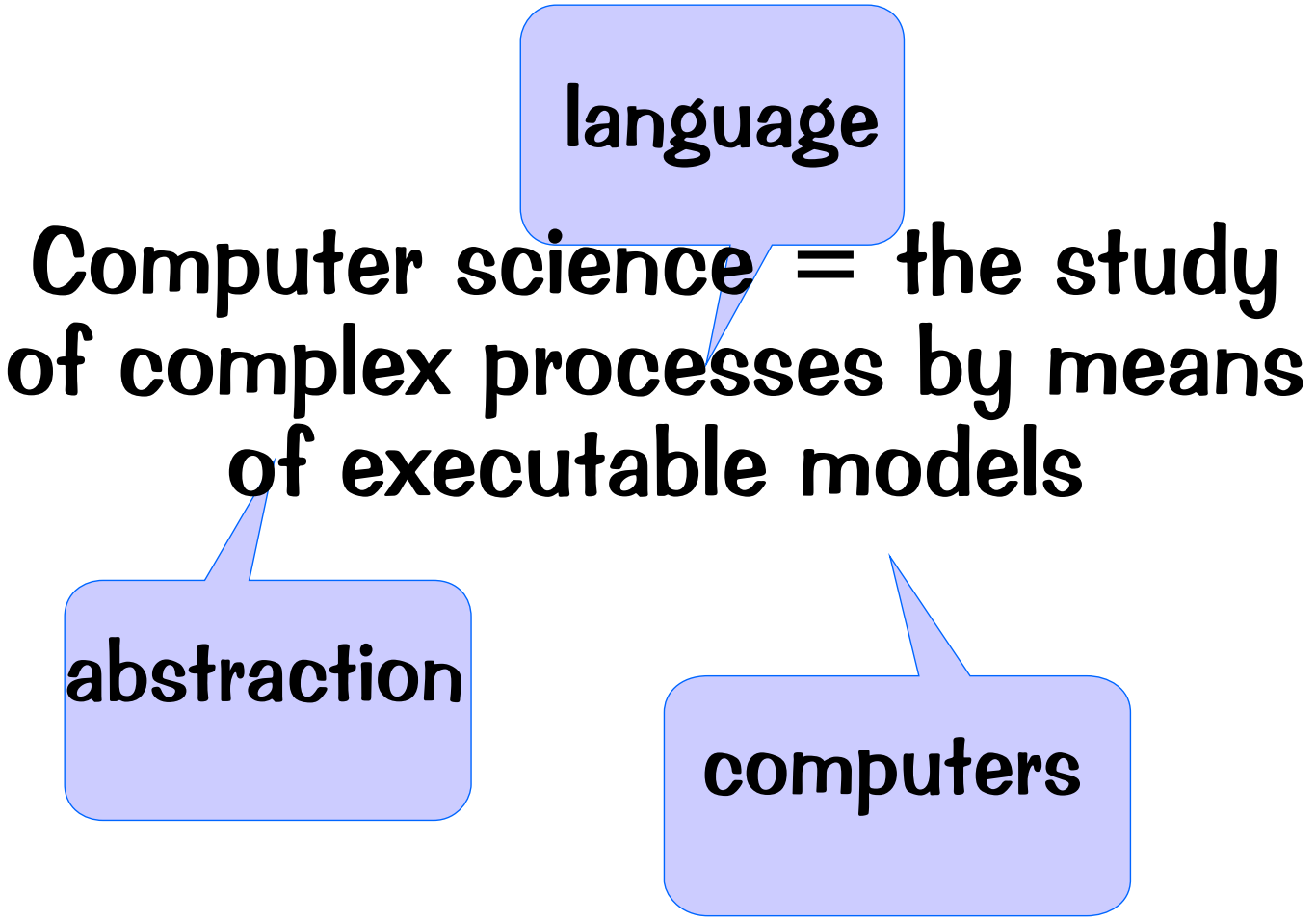
1. Informal Pico

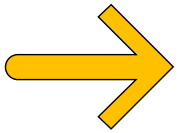
- processes
- constants
- variables
- functions
- examples


language
Computer science = the study
of complex processes by means
of executable models
abstraction
computers

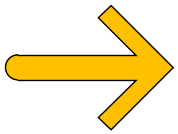
Processes...

- possess a *behaviour* ...
- recognise *time* ...
- are specified by a *language* ...

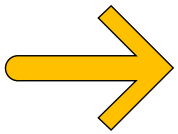
Executable processes...



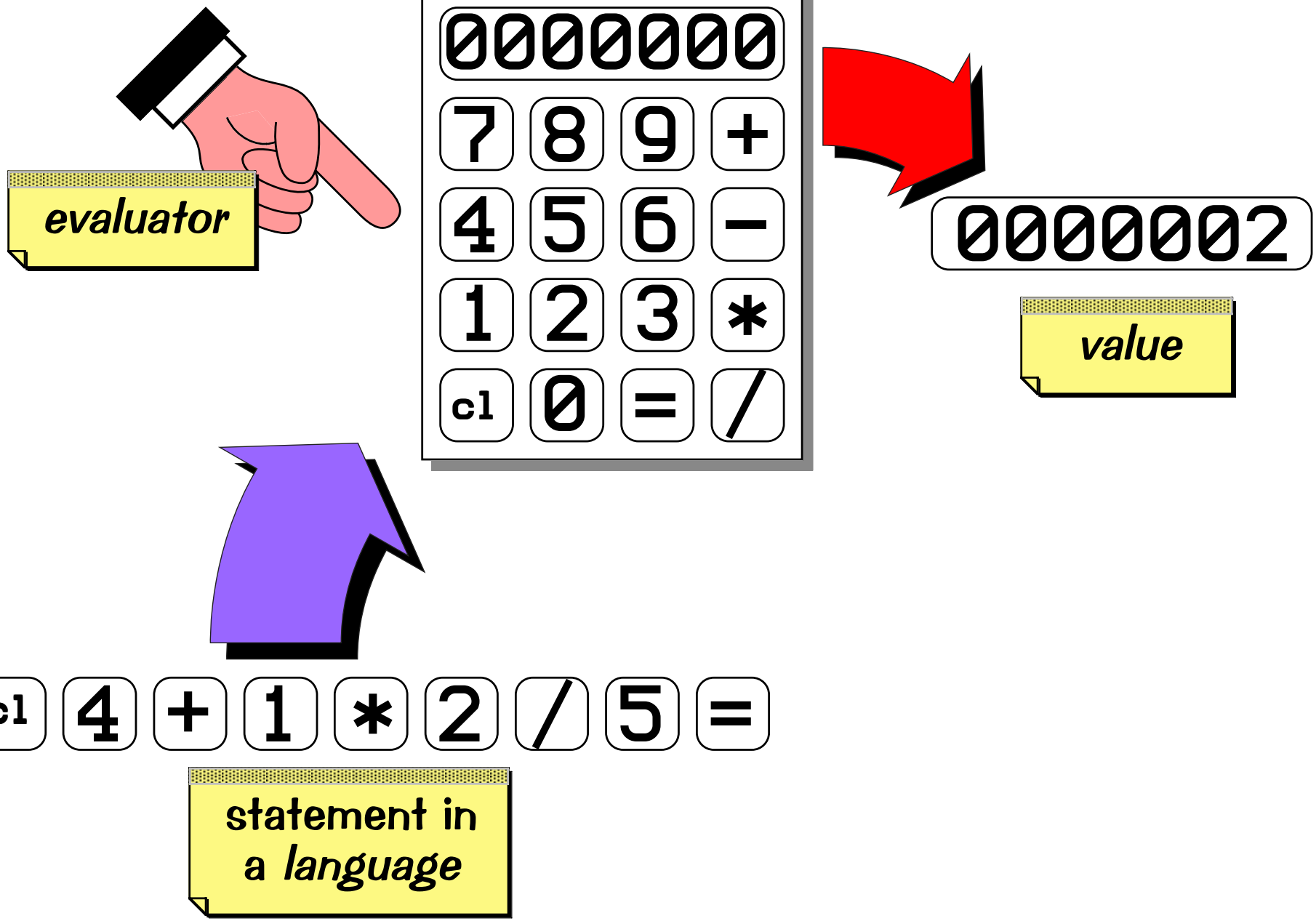
are *specified* in a language ...



