \{a, b, c, d\}\ TRU$$

$$= (\lambda p\ q.\ FALSE)\ \{a, b, c, d\}$$

$$= \{a, b, c, d\}\ FALSE\ TRUE$$

$$=$$

List (Church) example

$$HEAD\ \{a, b, c, d\}$$

$$= SECOND\ \{a, b, c, d\}\ TRU$$

$$= (\lambda p\ q.\ FALSE)\ \{a, b, c, d\}$$

$$= \{a, b, c, d\}\ FALSE\ TRUE$$

$$= PAIR\ FALSE\ (PAIR\ a\ \{b, c, d\})\ FALSE\ TRUE$$

$$= \ldots$$

List (Church) example (continued)

$$
\begin{aligned}
& HEAD\{a, b, c, d\} \\
&= \ldots \\
&= \\
&= \\
&= \\
&= \\
&=
\end{aligned}
$$

List (Church) example (continued)

$$HEAD\{a, b, c, d\}$$

$$= \ldots$$

$$E$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example (continued)

$$HEAD\{a, b, c, d\}$$

$$= \ldots$$

$$= E$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example (continued)

$$HEAD\{a, b, c, d\}$$

$$= \ldots$$

$$= E$$

$$=$$

$$=$$

$$=$$

$$=$$

$$=$$

List (Church) example (continued)

$$HEAD\ \{a, b, c, d\}$$

$$= \ldots$$

$$E$$

$$= FALSE\ FALSE\ (PAIR\ a\ \{b,$$

$$=$$

$$=$$

$$=$$

List (Church) example (continued)

$$
\begin{aligned}
& HEAD\ \{a, b, c, d\} \\
&= \ldots \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E \\
&= FALSE\ FALSE\ (PAIR\ a\ \{b, \\
&= \ldots = PAIR\ a\ \{b, c, d\}\ TRUE \\
&= \\
&= \\
&=
\end{aligned}
$$

List (Church) example (continued)

$$HEAD\ \{a, b, c, d\}$$

$$= \ldots$$

$$E$$

$$E$$

$$= FALSE\ FALSE\ (PAIR\ a\ \{b,$$

$$= \ldots = PAIR\ a\ \{b, c, d\}\ TRUE$$

$$= (\lambda xyz.zxy)\ a\ \{b, c, d\}\ TRUE$$

$$=$$

$$=$$

List (Church) example (continued)

$HEAD\ \{a, b, c, d\}$

$= \ldots$

E

E

$= FALSE\ FALSE\ (PAIR\ a\ \{b,$

$= \ldots = PAIR\ a\ \{b, c, d\}\ TRUE$

$= (\lambda xyz.zxy)\ a\ \{b, c, d\}\ TRUE$

$= \ldots = TRUE\ a\ \{b, c, d\})$ (3 β-reductions)

$=$

List (Church) example (continued)

$= HEAD \{a, b, c, d\}$

$= \ldots$

E

E

$= FALSE\ FALSE\ (PAIR\ a\ \{b,$

$= \ldots = PAIR\ a\ \{b, c, d\}\ TRUE$

$= (\lambda xyz.zxy)\ a\ \{b, c, d\}\ TRUE$

$= \ldots = TRUE\ a\ \{b, c, d\})$ (3 β-reductions)

$= \ldots = a$ (TRUE a b = a)

List (Church) cons

$$CONS = \lambda ht.PAIR\ FALSE\ (PAIR\ h\ t)$$

$$=$$

$$=$$

List (Church) cons

$$CONS = \lambda ht.PAIR\ FALSE\ (PAIR\ h\ t)$$

$$=$$

$$=$$

$$=$$

List (Church) cons

$$CONS = \lambda ht.PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= (\lambda t.PAIR\ FALSE\ (PA$$

$$=$$

$$=$$

List (Church) cons

$$CONS = \lambda ht.PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= (\lambda t.PAIR\ FALSE\ (PA$$

$$= PAIR\ FALSE\ (PAIR\ c\ N$$

$$=$$

List (Church) cons

$$CONS = \lambda ht.PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= (\lambda t.PAIR\ FALSE\ (PA$$

$$= PAIR\ FALSE\ (PAIR\ a\ N$$

$$= \{a\}$$

List (Church) cons

$$CONS \equiv \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$=$

$=$

$=$

List (Church) cons

$$CONS \equiv \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$=$

$=$

$=$

List (Church) cons

$$CONS \equiv \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$=$

$=$

$=$

List (Church) cons

$$CONS = \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= PAIR\ FALSE\ (PAIR\ b\ (CONS$$

$$=$$

$$=$$

List (Church) cons

$$CONS = \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= PAIR\ FALSE\ (PAIR\ b\ (CONS$$

$$= PAIR\ FALSE\ (PAIR\ b\ (CONS$$

$$=$$

$$=$$

List (Church) cons

$$CONS = \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$$= PAIR\ FALSE\ (PAIR\ b\ (CONS$$

$$= PAIR\ FALSE\ (PAIR\ b\ (CONS$$

$$= PAIR\ FALSE\ (PAIR\ b\ (PAIR \qquad))$$

$$=$$

List (Church) cons

$$CONS = \lambda h\, t.\, PAIR\ FALSE\ (PAIR\ h\ t)$$

$$\begin{aligned}
&= PAIR\ FALSE\ (PAIR\ b\ (CONS \\
&= PAIR\ FALSE\ (PAIR\ b\ (CONS \\
&= PAIR\ FALSE\ (PAIR\ b\ (PAIR \qquad\qquad)) \\
&= \{b, a\}
\end{aligned}$$

List encodings

We now have structured data *and* recursion!

