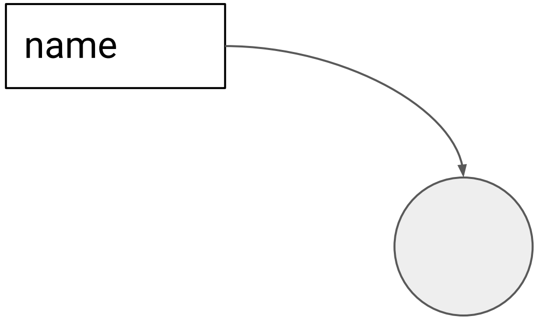
Scopes and Symbol Binding

Ken Q Pu, Faculty of Science, Ontario Tech University

Symbol Binding

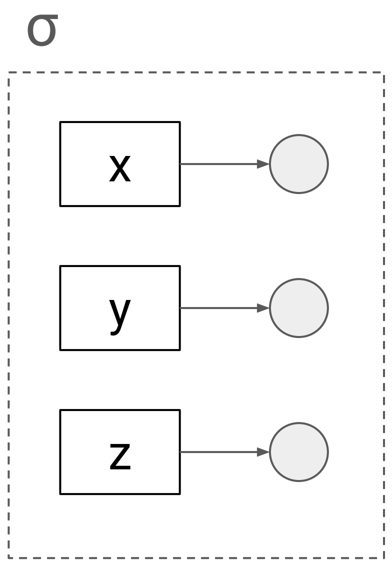
- A symbol is a name in the Lambda Calculus sense.
- A value is any data in Clojure (scalar, vector, list, function, ...)
- A binding is when a symbol is associated with its value



Scope

A scope is a collection of symbol names and their bindings.

$$\sigma : \text{Names} \longrightarrow \text{Values}$$

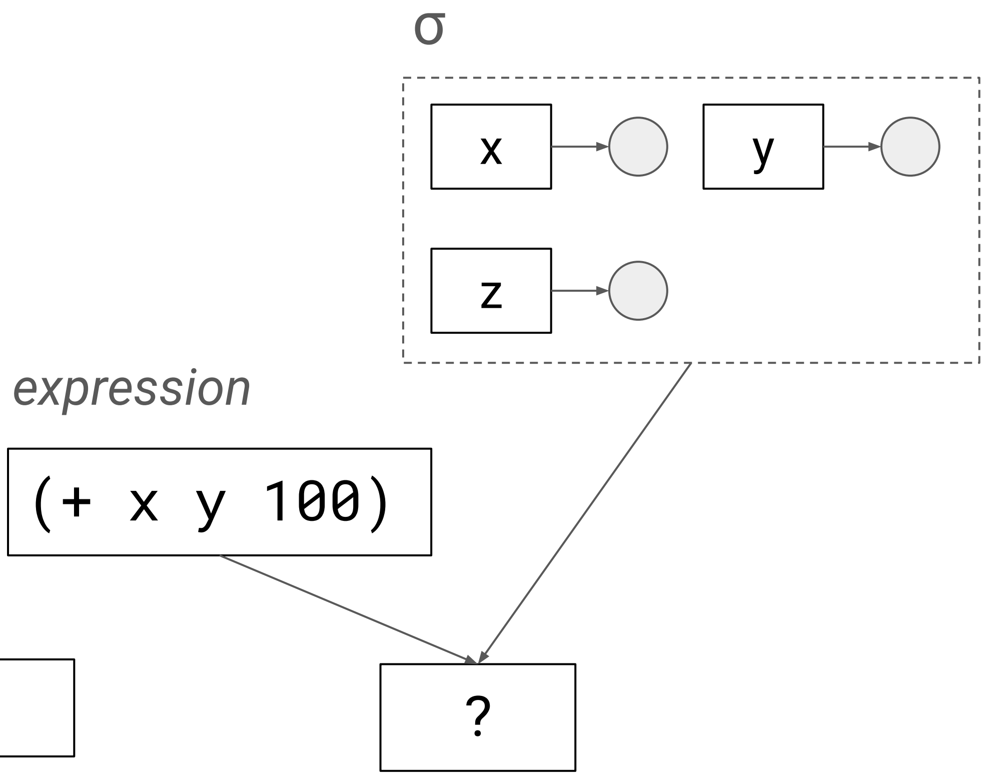


Eval revisited

To evaluate an expression, we need:

