1. Normal Form

- An expression containing no possible beta reductions is said to be in normal form. A normal form expression is one containing no redexes (reducible expressions), that is, one with no subexpressions of the form `(λx.f) g`.
- Examples of normal form expressions:
  - `x` where `x` is a variable.
  - `x e` where `x` is a variable and `e` is a normal form expression.
  - `λx.e` where `x` is a variable and `e` is a normal form expression.
- The expression `(λx.x x)(λx.x x)` does not have a normal form because the entire expression is a redex that always evaluates to itself. We can think of this expression as a representation for an infinite loop.

2. Reduction Strategies

- A reduction strategy specifies the order in which beta reductions for a lambda expression are made.
- We say a redex is to the left of another redex if its lambda appears further left.
- The leftmost outermost redex is the leftmost redex not contained in any other redex.
- The leftmost innermost redex is the leftmost redex not containing any other redex.

- Some reduction orders for a lambda expression may yield a normal form while other orders may not. For example, consider the given expression

  ```
  (λx.1)((λx.x x)(λx.x x))
  ```

  This expression has two redexes:

  1. The entire expression is a redex in which we can apply the function `(λx.1)` to the argument `((λx.x x)(λx.x x))` to yield the normal form 1. This redex is the leftmost outermost redex in the given expression.
  2. The subexpression `((λx.x x)(λx.x x))` is also a redex in which we can apply the function `(λx.x x)` to the argument `(λx.x x)`. Note that this redex is the leftmost innermost redex in the given expression. But if we evaluate this redex we get same subexpression: `(λx.x x)(λx.x x)` → `(λx.x x)(λx.x x)`. Thus, continuing to evaluate the leftmost innermost redex will not terminate and no normal form will result.

- As a second example, consider the expression

  `(λx. λy. y)((λz.z z)(λz.z z))`

  This expression has two redexes:

  0. The entire expression is a redex in which we apply the function `(λx. λy. y)` to the argument `((λz.z z)(λz.z z))` to yield the normal form `(λy. y)`. This redex is the leftmost outermost redex in the given expression.
  1. The subexpression `((λz.z z)(λz.z z))` is also a redex in which we can apply the function `(λz.z z)` to the argument `(λz.z z)`. Note that this redex is the leftmost innermost redex in the given expression. But if we evaluate this redex we get same subexpression: `((λz.z z)(λz.z z))` → `((λz.z z)(λz.z z))`. Thus, continuing to evaluate the leftmost innermost redex will not terminate and no normal form will result.

- There are two common reduction orders for lambda expressions: normal order evaluation and applicative order evaluation.

  Normal order evaluation

  - In normal order evaluation we always reduce the leftmost outermost redex at each step.
  - The first reduction order in each of the two examples above is a normal order evaluation.
  - A remarkable property of lambda calculus is that every lambda expression has a unique normal form if one exists. Moreover, if an expression has a normal form, then normal order evaluation will always find it.

  Applicative order evaluation

  - In applicative order evaluation we always reduce the leftmost innermost redex at each step.
  - Applicative order evaluates the arguments of a function before evaluating the function itself.
  - The second reduction order in each of the two examples above is an applicative order evaluation.
  - Thus, even though an expression may have a normal form, applicative order evaluation may fail to find it.

3. The Church-Rosser Theorems

- A remarkable property of lambda calculus is that every expression has a unique normal form if one exists.
- **Church-Rosser Theorem I:** If `e →* f` and `e →* g` by any two reduction orders, then there always exists a lambda expression `h` such that `f →* h` and `g →* h`.
  - A corollary of this theorem is that no lambda expression can be reduced to two distinct normal forms. To see this, suppose `f` and `g` are in normal form. The Church-Rosser theorem says there must be an expression `h` such that `f` and `g` are each reducible to `h`. Since `f` and `g` are in normal form, they cannot have any redexes so `f = g = h`.
  - This corollary says that all reduction sequences that terminate will always yield the same result and that result must be a normal form.
  - The term *confluent* is often applied to a rewriting system that has the Church-Rosser property.
- **Church-Rosser Theorem II:** If `e →* f` and `f` is in normal form, then there exists a normal order reduction sequence from `e` to `f`.

4. The Y Combinator

- The `Y` combinator (sometimes called the paradoxical combinator) is a function that takes a function `G` as an argument and returns `G(YG)`. With repeated applications we can get `G(G(YG)), G(G(G(YG))),... .`
- We can implement recursive functions using the `Y` combinator.
- `Y` is defined as follows:

```
(λf.(λx.f(x x))(λx.f(x x)))
```

- Let us evaluate `YG` where `G` is any expression:

```
(λf.(λx.f(x x))(λx'.f(x' x'))) G
→ (λx.G(x x))(λx'.G(x' x'))
→ G((λx'.G(x' x'))(λx'.G(x' x')))
↔ G((λf.(λx.f(x x))(λx.f(x x)))G)
= G(YG)
```

- Thus, `YG →* G(YG)`; that is, `YG` reduces to a call of `G` on `(YG)`.
- We will use `Y` to implement recursive functions.
- `Y` is an example of a fixed-point combinator.

5. Implementing Factorial using the Y Combinator

- If we could name lambda abstractions, we could define the factorial function with the following recursive definition:

```
FAC = (λn.IF (= n 0) 1 (* n (FAC (- n 1 ))))
```

where `IF` is a conditional function.

- However, functions in lambda calculus cannot be named; they are anonymous.
- But we can express recursion as the fixed-point of a function `G`. To do this, let us simplify the essence of the problem. We begin with a skeletal recursive definition:

```
FAC = λn.(... FAC ...)
```

- By performing beta abstraction on `FAC`, we can transform its definition to:

```
FAC = (λf.(λn.(... f ...))) FAC
    = G FAC
```

where

```
G = λf.λn.IF (= n 0) 1 (* n (f (- n 1 )))
```

Beta abstraction is just the reverse of beta reduction.

- The equation

```
FAC = G FAC
```

says that when the function `G` is applied to `FAC`, the result is `FAC`. That is, `FAC` is a fixed-point of `G`.

- We can use the Y combinator to implement `FAC`:

```
FAC = Y G
```

- As an example, let compute `FAC 1`:

```
FAC 1 = Y G 1
      = G (Y G) 1
      = λf.λn.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G) 1
      → λn.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))1
      → IF (= n 0) 1 (* n ((Y G) (- 1 1 )))
      → * 1 (Y G 0)
      = * 1 (G(Y G) 0)
      = * 1((λf.λn.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G) 0)
      → * 1((λn.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))0
      → * 1(IF (= 0 0) 1 (* 0 ((Y G) (- 0 1 )))
      → * 1 1
      → 1
```