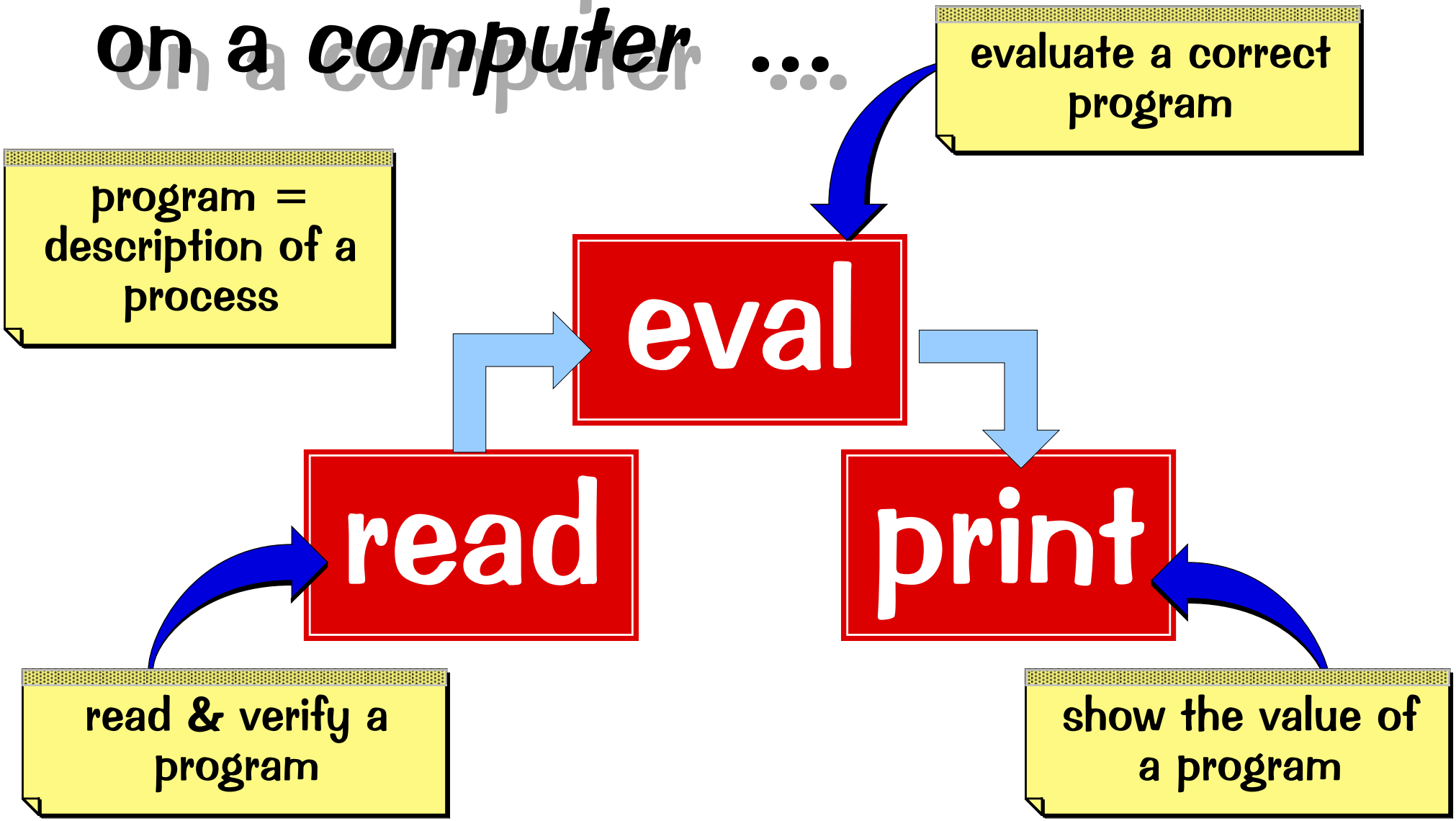
require an *evaluator* ...



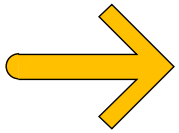
produce a *value* ...

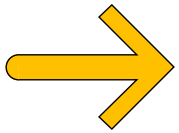

evaluator
0000000
7 8 9 +
4 5 6 −
1 2 3 *
cl 0 = /
0000002
value
cl 4 + 1 * 2 / 5 =
statement in
a language

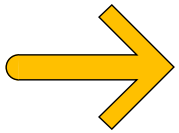
Executable process on a *computer* ...



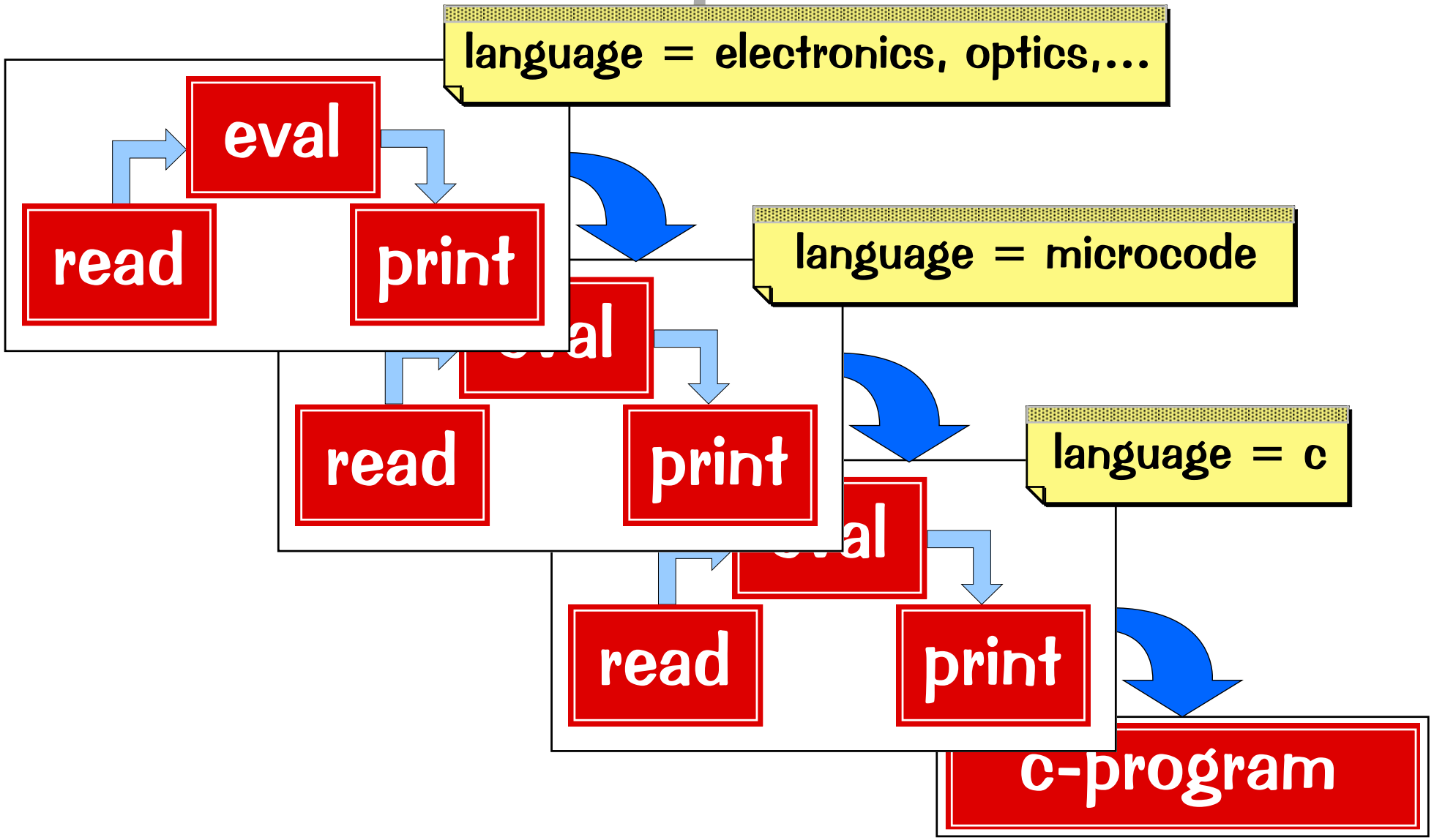
A computer ...


... is a process that evaluates processes

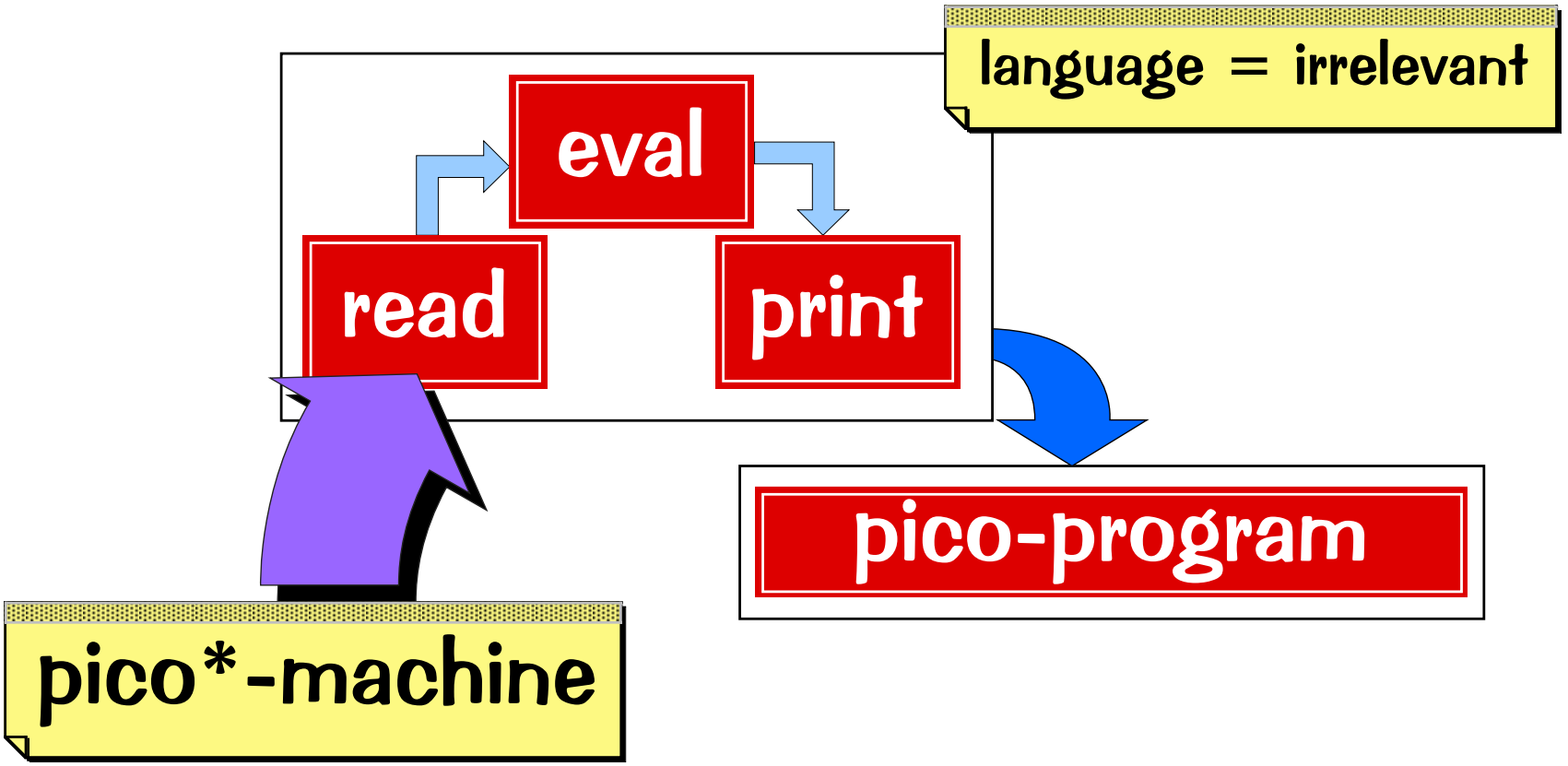

... is described by a program


... build one = write a program

for a PC this implies...