- Resolve the free variables to their values using the current scope
- Apply functions using beta-conversion

$$\boxed{\text{Eval : Expression, Scope} \longrightarrow \text{Values}}$$



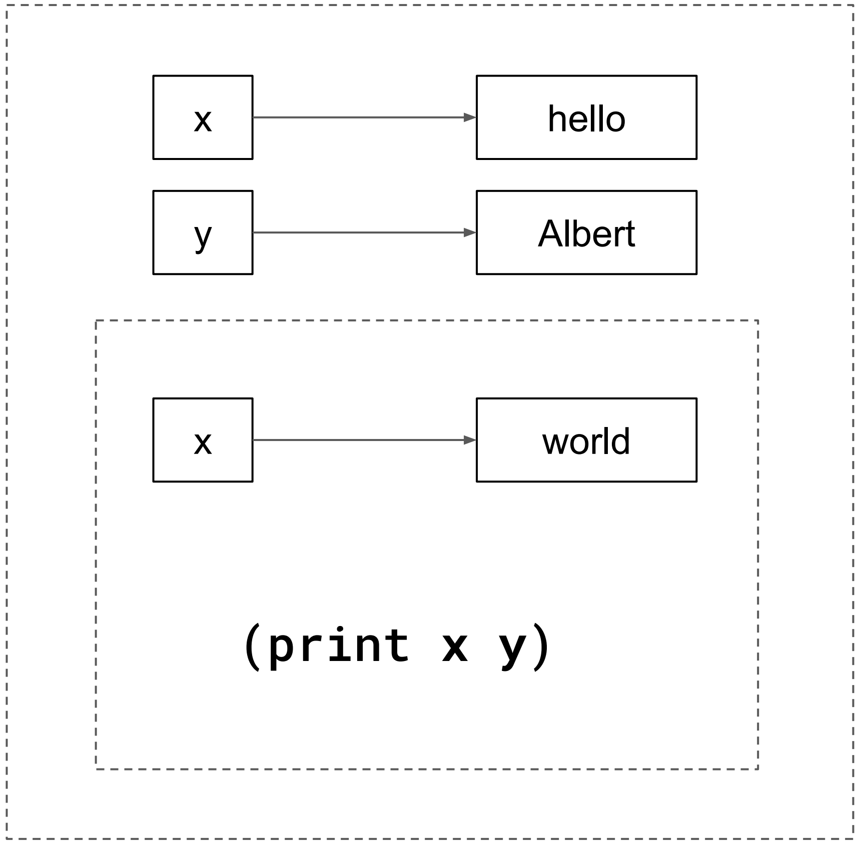
Nested Scopes

Scopes can be nested.

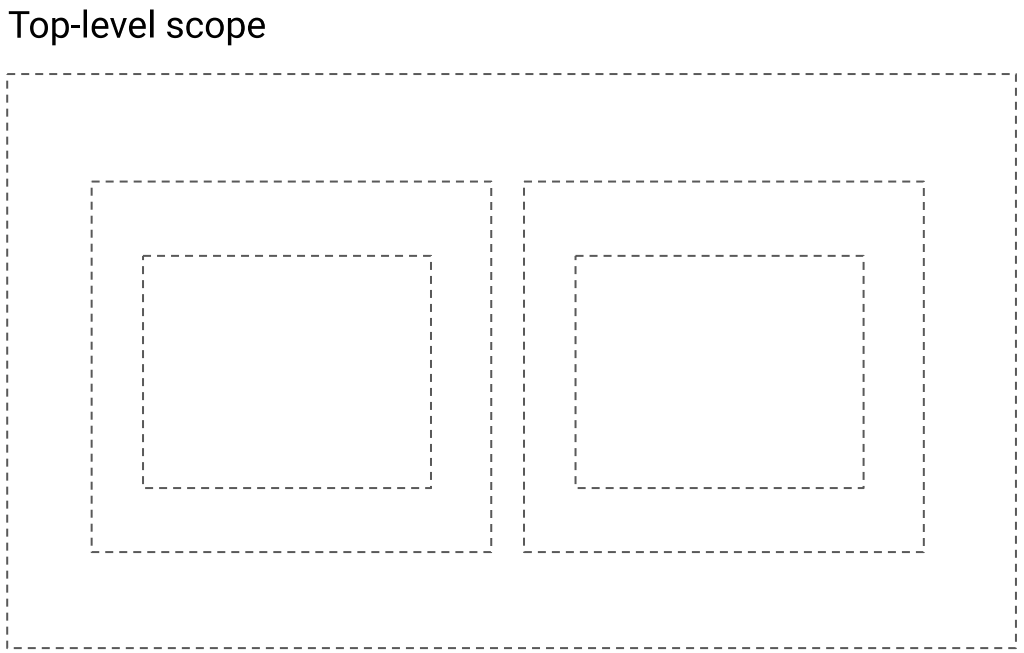
The scope of an expression is determined by the most inner bindings.

