Lec 4

Evaluation Strategies :

- Substitution model → name substituted with its defining expr.
- Call-by-value [Eager] ← Apply the fnc to reduced arguments
- Call-by-name [lazy] ← Apply the fnc to unreduced arguments

formalized in the λ-Calculus

- Both strategies Evaluate the same value so long as
  - reduced expr is a pure fnc
  - both evaluations terminate

- Which one is better ?

Call-by-name

1. Redundant repeats on computations

memoization ←

2. Termination is guaranteed

Call-by-name

Eg:

SumOfSquares 3 (2+2)

= (Square 3) + (Square (2+2))

= 3*3 + (Square (2+2))

Substitution

= 9 + (2+2) * (2+2)

→ Now operand resolution demands computation

Call-by-value

Adv: Evaluates every fnc argument only once

Call-by-name

Adv: Short circuit evaluation

Termination

- If CBV one terminates $\Rightarrow$ CBN on e will also terminate

  ↳ the other direction is not true

Eg:

$$Proj\ x\ y = x$$

$$loop = loop$$

CBN

$$Proj\ (2+2)\ loop = (2+2) = 4$$

CBV

$$= Proj\ 4\ loop = Proj\ 4\ loop = \cdots$$

Lexical Scoping and Nested forms

- How to create scopes & nested scopes?
  - Why do we want it?
    - to create a "block-structured" form in my program
  - How do we do it?
    - By creating local bindings which are not seen elsewhere

Sol 1:

Let Exprs

Syntax: let { bindings } in Expr

For eg:
```
let x = 2;
    y = 3;
in  x * y
```
→ indented as much as the "let"

Sol2: Where Clause

Sometimes it is convenient to scope bindings over several guarded equations

for Eg:

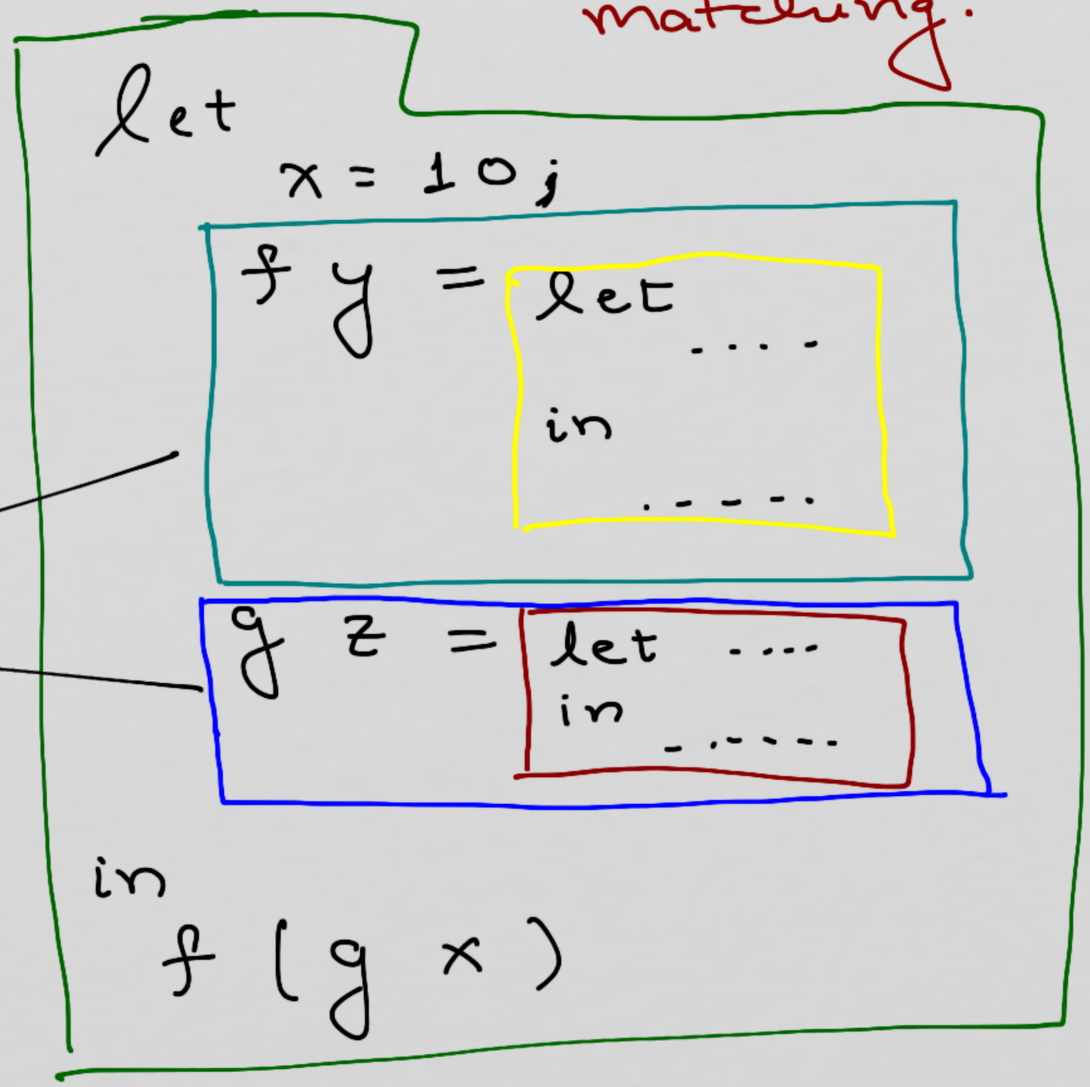
```
f x y | y > z  =  ....
      | y = z  =  ....
      | y < z  =  ....
      where z = x * x
```

guards

Note: The above scoping cannot be done with a let expr. This is because "let" scopes only over the expr which it encloses

Note:
- let expr itself is an expression
- Where clause is not an expr — it is a part of syntax for function declarations
- & Case expressions → related to pattern matching.

In general
- Disjoint scopes



Overlapping or spanning scopes here is not allowed

Names

- A name may occur either as being **defined** or as **use** of a previously defined name

- The **use** of a name refers to textually the most recent definition in the innermost enclosing scope.

Eg:

```
let x = 10; z = 5;
    f y = let x = 15;
          in  x + y * z
in
    f x ⟶ ? refers to which defn.
```