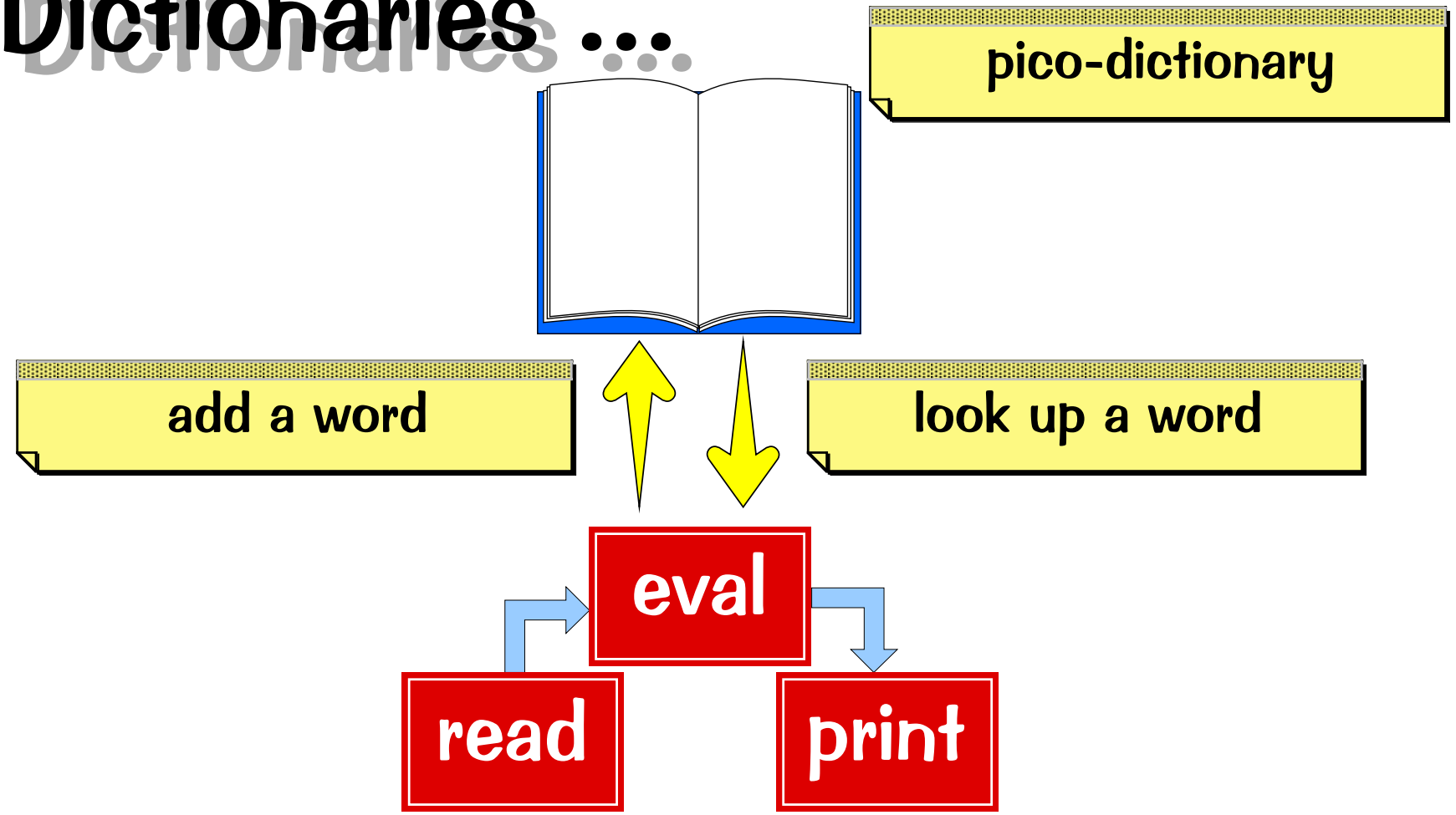
A simple computer model ...



*pico = 10^{-12} = *very* small

- the pico-machine evaluates pico-processes
- pico-processes are specified in pi

Dictionaries ...

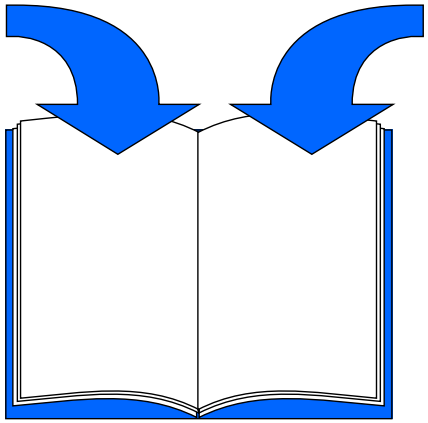


the pico-evaluator manages a dictionary wherein words are associated with values

Structure of the dictionary ...

words

values



example:

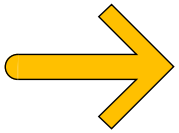
```
Prime
*
123
4.75
'monday'
```

word = variable

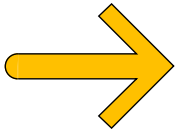
word = constant

example:

```
123
4.75
'monday'
[1,2,3]
<function f>
```



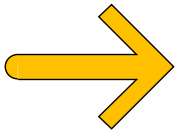
Initially the dictionary is filled with *constants* = names with an immutable value



There are *numerical* and *textual* constants

```
name = 123
       12.345
```

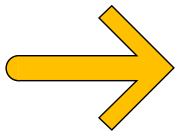
```
name = 'abracadabra'
```



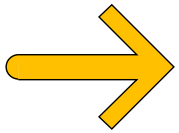
The dictionary can be boundlessly extended with *variables* = words with a mutable value

Creating a new name ...

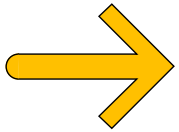
name : *expression*



the *name* is appended to the dictionary



the *expression* must be a well-formed pico-specification



the value of the *expression* is bound to the name

Changing the value of a name ...

name := *expression*

- the *name* must have been created previously
- the *expression* must be a well-formed pico-specification
- the value of the *expression* is bound to the name

Examples ...