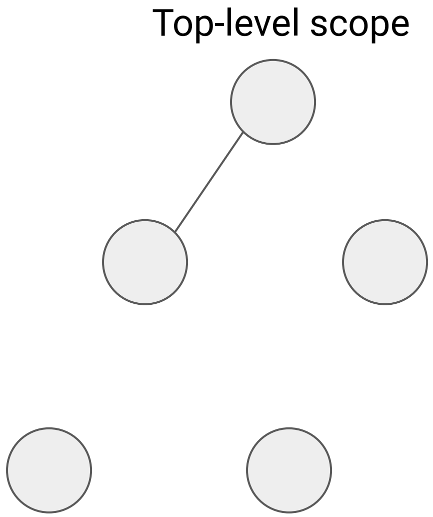
(print x y) has the bindings of:

- x -> "world"
- y -> "Albert"



Clojure's scopes

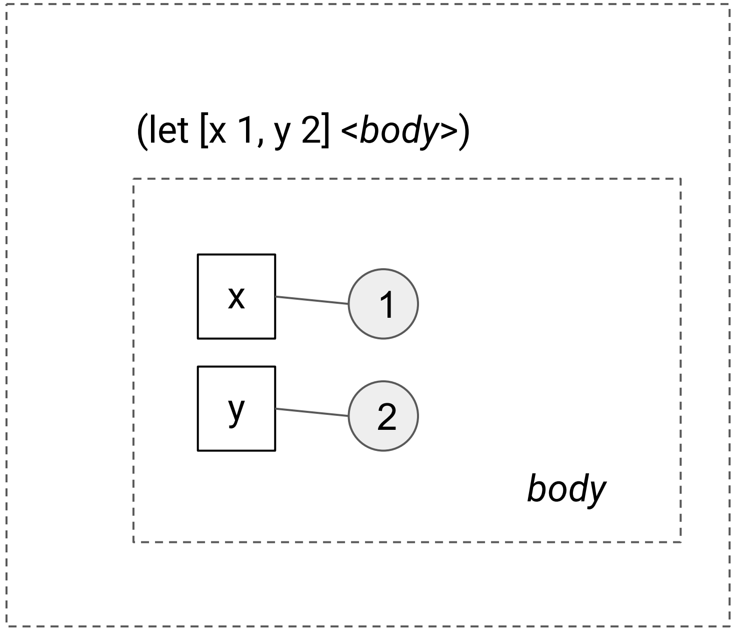




The **let**-form

The let-form creates a nested scopes.

```
(let [ <x> <expression>
       <y> <expression>
       ... ]
  <body>)
```



Examples

```
(for [[name grade] {"Jack" 89
                    "Jill" 90
                    "Joe"  76}
  :when (>= grade 80)
  :let [status (cond (>= grade 90) "great"
                     (>= 85)       "good"
                     :else         "not great"))]]
  (format "%s is %s" name status)
```

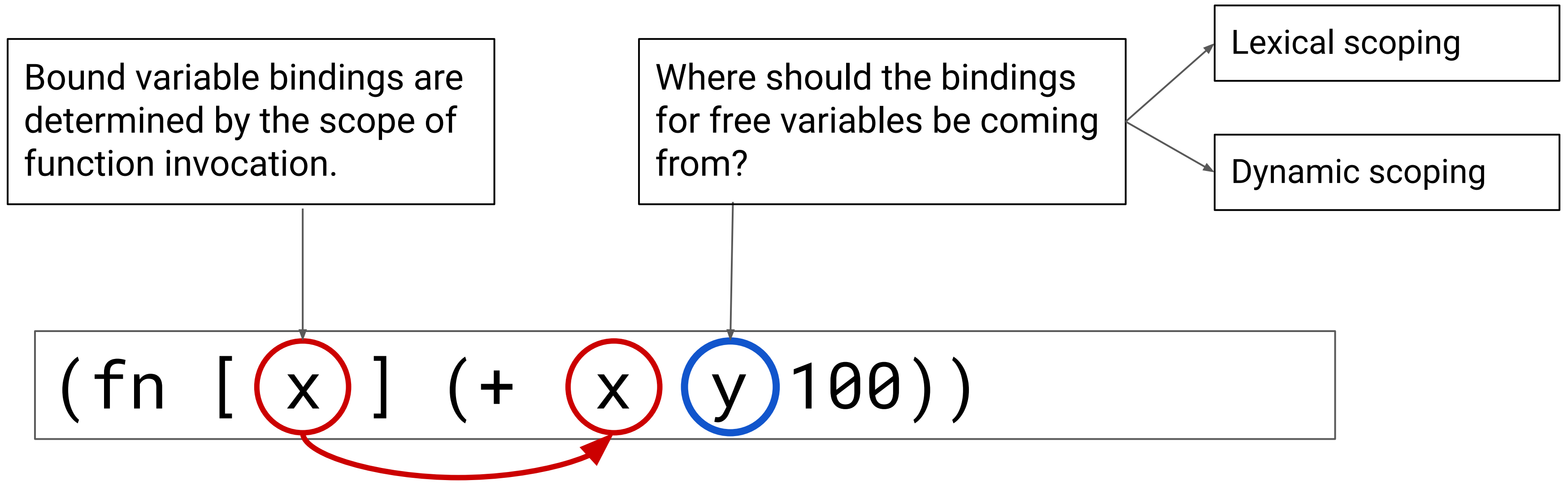
Let's refactor this Clojure code with nested scopes.

