

Python:

```
x = 3  
return x + (10 * x)
```

Racket:

```
(let ((x 3))  
  (+ x (* 10 x)))
```

.

$x += 5$  and  $x++$   
are

**syntactic abstractions**

of

$x = y + z$

.

**convenient  
but  
not necessary**

▪

In Racket,

**cond**

is a syntactic abstraction of

**if**

▪

In Racket,

**let**

is a syntactic abstraction of

**applying a function**

.

[ list comprehensions in Python ]

▪

[ Local variables bind a value to a name. ]

.

The syntax of Racket's let expression:

```
<let-expression> ::= (let <binding-list> <body>)  
  <binding-list> ::= ()  
                  | (<binding> . <binding-list>)  
  <binding> ::= (<var> <exp>)  
  <body> ::= <exp>
```

.



This:

```
(let ((<var_1> <exp_1>)
      (<var_2> <exp_2>)
      .
      .
      .
      (<var_n> <exp_n>))
  <body>)
```

is equivalent to:

```
((lambda (<var_1> <var_2>...<var_n>)
  <body>)
  <exp_1> <exp_2>... <exp_n>)
```

.

```
(let ((op (first exp))
      (arg1 (second exp))
      (arg2 (third exp)))
    (list arg1 op arg2))
```

is equivalent to:

```
((lambda (op arg1 arg2)
  (list arg1 op arg2))
 (first exp) (second exp) (third exp))
```

.

```
(let ((op (first exp))
      (arg1 (second exp))
      (arg2 (third exp)))
    (list arg1 op arg2))
```

is equivalent to:

```
((lambda (op arg1 arg2)
  (list arg1 op arg2))
 (first exp) (second exp) (third exp))
```

.

```
(let ((op (first exp))
      (arg1 (second exp))
      (arg2 (third exp)))
    (list arg1 op arg2))
```

is equivalent to:

```
((lambda (op arg1 arg2)
  (list arg1 op arg2))
 (first exp) (second exp) (third exp))
```

.

```
(let ((op (first exp))
      (arg1 (second exp))
      (arg2 (third exp)))
  (list arg1 op arg2))
```

is equivalent to:

```
((lambda (op arg1 arg2)
  (list arg1 op arg2))
 (first exp) (second exp) (third exp))
```

.

# **translational semantics**

▪

[ images showing compilation with and without preprocess ]

▪