expressions vs statements.

- value
- no side effects

no value
side effects

blurry

side effect - any change in the value of a variable that persists beyond execution of a statement.
side effects should have control.

expr. operators, operands.
  infix - C, Ada, C++, Java  3+4
  prefix - Lisp, Scheme (+ 3 4)
  postfix - Forth, PostScript 3 4 +
  HP-calculators.

rules for eval.
  most used: applicative order eval
  or strict eval
    operands are eval'd first
    then operators are applied.
  bottom-up eval of syntax tree:
    \[(3+4) \times (5-6)\]

? order of computation of sub-expressions?
can be left to right
may be deliberately unspecified
to promote efficiency.
can make a difference if there
are side effects. e.g. \((f(x)+4) \times (x-6)\)

short-circuit eval.
some expressions can be eval
without eval'ing all operands.

true or \(x\)
false and \(x\)
if \((i < \text{lastindex} \text{ and } a[i] \geq x)\)...
if \((i < \text{lastindex})\)
then if \((a[i] \geq x)\)...
order may matter.
if side effects are banned or
carefully controlled, we have an
important property:

any 2 expr. in a program
that have the same value
may be substituted for each
other anywhere in the program.

substitution rule or
 referential transparency

allows for a form of delayed eval called
normal order evaluation. Informally,
each operation begins eval before its
operands are eval’d, and each operand
is eval’d only if needed.

example p. 268.

```c
int double (int x) { return x+x; }
int square (int x) { return x*x; }
```

```c
square (double(2));
```
applicative order

\[ \text{square}(\text{double}(2)) \rightarrow \text{square}(2+2) \]
\[ \rightarrow \text{square}(4) \rightarrow 4 \times 4 \rightarrow 16 \]

normal order - bodies of \( f \)s are substituted before eval.

\[ \rightarrow \text{double}(2) \times \text{double}(2) \rightarrow \]
\[ (2+2) \times (2+2) \rightarrow 4 \times 4 \rightarrow 16 \]
square
1
double