Examples

```
(let [students {"Jack" 89
                "Jill" 90
                "Joe"  76}
      get-status (fn [grade]
                   (cond (>= grade 90) "great"
                         (>= 85)       "good"
                         :else         "not great")))]
  (for [[name grade] students]
       :when (>= grade 80)]
    (format "%s is %s" name (get-status grade))))
```

Functions

Function declaration creates a nested scope for its body with both **free variables** and **bound variables**.



Bound variable bindings are determined by the scope of function invocation.

Where should the bindings for free variables be coming from?

Lexical scoping

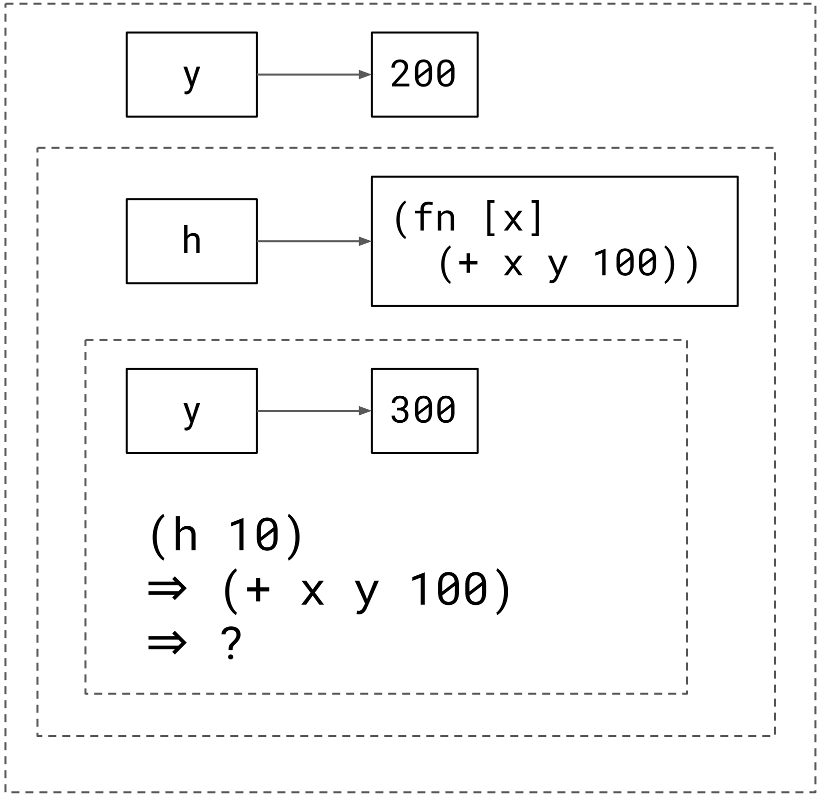
Dynamic scoping

```
(fn [x] (+ x y 100))
```

Lexical Scoping vs Dynamic Scoping

Both lexical scoping and dynamic scoping rules refer to how free variables in functions are determined when the function is invoked.

Modern programming languages strongly discourages dynamic scoping. So, lexical scoping is the **default** scoping rule.



