

# CS4450: Principles of Programming Languages

`let` & `let*` – aka “local binding”

Dr William Harrison



# let and let\* in Scheme

```
(let () (+ x y))  
(let ((x 1)) (+ x y))  
(let ((x 1) (y 8)) (+ x y))  
(let ((x 1) (y 8) (z 5)) (+ x y))
```

zero or more bindings, no repeats

```
(let* () (+ x y))  
(let* ((x 1)) (+ x y))  
(let* ((x 1) (y 8)) (+ x y))  
(let* ((x 1) (y 8) (z 5)) (+ x y))
```

zero or more bindings, repeats allowed



## Let binding: when in doubt...

```
> (let ((x 1)) (let ((x 7) (y x)) y))  
1  
> (let* ((x 1)) (let ((x 7) (y x)) y))  
1  
> (let ((x 7) (y x)) y)
```

Error: variable x is not bound.

Type (debug) to enter the debugger.

```
> (let* ((x 7) (y x)) y)  
7  
>
```



# Let binding: when in doubt...

```
> (let ((x 1)) (let ((x 7) (y x)) y))  
1  
> (let* ((x 1)) (let ((x 7) (y x)) y))  
1  
> (let ((x 7) (y x)) y)
```

Error: variable x is not bound.  
Type (debug) to enter the debugger.

```
> (let* ((x 7) (y x)) y)  
7  
>
```



evaluating `(let ((x 7) (y x)) y)`

`(let ((x 7) (y x)) y)`

no value for "x"

"x" has a binding

`(let ((x 1)) (let ((x 7) (y x)) y))`



# What is “**let**” for?

```
(let
  (( $v_1$   $e_1$ )
   ...
   ( $v_n$   $e_n$ ))
  body)
```

- Introduces local bindings for  $v_1 \dots v_n$  that can be used within **body**
- ...each  $v_i$  is bound to the corresponding value of  $e_i$
- ...previous bindings for  $v_1 \dots v_n$  don't apply within **body**



# Local binding “as it occurs in nature”

```
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i]=t[i]) != '\0') i++;
}
```

*inner part of **strcpy** is  
a form of let-binding*

```
(let
  ((i ?))
  (i=0; while ...)
)
```



# As a Haskell data type

productions for let's

```
Exp ::= ( let ( {(Name Exp)}*) Exp)
Exp ::= ( let* ( {(Name Exp)}*) Exp)
Exp ::= ( letrec ( {(Name Exp)}*) Exp)
```

```
data Exp = ...
  | Letexp [(String,Exp)] Exp
  | Letstar [(String,Exp)] Exp
  | Letrec [(String,Exp)] Exp
```

*\* checks for “no repeats” in let performed in the front end rather rather than encoded in the grammar*



# Step-by-step evaluation of let binding

```
(let
  (( $v_1$   $e_1$ )
   ...
   ( $v_n$   $e_n$ ))
  body)
```

- Assume we are evaluating in environment **env**
- Evaluate each of  $e_1 \dots e_n$  in **env**
- Produces a **Value**  $val_i$  is for each  $e_i$
- Extend **env** to **env'** with bindings  $v_i = val_i$
- evaluate **body** in **env'**



# What's going on here?

```
> (let ((x 7) (y x)) y)
```

```
Error: variable x is not bound.  
Type (debug) to enter the debugger.
```

DrScheme session



# Definition explains program behavior

```
> (let ((x 7) (y x)) y)
```

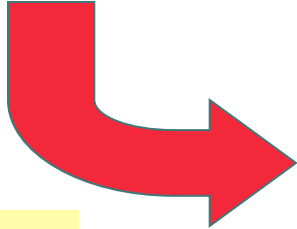
```
Error: variable x is not bound.  
Type (debug) to enter the debugger.
```

```
eval (Var "x") emptyenv  
= applyenv emptyenv "x"  
= lookup "x" []  
= error!
```

● ● ● | **let\***

```
(let*  
  ((v1 e1)  
   (v2 e2))  
  body)
```

```
(let  
  ((v1 e1))  
  (let  
    ((v2 e2))  
    body)  
)
```



*can be translated  
as standard **let***

*i.e., **let\*** is syntactic sugar*



# Step-by-step evaluation of `let*` binding

```
(let*  
  ((v1 e1)  
   ...  
   (vn en))  
  body)
```

- Assume we are evaluating in environment `env`
- Evaluate  $e_1$  in `env`
  - Produces `Value`  $val_1$
- Extend `env` to `env1` with bindings  $v_1=val_1$
- Now, process  $(v_2 e_2)$  in `env1`
  - repeat; produces new env `envn`
- evaluate `body` in `envn`



# Question

What should the following be?

```
(let* ((x 3) (y x) (x 1)) (+ x y))
```



# Recursive bindings in Scheme

```
(letrec
  ((fac (lambda (n)
          (if (= n 0) 1 (* n (fac (- n 1)))))))
  (fac 3))
```

...produces 6

```
(let
  ((fac (lambda (n)
          (if (= n 0) 1 (* n (fac (- n 1)))))))
  (fac 3))
```

...produces a runtime error