# 16. Scoping, Binding, Lexical Addressing

Lecture 16 -- Scoping, Binding, Lexical Addressing.pdf

Main Idea:
We can index the values in the environment instead of using variable names. This can also be done in static scopeing.

## Static (Lexical) Scoping

- Usage of the scopes for matching variables.

## Dynamic Scoping

- No scope approach, just match everything linearly, disregard functions.

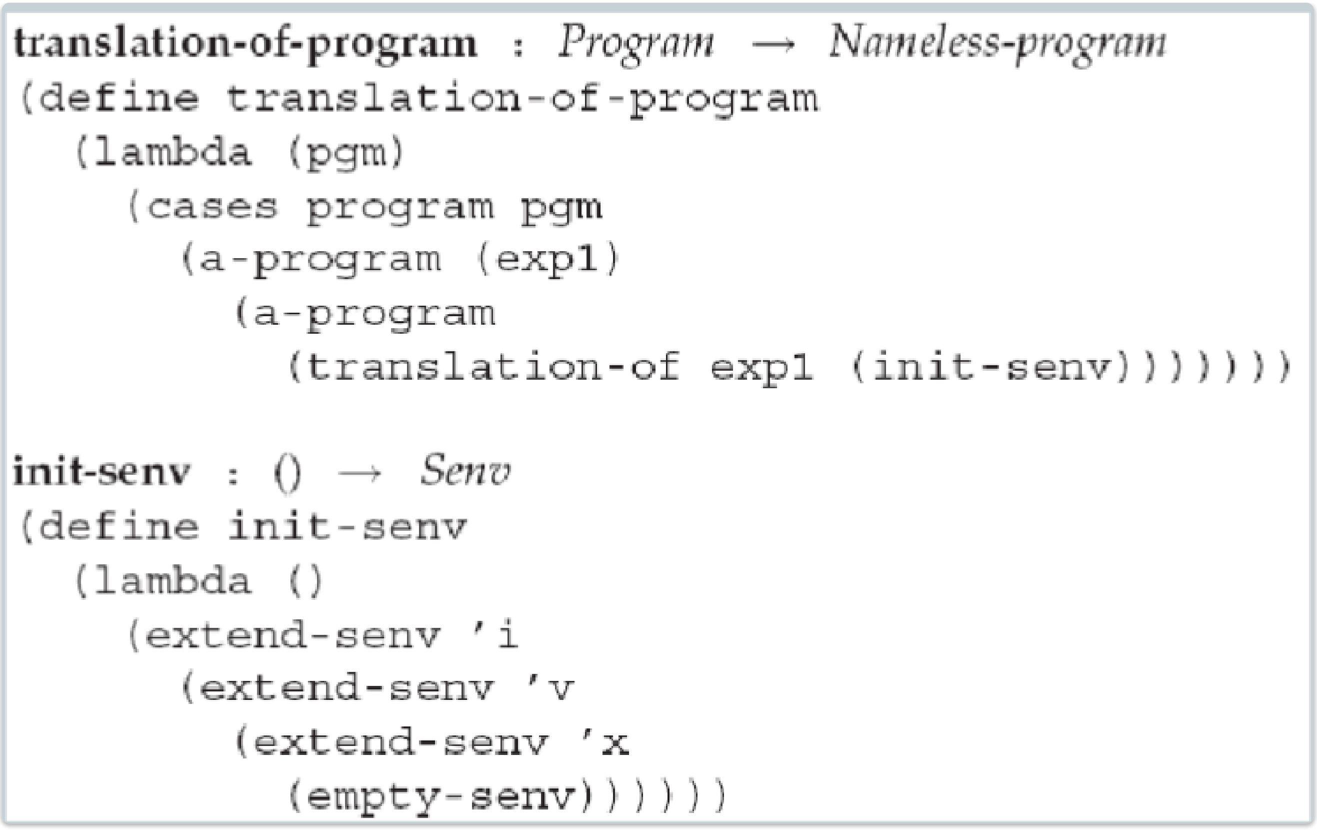
Example:

```scheme
(define (func-a)
  (define x 10)
  (func-b))

(define (func-b)
  x)

(func-a)
```