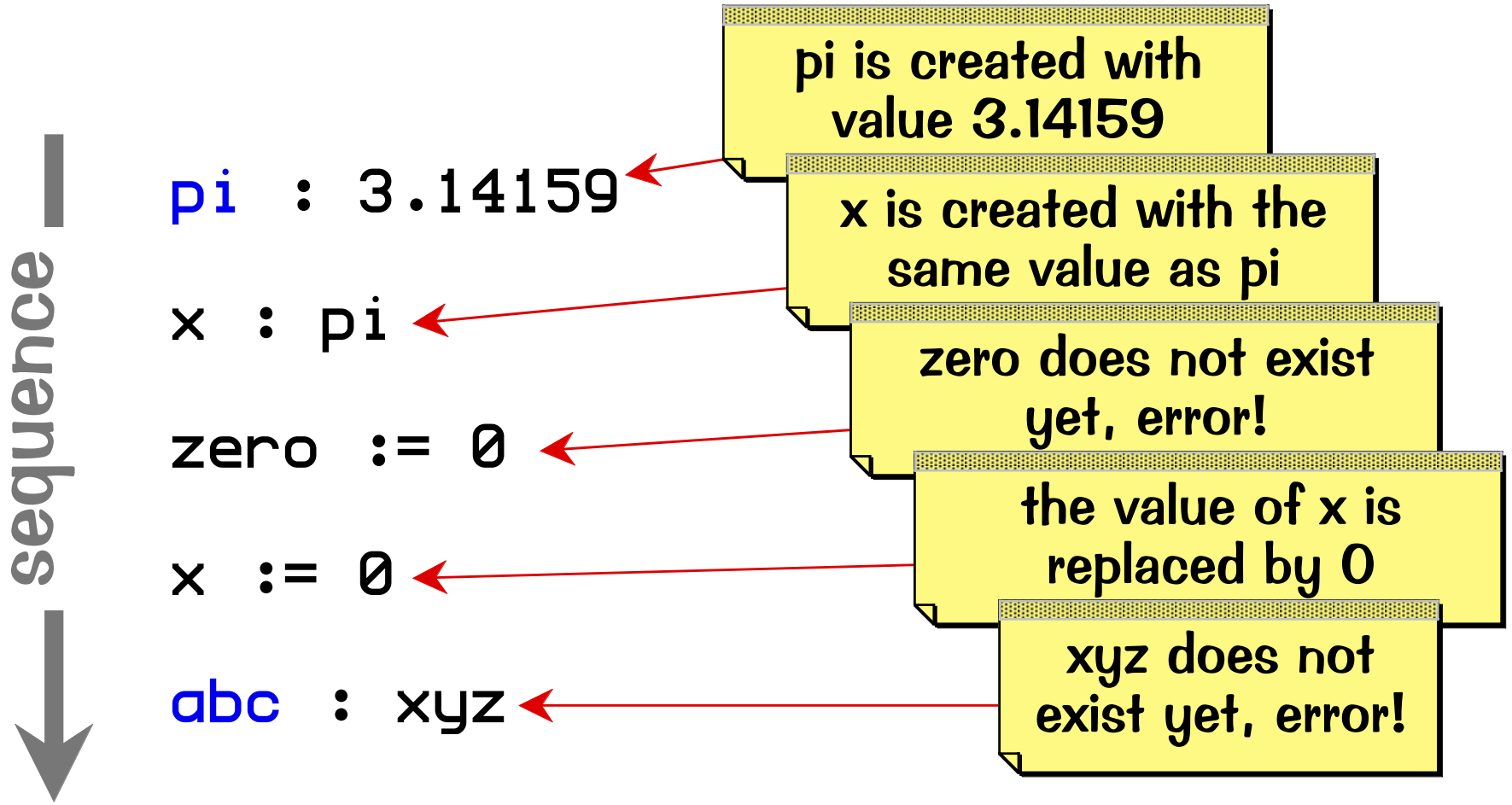

sequence
pi : 3.14159
x : pi
zero := 0
x := 0
abc : xyz
pi is created with value 3.14159
x is created with the same value as pi
zero does not exist yet, error!
the value of x is replaced by 0
xyz does not exist yet, error!

Table and function values...

in addition to numbers and text there are also *tables* and *functions*

- *textual* values = values of textual constants
- *numerical values* = values of numerical constants
- *tables*: are discussed later on
- *functions*: are discussed now

Functions are program fragments ...

name(*expression*$_1$, *expression*$_2$, ...)

- functions are bound to a *name*
- functions can be invoked
- when invoked, the value bound to the *name* determines what happens
- the *expressions* determine the values that the function is applied to

Operators ....

When the function name is an *operator* and the number of expressions does not exceed 2, arithmetic notation is allowed

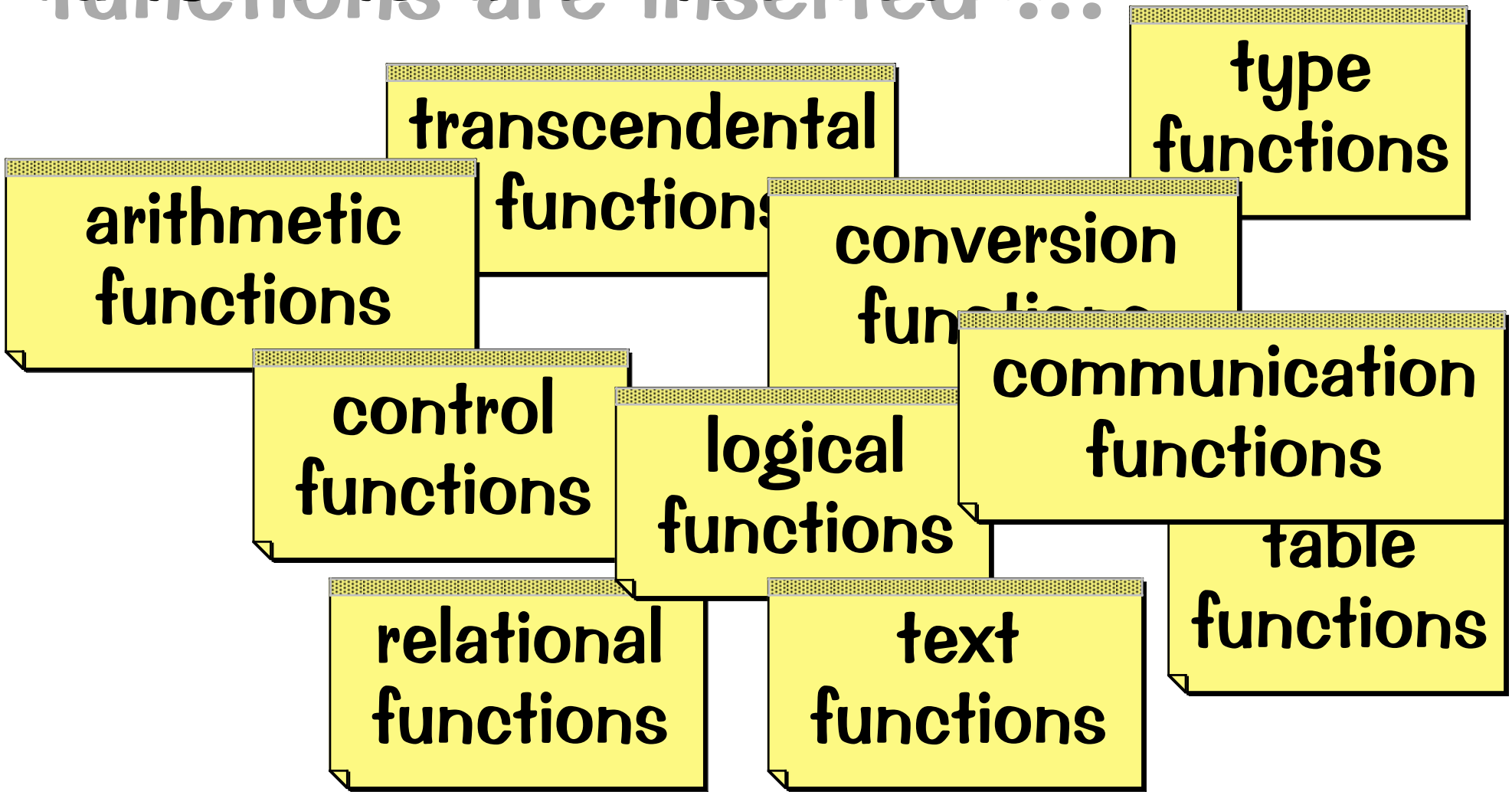
`+(1,2)`	⇔	`1+2`
`-(3.45)`	⇔	`-3.45`
`+(x,*(2,y))`	⇔	`x+2*y`
`*(x,+(2,y))`	⇔	`x*(2+y)`

