What is a List Comprehension?

Each language has a different syntax for it.
So the syntax isn't what's special.

Look at the WP description of mathematical set comprehension:

\[ S = \{ 2 \cdot x \mid x \in \mathbb{N}, x^2 > 3 \} \]

It identifies the meaningful components (the four named ones), as we do in this course for any operation, and you'll do from now on to much more easily learn new languages, or new constructs in familiar languages.

The surface syntax parts "{", "|", "\in", "," "} are irrelevant except to indicate that we're doing set comprehension.

# In Python.

[ <element-exp> for <var-name> in <list-exp> if <boolean-exp>]

The "for", "in" and "if" are just to separate the components.
So [ 2*x for x in [0, 1, 2, 3] if x**2 > 3 ] is just a way to say:

A list comprehension where the element expression is "2\cdot x", the variable name is "x", the list expression is "[0, 1, 2, 3]", and the boolean expression is "x**2 > 3".

The Wikipedia article used to say (until a student corrected it in lecture!) "output function" instead of "output expression". But that was wrong, and you can see (run them!) the difference in:

[ 2*x for x in [0, 1, 2, 3] if x**2 > 3 ]

def double(x): return 2 * x

[ double for x in [0, 1, 2, 3] if x**2 > 3 ]

EXERCISE: Use `double` to get the meaning of the first one.
EXERCISE: Predict and run:

def big_enough(x): return x**2 > 3

[ 2*x for x in [0, 1, 2, 3] if big_enough ]

# If the syntax isn't the special aspect, what is?

In most PLs the user can't add an operation that combines the components and produces the desired result.
In Python the following is doomed, as discussed ad nauseum in Weeks 1 to 3:

```python
def list_comprehension(a, b, c, d):
    return "nobody can make this work"
```

```python
list_comprehension(2*x, x, [0, 1, 2, 3], x**2 > 3)
```

Run that if you're not ABSOLUTELY CERTAIN of all the errors in that line
(and that they're all independent of the body of `list_comprehension`). |#

### A List Comprehension is an operation combining:

An expression (for the elements), which can be open in a variable name.

A variable name (to refer to an element of the source list).

A source list value (in other words, the expression producing the list is
not treated specially, it is evaluated as usual and a list comprehension
only needs the run-time list value).

A boolean expression (an expression which when evaluated at run-time will
produce a boolean value), which can be open in the variable name. |#

### Building the List Comprehension Operation.

To focus on the three different kinds of components involved, we'll
make a version without the boolean expression. Let's name it `list/for`.

Syntactic Decisions:

1. Order, and possibly grouping (especially to remove any ambiguity)
   of the components.
2. Extra fixed surface syntax.

Suppose we're making this for a Formal Verification library, and that domain's
application developers are used to the mathematical set comprehension order,
and they like `:` and `e` (not just space) between the components. |#

### Implementation.

Step 1: example usage (test case!) and what it should transform into. |#

```racket
#:list/for (* 2 x) ; x ∈ /list 0 1 2 3)
#:map λ x (* 2 x) /list 0 1 2 3)
; EXERCISE: Predict the error:
#:map (* 2 x) /list 0 1 2 3)
```

### Step 2: note where the components went, and generalize.

Notice the variable name went into the function as it parameter name.

Use angle-brackets for parts that stand for pieces of the user's code.
In a part name, include whether it must be a name or an expression.
For expression parts that must return a certain type of value at run-time,
include that in the name. |#
#; list/for <element-expression-open-in-variable-name>
  : <variable-name> ∈ <list-expression>>
#; map λ <variable-name> <element-expression> <list-expression>>

(define-syntax
  (list/for <element-expression> : <variable-name> ∈ <list-expression>>)
  map λ <variable-name> <element-expression>
  <list-expression>>)

; EXERCISE: 1. Compile-time expand all uses in the rest of the file.
  2. Predict the run-time values.

(list/for (* 2 x) : x ∈ (list 0 1 2 3))
define x 10
define y 100
(list/for (* x y) : x ∈ (list 0 1 2 3))
(list/for (* x y) : y ∈ (list 0 1 2 3))

; This new kind of expression can be used anywhere an expression is allowed.
define l (list 0 1 2 3))
(list/for (list/for (list row column) : column ∈ l)
  : row ∈ (list/for (sqr j) : j ∈ l))

#| CS uses angle-brackets as a convention for parts not taken literally.
Does `define-syntax-rule` interpret that convention? No. |#
(list/for (expt 2 n) hello n there (list 0 1 2 3))

#| Slightly more powerful way to make new syntactic forms: |#
(define-syntax list/for*
  syntax-rules
    ; This group is for components that must be written literally.
    : e
    ; While we're at it, `id` is the PL term for names in code.
    (list/for* <element-expression-open-in-id>
      : <id> ∈ <list-expression>
      map λ <id> <element-expression-open-in-id>
      <list-expression>>))))

; EXERCISE: Predict the precise error, then uncomment and run.
#; list/for* expt 2 n: hello n there (list 0 1 2 3))

(list/for* 123 : a ∈ (list))

; In programming the difference between an expression and a function is vital.
; These produce different values:
(list/for* λ x: (* 2 x) : x ∈ (list 1 2 3))
(list/for* (* 2 x) : x ∈ (list 1 2 3))