Answer:

- Under Static (Lexical) Scoping: The Scheme interpreter with static scoping will result in an error when `(func-a)` is executed. This is because `x` is lexically scoped within `func-a`, and thus, it is not accessible within `func-b`. In lexical scoping, a variable is only accessible within the block where it is defined and its sub-blocks.
- Under Dynamic Scoping: If the Scheme interpreter uses dynamic scoping, executing `(func-a)` will return `10`. This is because, in dynamic scoping, the visibility of a variable is based on the call stack, not the lexical structure of the code. When `func-b` is called from `func-a`, `x` is present in the call chain, hence accessible in `func-b`.

# Translator (named lang nameless lang)

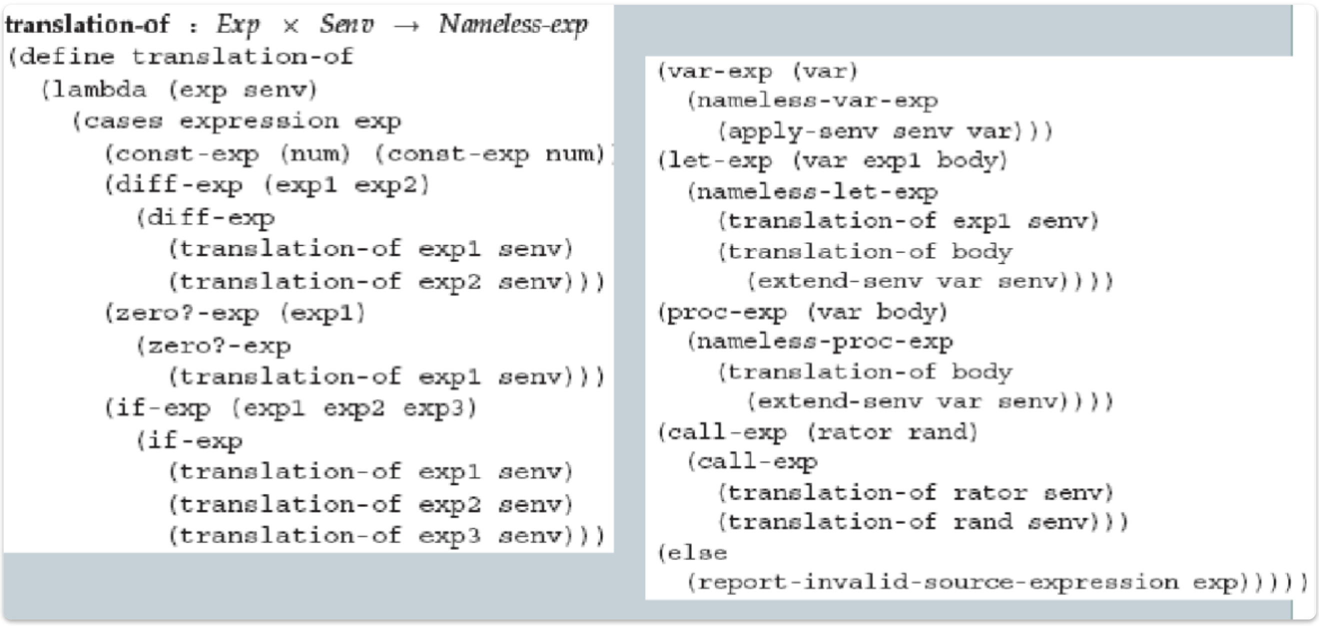


**translation-of-program** : *Program* → *Nameless-program*

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))
```

**init-senv** : () → *Senv*

```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```



**translation-of** : *Exp* × *Senv* → *Nameless-exp*

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp)))))
```