Lexical Scoping

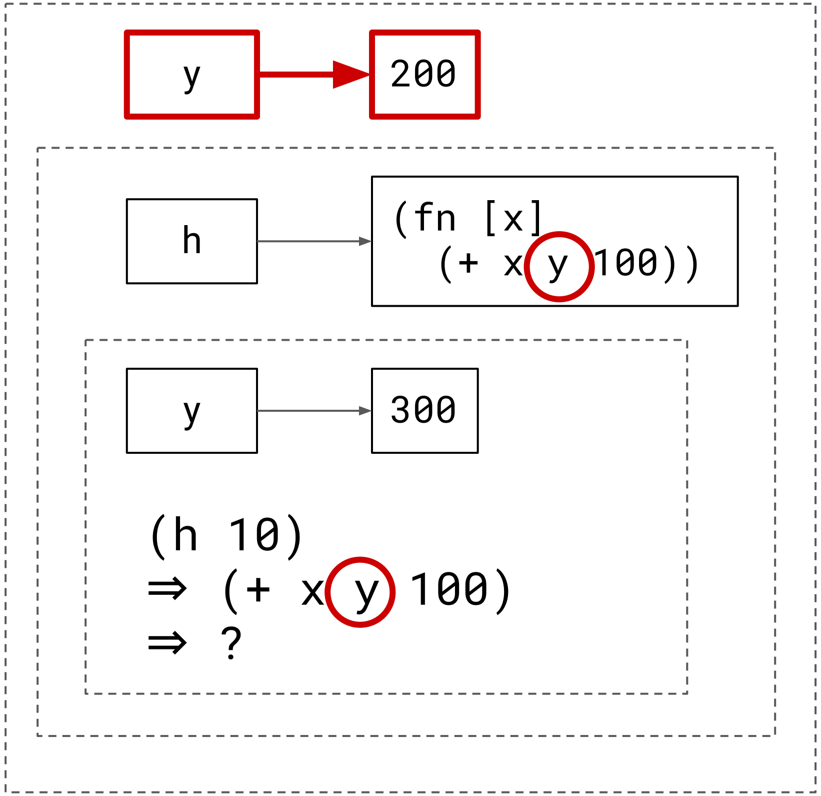
Lexical scoping rule states that **free variables** in a function body is to be resolved according to the scope of **function declaration**.

This is the default scoping rule for Clojure (Javascript, Python, etc).

Lexical Scoping

```
(let [y 200
      h (fn [x]
          (+ x y 100))]
  (let [y 300]
    (h 10)))
```

```
(h 10)
⇒ (+ x y 100)
⇒ (+ 10 200 100)
⇒ 310
```



Dynamic Scoping

Dynamic scoping rule states that **free variables** in the function body is resolved using the scope of the function invocation.

Clojure highly discourages dynamic scoping. Maintaining consistent behaviour of functions is very difficult when the author of the function does not know the values of the free variables in the body.

Clojure supports dynamic scoping with special forms.

We will cover this after the **def**-form.

Def-form: top-level bindings

The def-form is used to create bindings at the top-level scope.

```
(def <symbol> <expression>)
```

We should consider symbols created by def-forms ***global variables****.*

Def-form: top-level bindings

```
(def students {"Jack" 89
               "Jill" 90
               "Joe"  76})

(def get-status (fn [grade]
                  (cond (>= grade 90) "great"
                        (>= 85)       "good"
                        :else         "not great"))))

(for [[name grade] students]
      :when (>= grade 80)]
  (format "%s is %s" name (get-status grade)))
```

Is it a good idea to have global variable names such as students and get-status?

Back to dynamic scoping

Clojure imposes several restrictions on the function declaration and function invocation to activate dynamic scoping rule on selected free variables.

(1) Free variables need to be declared to be dynamic at the top-level. They are called dynamic vars.

```
(def ^:dynamic <symbol>)
```

(2) Dynamic top-level symbols can be used as free variables in the function body.

(3) The binding of a dynamic var is provided by **binding**-form when invoking the function:

```
(binding [<symbol> <expression>]
  (<function> ...))
```

A case for dynamic scoping

```
(def ^:dynamic tax-rate)

(def total-cost
  (fn [price]
    (let [tax (* tax-rate price)]
      (+ price tax))))
```

```
(def alberta-cost [price]
  (binding [tax-rate 0.05]
    (total-cost price)))

(def ontario-cost [price]
  (binding [tax-rate 0.13]
    (total-cost price)))

(def nova-scotia-cost [price]
  (binding [tax-rate 0.15]
    (total-cost price)))
```

Top-level function bindings

```
(def get-student-status
 (fn [student grade]
  ...))
```

```
(defn get-student-status [student grade]
  ...)
```