Principle in binding of variables

"A bound variable in an expression can be renamed uniformly to another variable that does not appear free in the expression without changing the meaning of the expression."

- Example:

$$\forall x\ (\exists y\ f(x,y)) = \forall x\ (\exists v\ f(x,v))$$
$$= \forall y\ (\exists v\ f(y,v))$$
$$= \forall y\ (\exists x\ f(y,x))$$
$$\neq \forall y\ (\exists y\ f(y,y))$$

→ y was free in $f(y,x)$

The declaration $\forall y$ *in* $\forall y\ (\exists y\ f(y,y))$ *is vacuous and invisible to* $f(y,y)$ *in* $(\exists y\ f(y,y))$ *because* $\exists y$ *supercedes* $\forall y$*.*

Using the principle

- What does this mean to a programming language?

code 1 → `int x; { int * y;  … = x + *y; … }`

code 2 → `int x; { int * v;  … = x + *v; … }`

code 3 → `int y; { int * v;  … = y + *v; … }`

code 4 → `int y; { int * x;  … = y + *x; … }`

code 5 → `int y; { int * y;  … = y + *y; … }`

➔

useful for detecting homework program copying?

Static vs. dynamic scopes

- Each language has its own rules that tell us where to find the declaration for a name in a program. The rules are called the **scope rules** (or scope regime).
- Scope rules can be categorized largely into two kinds:
  - static scope (Fortran, Pascal, C, Scheme, Common Lisp)
  - dynamic scope (pure Lisp, APL, SmallTalk)
- static scope
  - A scope of a variable is the region of text for which a specific binding of the variable is visible.
  - Therefore, the connection between references and declarations can be made lexically, based on the text of the program.
  - At compile time, a free variable is bound by a declaration in textually enclosing scopes/environments.

Static vs. dynamic scopes

- dynamic scope
  - The connection between references and declarations cannot be determined lexically since, in general, a variable is not declared until run-time, and may even be redeclared as the program executes.
  - At run-time, a free variable in a procedure is bound by a declaration in the environment from which the procedure is called.
- Program execution may behave differently depending on whether static or dynamic scoping is used.
  - Consider the following program:

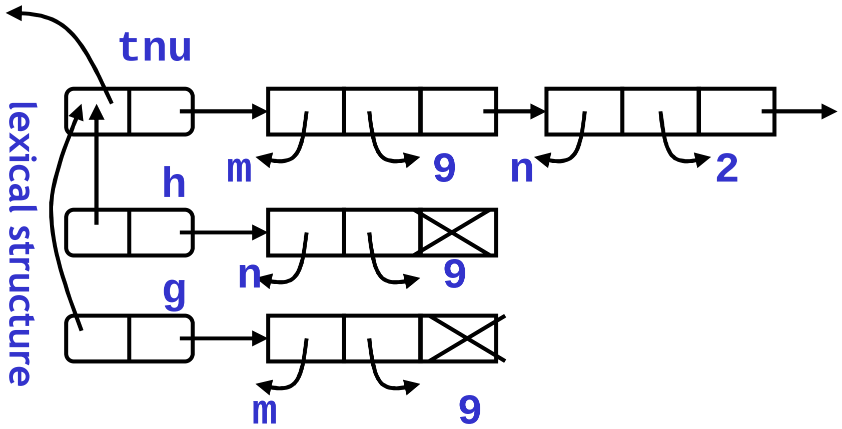
```
(define g (lambda (m) (- m n)))
```

```
> (define (tnu m n) (begin              // main program tnu
        (define (g m) (- m n))          // subprogram g
        (define (h n) (* m (g n)))      // subprogram h
        (+ (h m) n) ))                  // body of program tnu
(tnu)
> (tnu 9 2)
???
```

// What will be the output if this is a pure
// Lisp code? What if it is a Scheme code?