binary notation

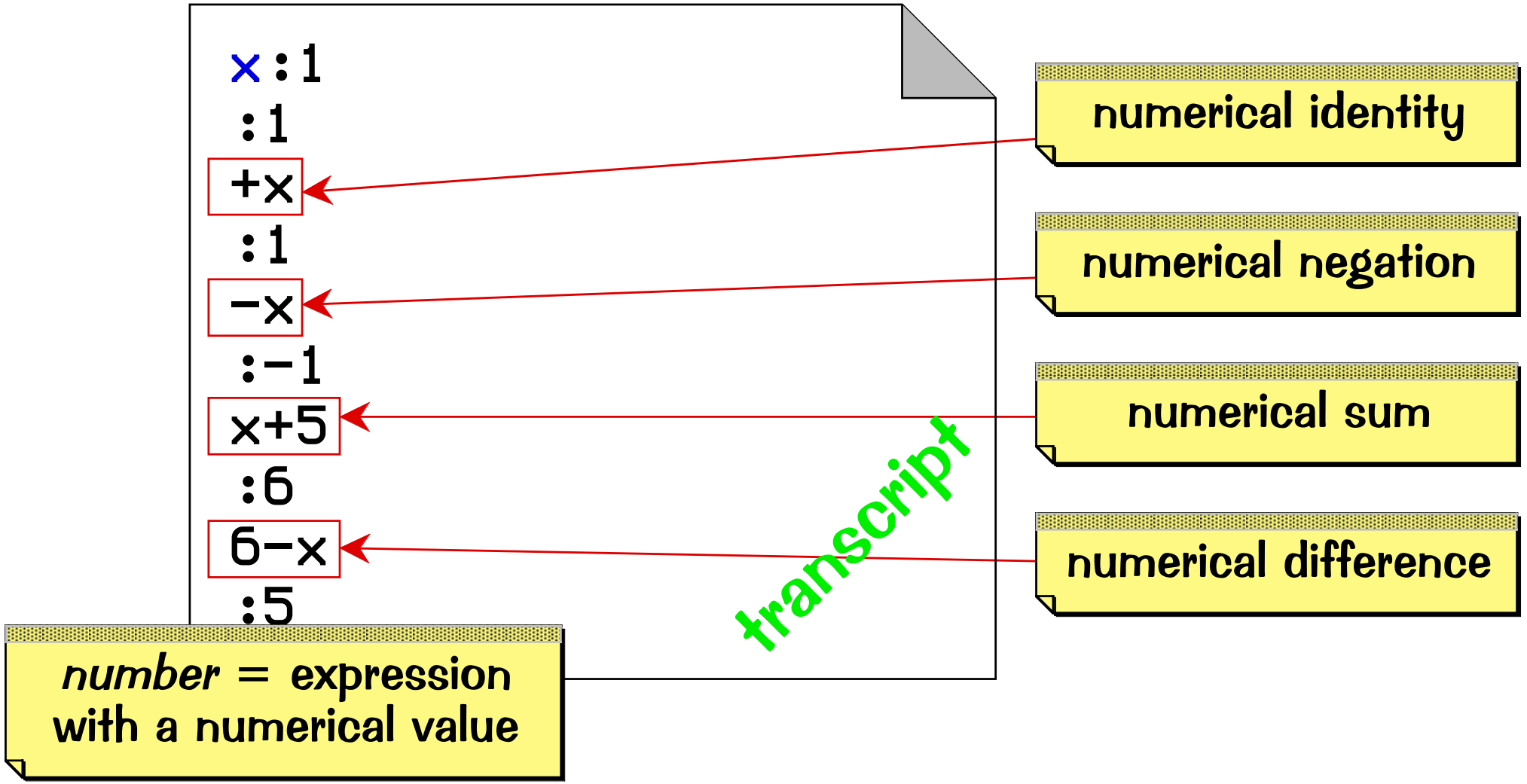
unary notation

usual precedence rules

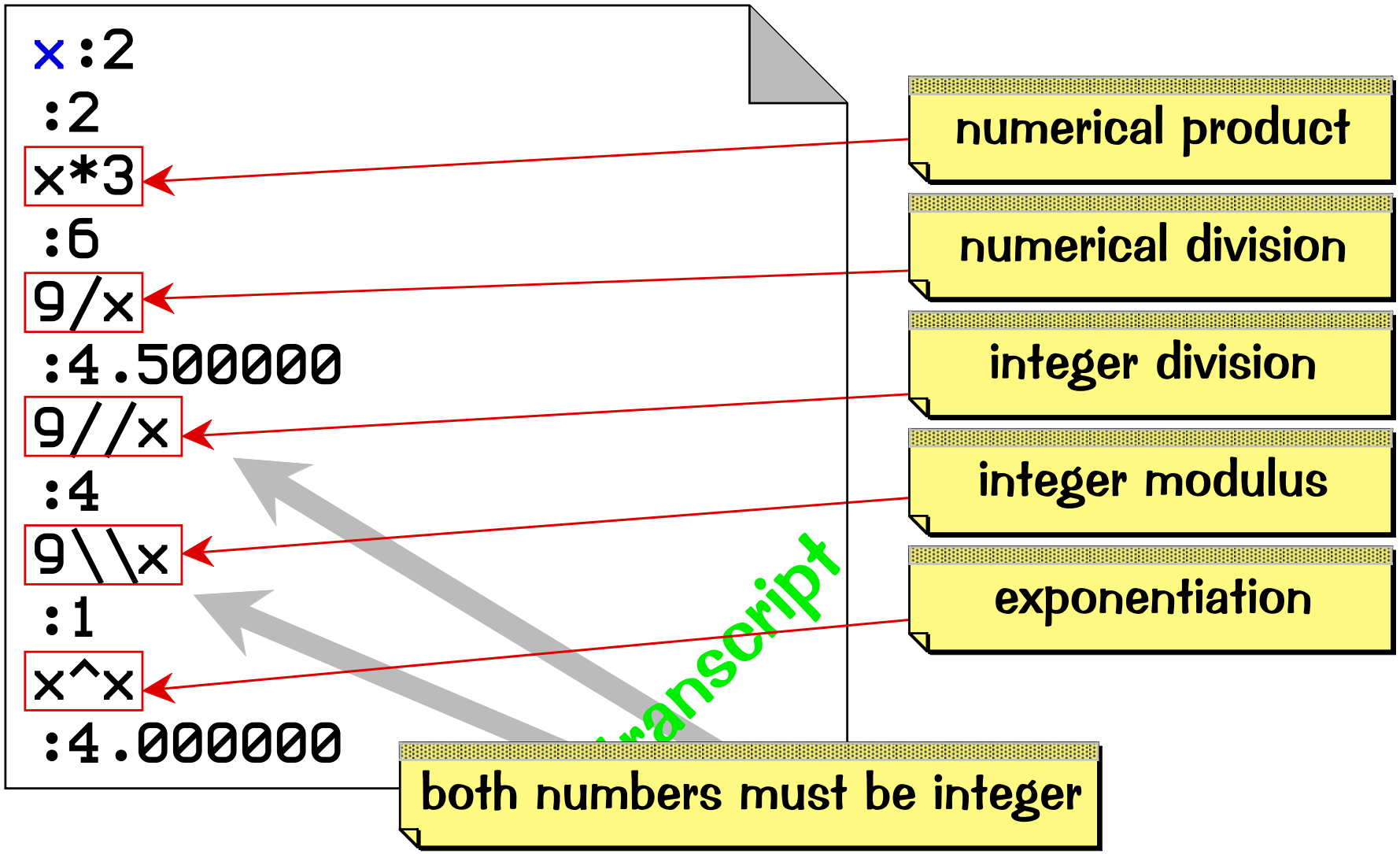
During startup, predefined functions are inserted...



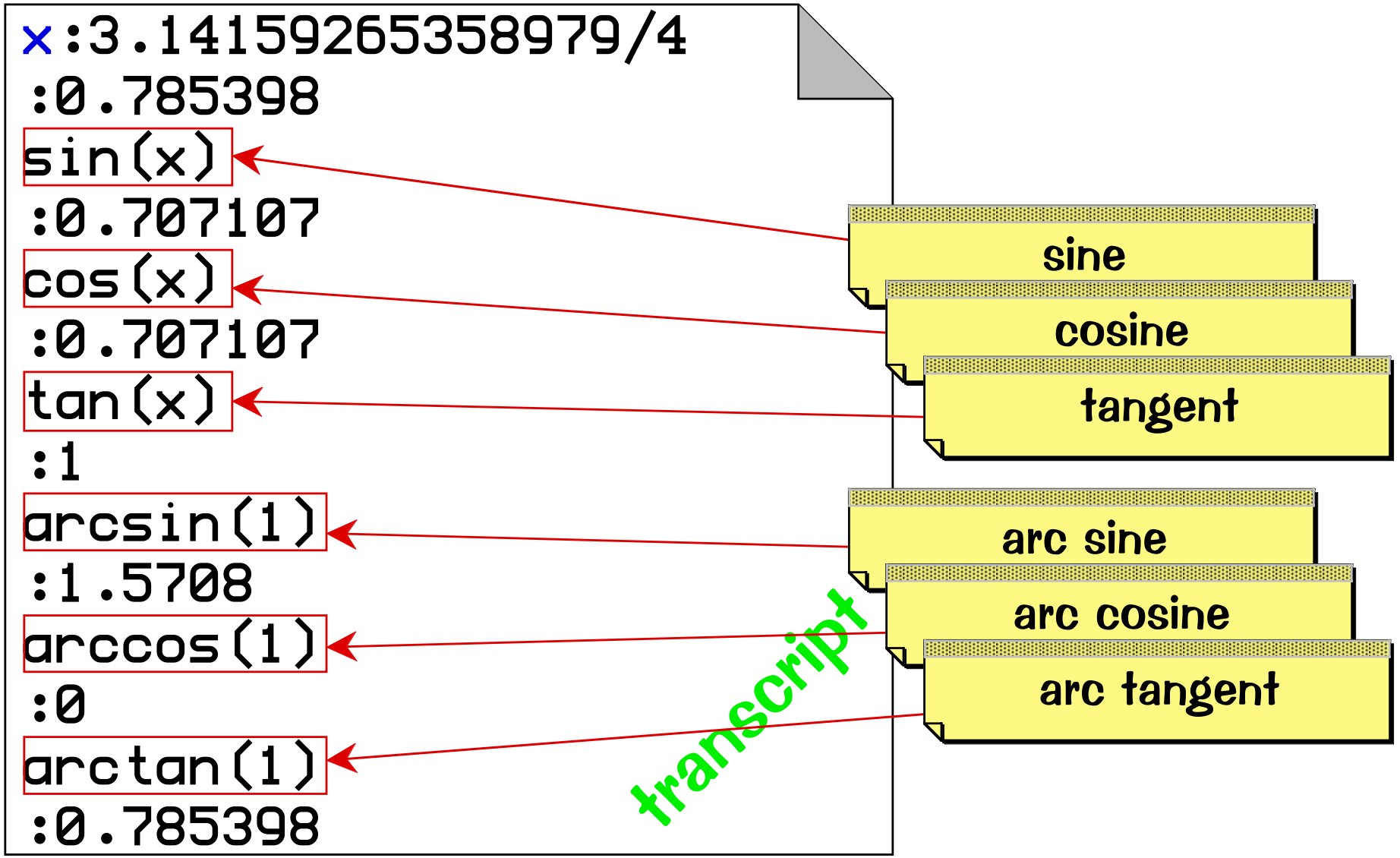
Arithmetic functions ...



