The Lambda Calculus (1930s)

- Variables
- Function application
- Lambda abstraction

Just these three elements form a complete computational system
Original Syntax

\[ e ::= x \quad \text{Variables} \\
    | \lambda x . e \quad \text{Lambdas} \\
    | e_0 \ e_1 \quad \text{Applications} \]
Scheme Syntax

\[ e ::= x \quad \text{Variables} \]
\[ \quad \mid \lambda (x) e \quad \text{Lambdas} \]
\[ \quad \mid (e_0 e_1) \quad \text{Applications} \]
(define (expr? e)
  (match e
    [([? symbol? x) #t]
     ['(lambda (,[? symbol? x]) ,(? expr? e-body)) #t]
     ['(,(? expr? e0) ,(? expr? e1)) #t]
     [_ #f])))
Lambda Calculus vs. Turing machines

Lambda Calculus equivalent (in expressivity) to Turing machines.

The **Church-Turing Thesis** states that turing machines / lambda calculus can encode any computable function.
In fact, it is possible to encode (most of) any Scheme program as a lambda calculus expression via a Church/Boehm encoding.
Now let’s look at the three lambda calculus forms in detail...
An expression, *abstracted* over all possible values for a formal parameter, in this case, $x$.

$$(\lambda \ (x) \ e)$$

- **Formal parameter**
- **Function body**
An expression, abstracted over all possible values for a formal parameter, in this case, x.

\[(\lambda (x) e)\]

Formal parameter \quad Function body

In fact, you can read lambdas *mathematically* as “for all.” This observation forms the basis for universal quantification in higher-order logics implemented using typed lambda calculus variants!
Next we have applications.

\[(e \ e)\]

Expression in **function position**

Expression in **argument position**
Variables are only defined/assigned when a function is applied and its parameter bound to an argument.
How do we compute with the lambda calculus..?

Answer: via reductions, which define equivalent / transformed terms.
The most important reduction is $\beta$, which applies a function by substituting arguments

$$(((\lambda (f) \ (f \ (f \ (\lambda (x) \ x)))) \ (\lambda (x) \ x))$$
The **most important** reduction is $\beta$, which applies a function by substituting arguments

$$(((\lambda(f)(f(\lambda(x)x))))(\lambda(x)x))$$

$\downarrow$$\beta$

$$(((\lambda(x)x)(((\lambda(x)x)(\lambda(x)x))))$$
The **most important** reduction is $\beta$, which applies a function by substituting arguments

\[
((\lambda \ f) \ (f \ (f \ (\lambda \ (x) \ x)))) \ (\lambda \ (x) \ x))
\]

\[
\downarrow \quad \beta
\]

\[
((\lambda \ (x) \ x) \ ((\lambda \ (x) \ x) \ (\lambda \ (x) \ x)))
\]

\[
\downarrow \quad \beta
\]

\[
((\lambda \ (x) \ x) \ (\lambda \ (x) \ x))
\]
The **most important** reduction is $\beta$, which applies a function by substituting arguments

\[
((\lambda (f) (f (f (\lambda (x) x)))) (\lambda (x) x)) \\
\downarrow \beta \\
((\lambda (x) x) ((\lambda (x) x) (\lambda (x) x))) \\
\downarrow \beta \\
((\lambda (x) x) (\lambda (x) x)) \\
\downarrow \beta \\
(\lambda (x) x)
\]
Textual substitution. This says: replace every $x$ in $E_0$ with $E_1$.

$$(((\lambda (x) E_0) E_1) \rightarrow_\beta E_0[x \leftarrow E_1]$$

Next lecture: carefully defining substitution!
\[
\frac{((\lambda (x) x) (\lambda (x) x))}{\beta}
\]
\[
\frac{x [x \leftarrow (\lambda (x) x)]}{\beta}
\]
\[ (((\lambda (x) x) (\lambda (x) x)) \xrightarrow{\beta} (\lambda (x) x)) \]
Can you beta-reduce the following term more than once…?

\(((\lambda(x)(x\ x))\ (\lambda(x)(x\ x)))\)
\[ ((\lambda (x) (x x)) \ (\lambda (x) (x x))) \]

\[ \beta \]

\[ ((\lambda (x) (x x)) \ (\lambda (x) (x x))) \]

\[ \beta \]

\[ ((\lambda (x) (x x)) \ (\lambda (x) (x x))) \]

\[ \beta \]

\[ ((\lambda (x) (x x)) \ (\lambda (x) (x x))) \]

\[ \beta \]

\[ (\lambda (x) x x) \ (\lambda (x) x x) \]

\[ \beta \]
This specific program is known as \( \Omega \) (Omega)
\[\Omega\text{ is the smallest non-terminating program!}\]

Note how it reduces to itself in a single step!

\[
\begin{align*}
(\lambda (x) (x\ x)) (\lambda (x) (x\ x)) & \xrightarrow{\beta} \\
(\lambda (x) (x\ x)) (\lambda (x) (x\ x)) &
\end{align*}
\]