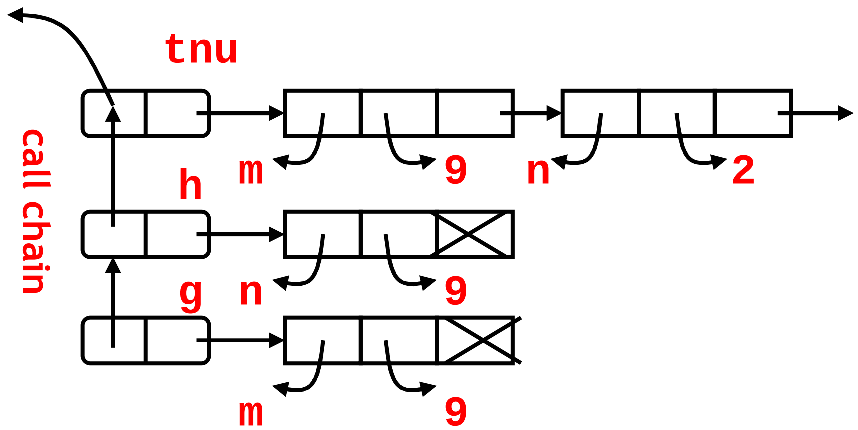
Environments with static/dynamic scopes

- Environment configuration when g is called



Static Scoping (Scheme)

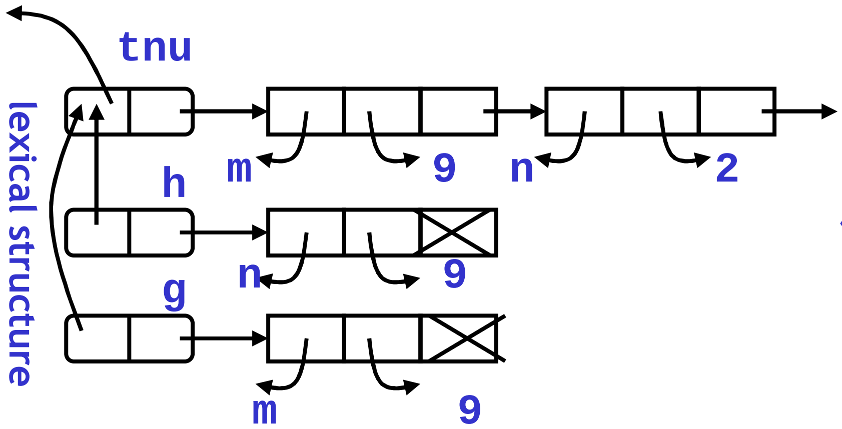
Dynamic Scoping (Pure Lisp)



Note: *different scope rules differentiate binding of a free variable!*

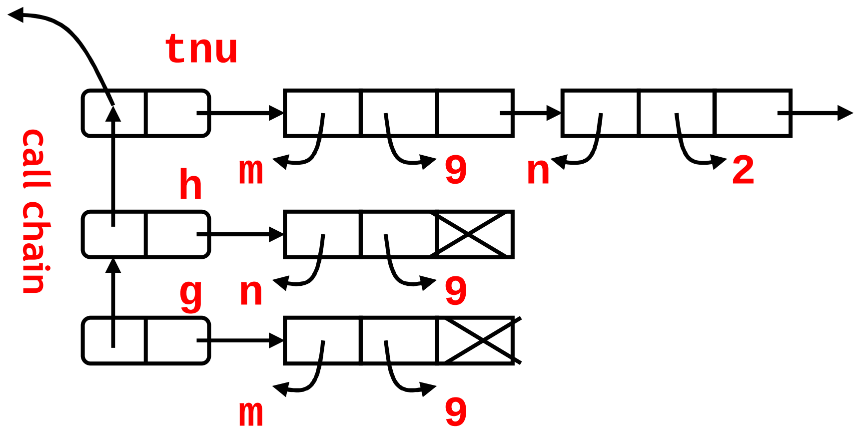
Environments with static/dynamic scopes

- Environment configuration when g is called




```
(define (tnu m n) (begin
    (define (g m) (- m n))
    (define (h n) (* m (g n)))
    (+ (h m) n) ))
```

```
(define (tnu m n) (begin
    (define (g m) (- m n))
    (define (h n) (* m (g n)))
    (+ (h m) n) ))
```