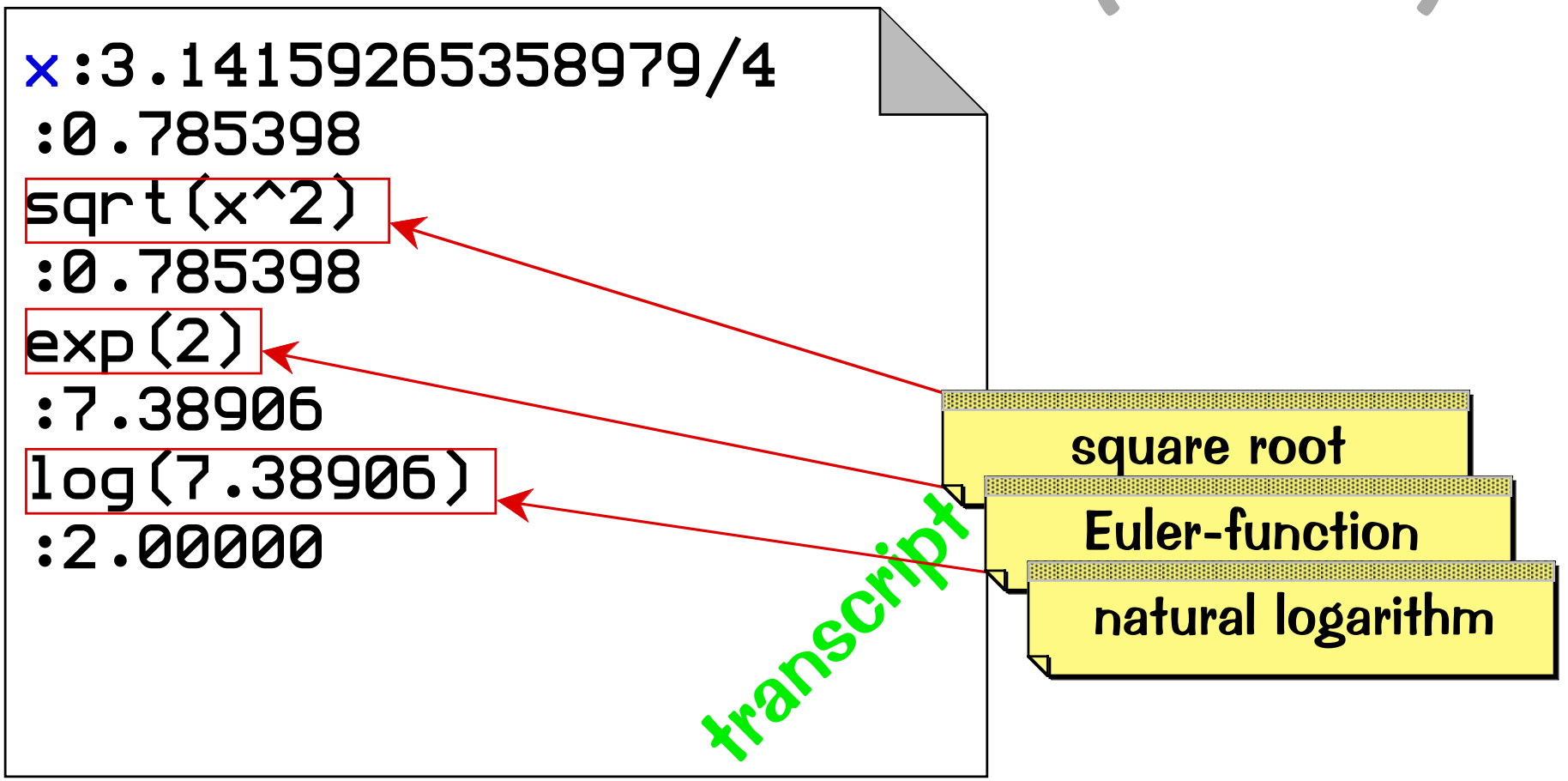
Arithmetic functions (cont'd) ...



Transcendental functions ...



Transcendental functions (cont'd) ...



Logical functions ...

```
true
:<native function true>
false
:<native function false>
true(display('ok'),display('ko'))
:ok
false(display('ok'),display('ko'))
:ko
```

transcript

true(u_1,u_2) ⇒ u_1

false(u_1,u_2) ⇒ u_2

Logical functions(cont'd) ...

```
and(true,true)
:<native function true>
and(true,false)
:<native function false>
and(false,true)
:<native function false>
and(false,false)
:<native function false>
```

transcript

and(U_1,U_2) ⇒ U_1(U_2,false)

```
or(true,true)
:<native function true>
or(true,false)
:<native function true>
or(false,true)
:<native function true>
or(false,false)
:<native function false>
```

transcript

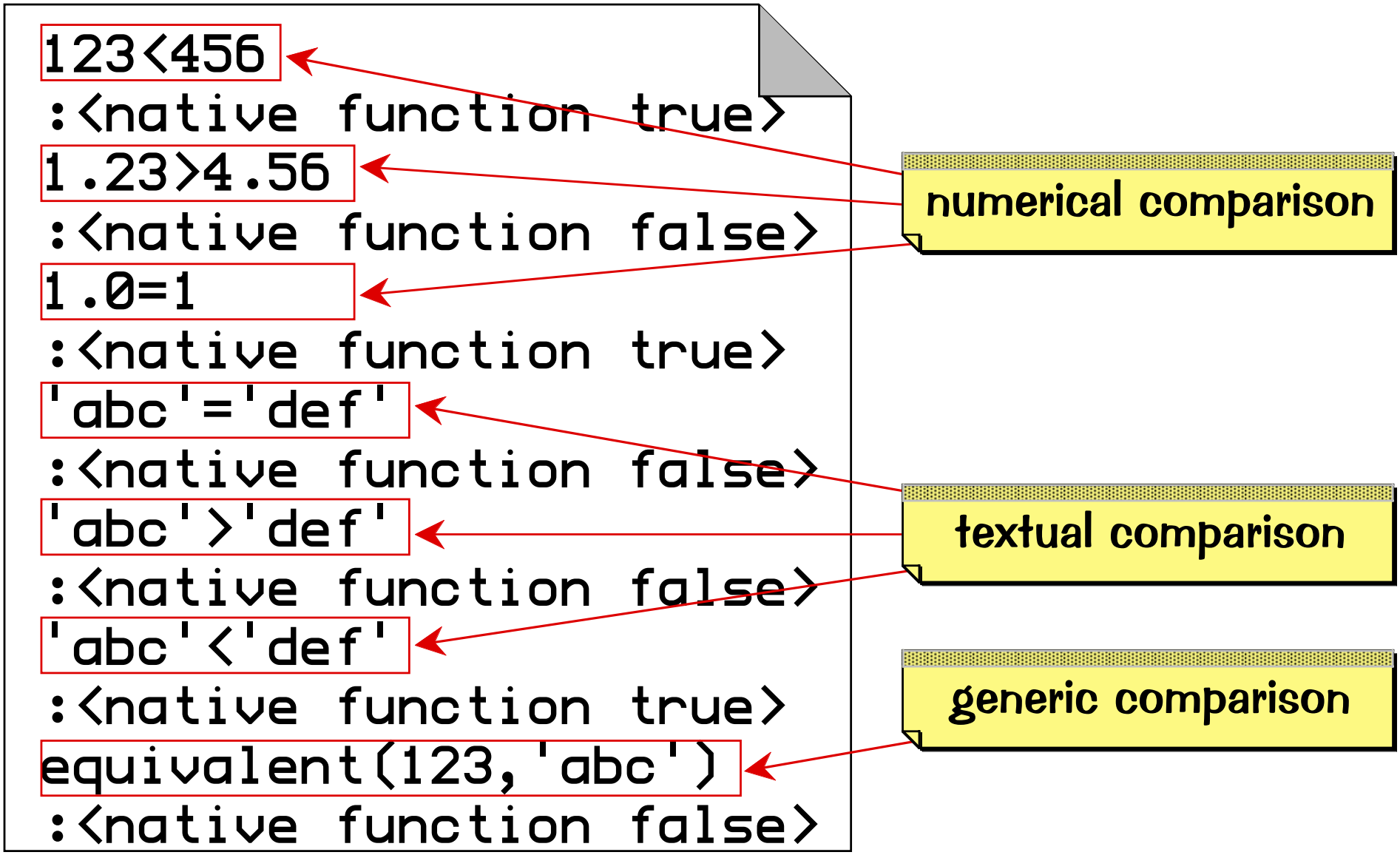
```
not(true)
:<native function false>
not(false)
:<native function true>
```

transcript

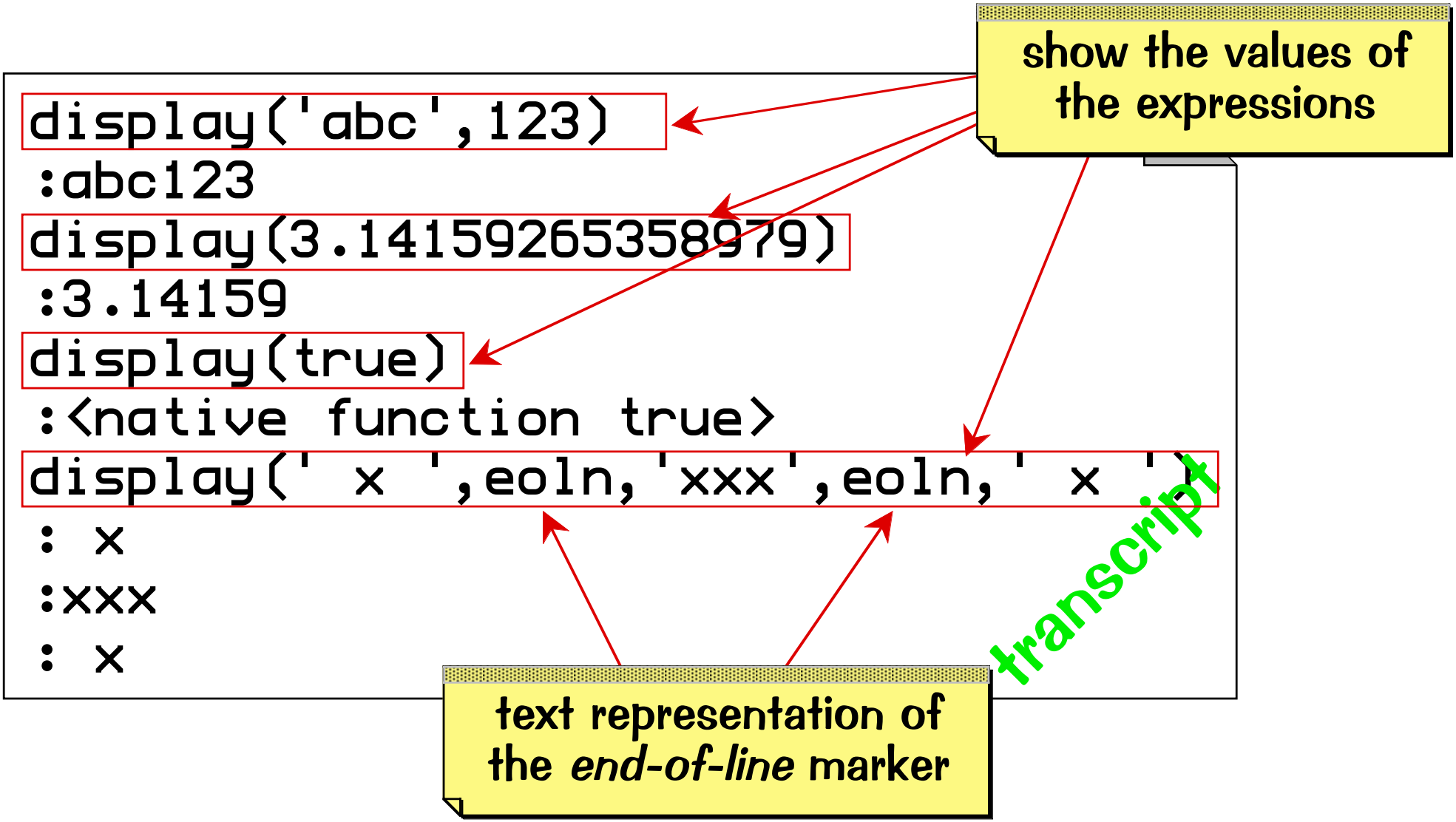
not(U) ⇒ U(false,true)