List encodings

We now have structured data *and* recursion!

Don't forget, just as there are many ways to represe
in imperative programming, there are many pos
lists and other structures in lambda calculus.

OUTLINE

- Revision - Lambda Calculus
-
-
  - numbers (a different way)
  - pairs
  - lists

- **Functional Programming**

Fibonacci

In the last tutorial, you probably implemented Fibonacci like this:

```
(defun fib (x)
  (if (< x 2)
```

Fibonacci

In the last tutorial, you probably implemented Fibonacci like this:

```
(defun fib (x)
  (if (< x 2)
```


This works, but is not very efficient (exponential ti

Fibonacci

In the last tutorial, you probably implemented Fibonacci like this:

```
(defun fib (x)
  (if (< x 2)
```


This works, but is not very efficient (exponential ti

In imperative programming you would use varia
sequence (linear time complexity).

Fibonacci

In the last tutorial, you probably implemented Fibonacci like this:

```
(defun fib (x)
  (if (< x 2)
```


This works, but is not very efficient (exponential ti

In imperative programming you would use varia
sequence (linear time complexity).

A comparable approach in FP is to compute the sequence, e.g. as a list.

Lists in LISP


```
nil                 ; an empty list
(con
(list
(car l)
(cdr l)
(last l)            ; the last element l
(append a b)        ; combine two lists
(member e l)        ; first sublist starting e in l
(reverse l)         ; a mirror of the list
```

Lists in LISP (examples)

```
? (list 1 2 3)
(1 2 3)
? (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
? (
NIL
? (
(3 5)
? (cdr (list 1 2 3 4 5))
(2 3 4 5)
? (cdr (cdr (list 1 2 3 4 5)))
(3 4 5)
? (car (cdr (list 1 2 3 4 5)))
2
```

Fibonacci

Idea: given part of the Fibonacci sequence and a number, add that many more elements of the sequence.

```
(defun fib (n a)
  (if (
      (cons
        (+ (car a) (car (cdr a)))
        a
) ) ) )

(fib 100 (list 1 0))
```

Fibonacci

Making it a bit nicer:

- We can make optional (default) arguments
- (car (cdr x)) = (cadr x)
  - You can repeat the a, d as many times as required


```
(def
  (if (
    (fib (- n 1)
      (cons
        (+ (car a) (cadr a))
        a
) ) ) )

(fib 100)
```

A note on loops in LISP

I avoided using loops to keep the first example closer to lamda calculus.

Ther


```
(def
  (loop for f1 = 0 then f2
        and f2 = 1 then (+ f1 f2)
        repeat n finally (r
```



```
(fib 100)
```

1

[1] source: https://www.cliki.net/Fibonacci

Java

```
public boolean isPrime(long number) {
    return number > 1 &&
        LongStream
            .rangeClosed(2, (long) Math.sqrt(number))

}
isPrim
2
```


- ".rangeClosed" gives a stream of values wi
- ".noneMatch" checks the stream agains
- "variable -> expression" is a lambda abstraction!
  - It takes a value (index) from the range, and tests if it divides the number we're checking.

[2] source: https://www.voxxed.com/2015/12/functional-vs-imperative-programming-fibonacci-prime-and-factorial-in-

Java


```
public boolean isPrime(long number) {
    return number > 1 &&
        LongStream
```



```
}
isPrime(92200000000000039L)
```


3

Adding ".parallel()" is enough magic sauce to ge
good speedup.

[3] source: https://www.voxxed.com/2015/12/functional-vs-imperative-programming-fibonacci-prime-and-factorial-in-

Python

If you write much Python, you probably write more functional programming code than you thought.

```
>
>
5
>
5
>>> [x + 5 for x in grades]
[48, 73, 40, 94, 72, 70, 75]
>>> max([x + 5 for x in g
94
```

Python

```
>>> from functools import reduce
>>> prices = [43, 68, 35, 89, 67, 65, 70]
>>> sales = [3, 5, 0, 3, 2, 10, 30]
>>>
3
```

- zip combines elements from two iterables i
  - e.g. [(43, 3), (68, 5), ...]

Python

```
>>> from functools import reduce
>>> prices = [43, 68, 35, 89, 67, 65, 70]
>>> sales = [3, 5, 0, 3, 2, 10, 30]
>
3
```


- zip combines elements from two iterables i
  - e.g. [(43, 3), (68, 5), ...]
- map applies a function to every element of an it
  - e.g. [43*3, 68*5, ...]

Python

```
>>> from functools import reduce
>>> prices = [43, 68, 35, 89, 67, 65, 70]
>>> sales = [3, 5, 0, 3, 2, 10, 30]
>
3
```


- zip combines elements from two iterables i
  - e.g. [(43, 3) (68, 5), ...]
- map applies a function to every element of an it
  - e.g. [43*3, 68*5, ...]
- reduce combines the elements using a two parameter function
  - (((0+129) + 340) + 0) + ...

Review

- Revision - Lambda Calculus
  - When α-reductions are *required*
  - η-reductions
- Y Combinator
- Encodings
  - numbers (a different way)
  - pairs
  - lists
- Functional Programming
  - Using lists in LISP
  - Stream processing in Java
  - Some ubiquitous Python