Note: *different scope rules differentiate binding of a free variable!*

Implicit parameter passing

"Develop a routine **add-nth-powers(x y n)** *which returns* x^n+y^n*."*

- Using dynamic scoping

```
> (define (add-nth-powers x y n) (+ (n-expt x) (n-expt y)))
> (define (n-expt m) (expt m n))                // returns m^n
> (add-nth-powers 4 3 2)                        // the output = ?
```

→ *This works because dynamic scoping allows* **implicit parameter passing.**

- What if static scoping is used?

  Static scoping does not allow implicit parameter passing. So …

  1. pass the parameter n explicitly; or…

```
> (define (add-nth-powers x y n) (+ (n-expt x n) (n-expt y n)))
> (define (n-expt m n) (expt m n))
```

  2. use a global variable n-global.

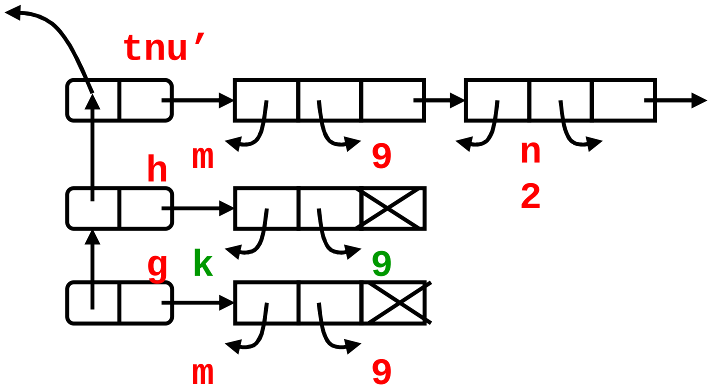
```
> (define n-global 0)                     // meaningless initialization
> (define (add-nth-powers x y n)
        (begin (set! n-global n) (+ (n-expt x) (n-expt y))))
> (define (n-expt m) (expt m n-global))
```

  → *any other solution?*

Problems of dynamic scoping

- more difficult to read and understand a program because
  - The *principle of binding* is violated.

```
> (define (tnu' m n) (begin
     (define (g m) (- m n))
     (define (h k) (* m (g k)))
     (+ (h m) n) ))
```



→ **tnu** *and* **tnu'** *are different if dynamic scoping is used!*

  - The meaning of a routine with free variables depends on call chain because of a *screening problem*.

    e.g.) In the routine **tnu,** the subroutine **h** captures the free variable **n** in the routine **g**, changing the meaning of **g**.

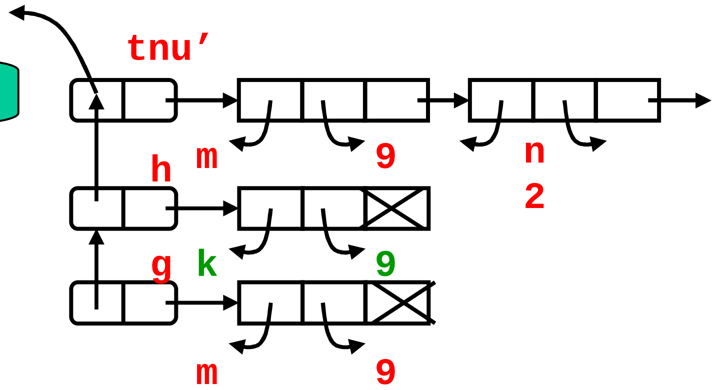
- Typically more expensive because variable name comparison thru a call chain is involved in program execution.