or(U_1,U_2) ⇒ U_1(true,U_2)

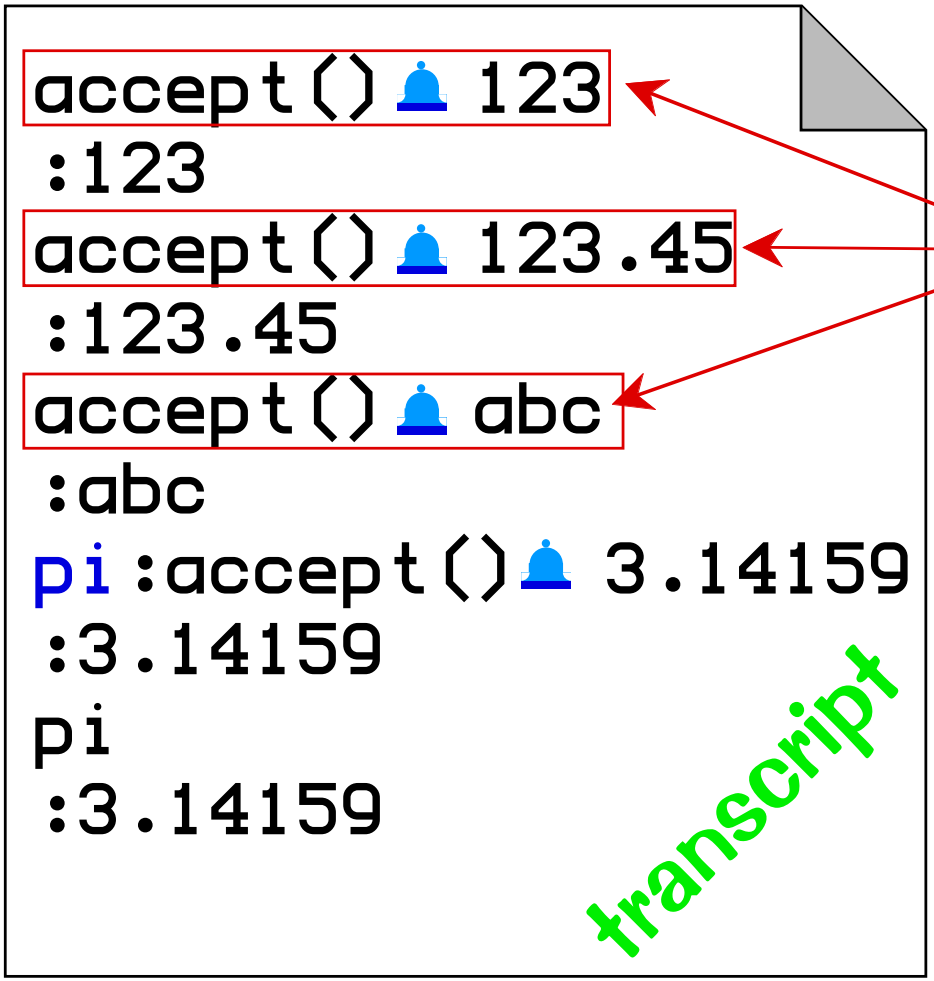
Relational functions ...



Communication functions ...



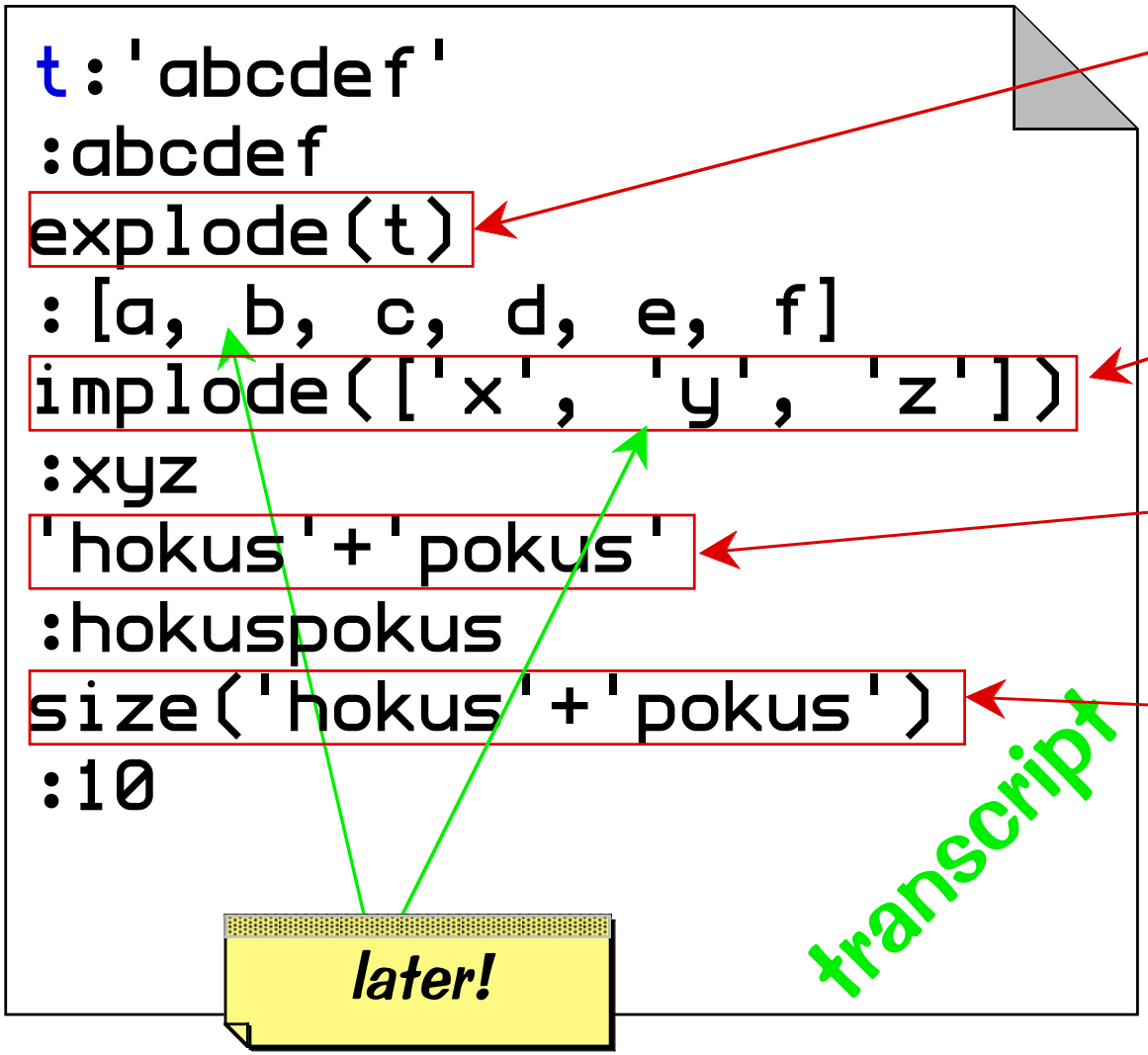
Communication functions (cont'd)...



construct text out of the entered characters

= *beep*

Text functions ...



separate into characters

join characters

join text

text size

Control functions: begin ...

```
begin(display('name?'),
      name: accept(),
      eoln+'You are '+name)
:name? Napoleon
:You are Napoleon
{display('name?');
 name: accept();
 eoln+'You are '+name}
:name? Bonaparte
:You are Bonaparte
```

arbitrary expressions

evaluate all expressions

return the value of the last expression

syntactically equivalent:
begin(a,b,...,z) ≡ {a;b;...;z}

Control functions: if ...

```
x: 123
:123
y:456
:456
if(x>y,x,y)
:456
if(x>y,x-y,y-x)
:333
if(x\\2=0,display('even'),display('odd'))
:odd
display(if(x\\2=0,'even','odd'))
:odd
```

if(*cond*,*then*,*else*)
⇒
cond(*then*,*else*)

transcript

Control functions: loops...

```
k:1000
:1000
log2:0
:0
while((k:=k//2)>0,log2:=log2+1)
:9
```

transcript

```
{p:1;q:1;r:0}
:0
until(p>100000000,{r:=p+q;q:=p;p:=r})
:102334155
p/q
:1.61803
```

transcript

will be discussed later

```
x:1
:1
for(n:1,n:=n+1,not(n>10),x:=3*x)
:59049
3^10
:59049
```

transcript

Table functions...

