Today, we will learn about

- Lambda calculus encodings
- Church numerals
- Recursion and fixed point-combinators
**Lambda calculus encodings**

- The pure lambda calculus contains only functions as values.
- It is not exactly easy to write large or interesting programs in the pure lambda calculus.
- We can however encode objects, such as booleans, and integers.
Booleans
Booleans

We want to define functions \textit{TRUE}, \textit{FALSE}, \textit{AND}, \textit{IF}, and other operators such that the expected behavior holds, for example:

\[
\text{AND TRUE FALSE} = \text{FALSE} \\
\text{IF TRUE } e_1 e_2 = e_1 \\
\text{IF FALSE } e_1 e_2 = e_2
\]
TRUE and FALSE

\[ TRUE \triangleq \lambda x. \lambda y. x \]

\[ FALSE \triangleq \lambda x. \lambda y. y \]
The function $IF$ should behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b = TRUE \text{ then } t \text{ else } f.$$ 

The definitions for $TRUE$ and $FALSE$ make this very easy.

$$IF \triangleq \lambda b. \lambda t. \lambda f. \, b \, t \, f$$
NOT, AND, OR

\[
\text{NOT} \triangleq \lambda b. \, b \, \text{FALSE} \, \text{TRUE} \\
\text{AND} \triangleq \lambda b_1. \lambda b_2. \, b_1 \, b_2 \, \text{FALSE} \\
\text{OR} \triangleq \lambda b_1. \lambda b_2. \, b_1 \, \text{TRUE} \, b_2
\]
Church numerals

Church numerals encode the natural number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

$$
\bar{0} \triangleq \lambda f. \lambda x. x \\
\bar{1} = \lambda f. \lambda x. f \ x \\
\bar{2} = \lambda f. \lambda x. f \ (f \ x) \\
SUCC \triangleq \lambda n. \lambda f. \lambda x. f \ (n \ f \ x)
$$
Let us define addition now. Intuitively, the natural number $n_1 + n_2$ is the result of apply the successor function $n_1$ times to $n_2$.

$$ADD \triangleq \lambda n_1. \lambda n_2. n_1 \text{ SUCC } n_2$$
Recursion and the fixed-point combinators
We would like to define a function that computes factorials.

\[ \text{FACT } \triangleq \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times \text{FACT} (n - 1) \]
Recursion and the fixed-point combinators

\[
\text{FACT} \triangleq \lambda n. \text{IF} (\text{ISZERO } n) 1 (\text{TIMES } n (\text{FACT} (\text{PRED } n)))
\]
Recursion and the fixed-point combinators

Note that this is not a definition, it’s a recursive equation.
Recursion Removal Trick

- We can perform a “trick” to define a function $FACT$ that satisfies the recursive equation above.

- First, let’s define a new function $FACT’$ that looks like $FACT$, but takes an additional argument $f$.

- We assume that the function $f$ will be instantiated with an actual parameter of... $FACT’$. 
$FACT' \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f \ f \ (n - 1))$
Now we can define the factorial function $FACT$ in terms of $FACT'$. 

$$FACT \triangleq FACT' \cdot FACT'$$
Let’s try evaluating $FACT\ 3 = m$.

$m = (FACT'\ FACT')\ 3$

$= (((\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f\ f\ (n - 1)))\ FACT')\ 3$

$\rightarrow (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (FACT'\ FACT'\ (n - 1)))\ 3$

$\rightarrow \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times (FACT'\ FACT'\ (3 - 1))$

$\rightarrow 3 \times (FACT'\ FACT'\ (3 - 1))$

$\rightarrow \ldots$

$\rightarrow 3 \times 2 \times 1 \times 1$

$\rightarrow^* 6$
So we now have a technique for writing a recursive function $f$: write a function $f'$ that explicitly takes a copy of itself as an argument, and then define

$$f \triangleq f' f'.$$
Fixed point combinators

Alternatively, we can express a recursive function as the fixed point of some other, higher-order function, and then find that fixed point.
Thus $FACT$ is a fixed point of the following function.

$$ G \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f \ (n - 1)) $$
Recall that if $g$ is a fixed point of $G$, then we have $G \ g = g$. 
Fixed point combinator

- A *combinator* is simply a closed lambda term

- Our functions *SUCC* and *ADD* are examples of combinators.

- It is possible to define programs using only combinators, thus avoiding the use of variables completely.
The Y combinator

The Y combinator is defined as

\[ Y \triangleq \lambda f. (\lambda x. f (x \, x)) (\lambda x. f (x \, x)). \]

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators.
The fixed point of the higher order function $G$ is equal to $G (G (G (G (G \ldots))))$. Intuitively, the $Y$ combinator unrolls this equality, as needed.
Let’s see it in action, on our function $G$, where

$$G = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n - 1))$$

and the factorial function is the fixed point of $G$. (We will use CBN semantics.)
\[ \text{FACT} = Y \ G \\
\quad = (\lambda f. (\lambda x. f \ (x \ x)) \ (\lambda x. f \ (x \ x))) \ G \\
\quad \rightarrow (\lambda x. G \ (x \ x)) \ (\lambda x. G \ (x \ x)) \\
\quad \rightarrow G \ ((\lambda x. G \ (x \ x)) \ (\lambda x. G \ (x \ x))) \\
\quad =_{\beta} G \ (\text{FACT}) \\
\quad = (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f \ (n - 1))) \ \text{FACT} \\
\quad \rightarrow \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{FACT} \ (n - 1)) \]
Note that the $Y$ combinator works under CBN semantics, but not CBV. (What happens when we evaluate $Y \ G$ under CBV?)
There is a variant of the $Y$ combinator, $Z$, that works under CBV semantics. It is defined as

$$Z \triangleq \lambda f. (\lambda x. f (\lambda y. x \: x \: y)) \: (\lambda x. f (\lambda y. x \: x \: y)).$$
The Turing fixed-point combinator, denoted \( \Theta \), was discovered by Alan Turing.
The Turing fixed-point combinator

Suppose we have a higher order function $f$, and want the fixed point of $f$. We know that $\Theta f$ is a fixed point of $f$, so we have

$$\Theta f = f (\Theta f).$$
This means, that we can write the following recursive equation for \( \Theta \).

\[
\Theta = \lambda f. f \ (\Theta \ f)
\]

Now we can use the recursion removal trick we described earlier! Let’s define
\( \Theta' = \lambda t. \lambda f. f \ (t \ t \ f) \), and define

\[
\Theta \triangleq \Theta' \ \Theta'
\]

\[
= (\lambda t. \lambda f. f \ (t \ t \ f)) \ (\lambda t. \lambda f. f \ (t \ t \ f))
\]
Let’s try out the Turing combinator on our higher order function $G$ that we used to define $FACT$. Again, we will use CBN semantics.

\[
FACT = \Theta G \\
= ((\lambda t. \lambda f. f \, (t \, t \, f)) \, (\lambda t. \lambda f. f \, (t \, t \, f))) \, G \\
\longrightarrow (\lambda f. f \, ((\lambda t. \lambda f. f \, (t \, t \, f)) \, (\lambda t. \lambda f. f \, (t \, t \, f))) \, f)) \, G \\
\longrightarrow G \, ((\lambda t. \lambda f. f \, (t \, t \, f)) \, (\lambda t. \lambda f. f \, (t \, t \, f))) \, G \\
= G \, (\Theta G) \\
= (\lambda f . \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f \, (n - 1))) \, (\Theta G) \\
\longrightarrow \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((\Theta G) \, (n - 1)) \\
= \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (FACT \, (n - 1))
\]