Problems of dynamic scoping

- more difficult to read and understand a program because
  - The *principle of binding* is violated.

In tnu()

```
> (define (tnu' m n) (begi
    (define (g m) (- m n))
    (define (h k) (* m (g k)))
    (+ (h m) n) ))
```



→ **tnu** *and* **tnu'** *are different if dynamic scoping is used!*

  - The meaning of a routine with free variables depends on call chain because of a *screening problem*.

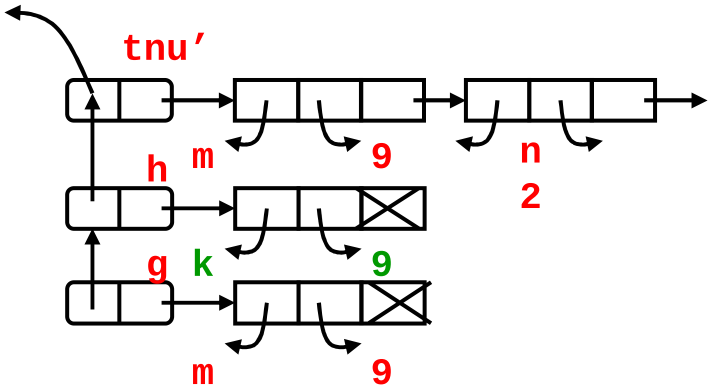
    e.g.) In the routine **tnu,** the subroutine **h** captures the free variable **n** in the routine **g**, changing the meaning of **g**.

- Typically more expensive because variable name comparison thru a call chain is involved in program execution.

Problems of dynamic scoping

- more difficult to read and understand a program because
  - The *principle of binding* is violated.

```
> (define (tnu' m n) (begin
     (define (g m) (- m n))
     (define (h k) (* m (g k)))
     (+ (h m) n) ))
```



→ **tnu** *and* **tnu'** *are different if dynamic scoping is used!*

  - The meaning of a routine with free variables depends on call chain because of a *screening problem*.

    e.g.) In the routine **tnu,** the subroutine **h** captures the free variable **n** in the routine **g**, changing the meaning of **g**.

- Typically more expensive because variable name comparison thru a call chain is involved in program execution.

Life time

- Life time of a variable is the interval of time for which a specific binding of the variable is active. → *cf: scope*
- During the life time of a variable, a variable is bound to memory storage.
- Let **x** be a **simple/automatic** variable declared inside a scope **S**.
  - The life time of **x** begins when program execution enters the scope **S**, and ends when execution leaves the scope.
  - Only when the binding of **x** is visible, **x** is a **live** variable.

Life time

- Let x be a **static** variable declared inside a scope S.
  - The life time of x begins when program execution starts, and ends when program execution terminates.
  - Even when the binding of x is not visible, x remains a live variable.

C++

```
int f() {
  static int die_hard;     // static variable, as it says
  int temp;                // automatic variable
  … f() …
}
main() {
   … f() …
}
```

  - What are the life time and scope of **`die_hard`**?
  - What are the life time and scope of **`temp`**?
  - What happens to them when **`f`** is recursively called?

Life time

- Let x be a **static** variable declared inside a scope S.
  - The life time of x begins when program execution starts, and ends when program execution terminates.
  - Even when the binding of x is not visible, x remains a live variable.

C++

```
int f() {
  static int die_hard;        // static variable, as it says
  int temp;                   // automatic variable
  … f() …
}
main() {
   … f() …
}
```

  - What are the life time and scope of **`die_hard`**?
    - lifetime: during main(), scope: inside f()
  - What are the life time and scope of **`temp`**?
    - lifetime: during f(), scope: inside f()
  - What happens to them when **`f`** is recursively called?
    - `die_hard`: only one `die_hard` is used
    - `temp`: different `temp`s are born for each recursive call