creating a table

```
t:tab('a','e','i','o','u')
:[a, e, i, o, u]
z:accept()
:o
for(i:1,i:=i+1,not(i>size(t)),
    if(t[i]=z,
       display(z,' is a vowel'),
       eoln))
:o is a vowel
:
t:=['a','e','i','o','u']
:[a, e, i, o, u]
```

will be discussed later

size of a table

syntactically equivalent:
tab(a,b,...,z) ≡ [a,b,...,z]

transcript

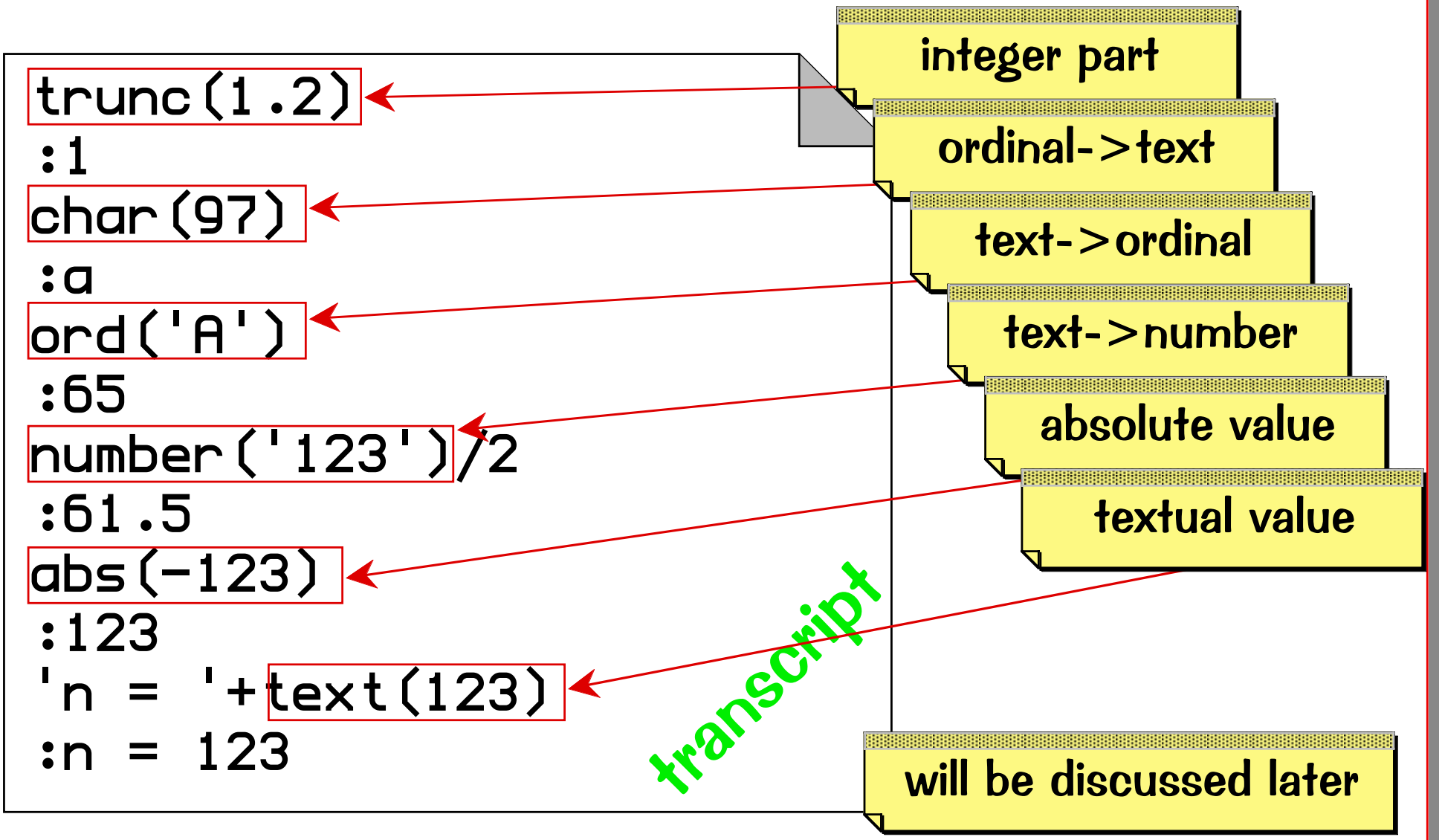
Type functions ...

```
is_number(1)
:<native function true>
is_fraction(2.4)
:<native function true>
is_text('abc')
:<native function true>
is_table([1,2,3])
:<native function true>
is_function(+)
:<native function true>
is_void(void)
:<native function true>
```

will be discussed later

transcript

Conversion functions ...



Own functions ...

name(*name*$_1$, *name*$_2$, ...) : *expression*

the *arguments** of the function

the *meaning* of the function

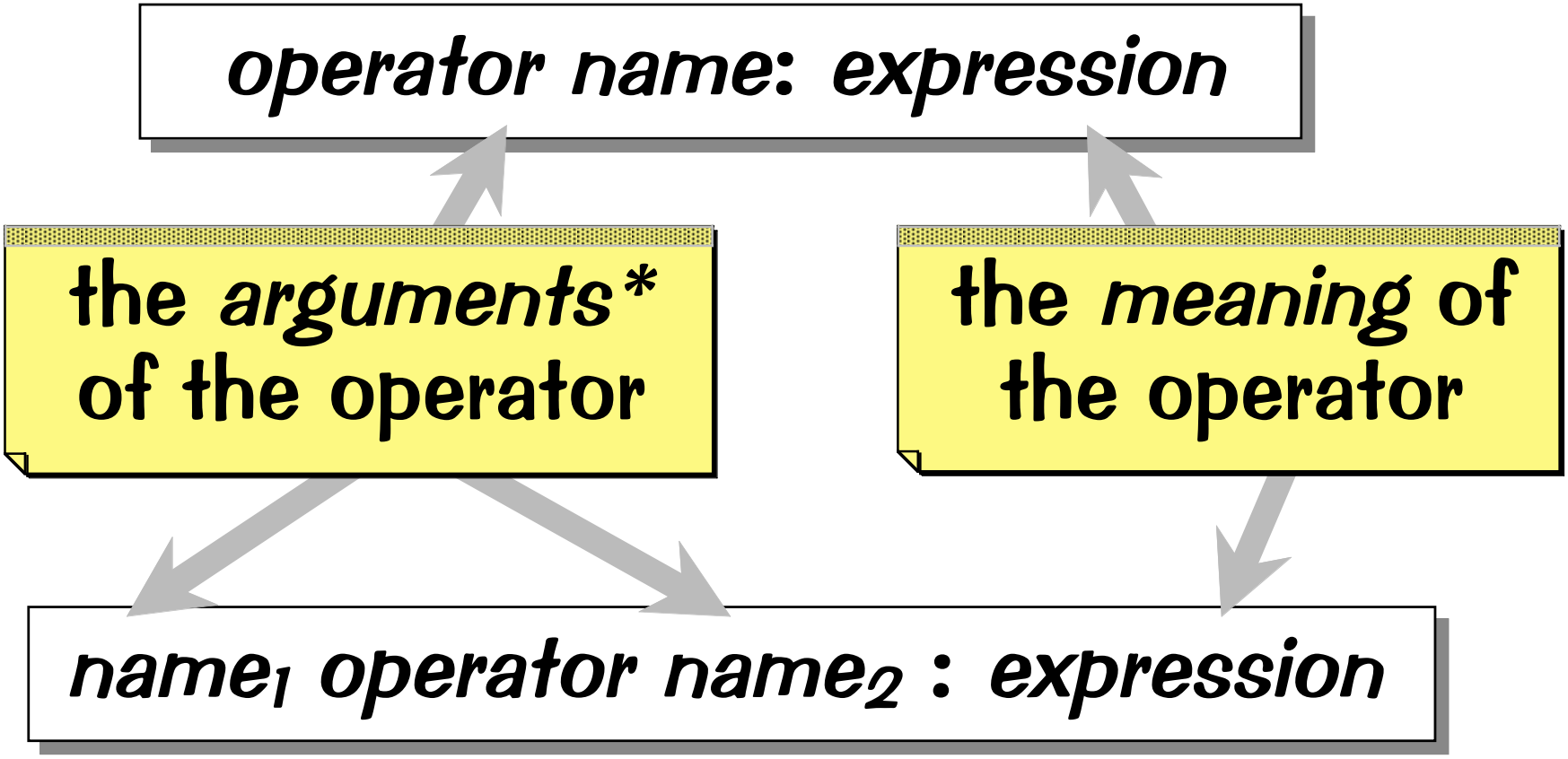
*limited to *name* until further notice

Own functions(cont'd) ...

```
average(a,b): (a+b)/2
:<function average>
average(1,2)
:1.500000
maximum(x,y): if(x>y,x,y)
:<function maximum>
maximum(4,5)
:5
maximum(average(1,5),average(2,3))
:3.000000
```

transcript

Own operators ...



*limited to *name* until further notice

Own operators(cont'd) ...

```
\p: not(p)
:<function \>
p&q: and(p,q)
:<function &>
p|q: or(p,q)
:<function |>
{a:1;b:2;c:3}
:3
\(a=b)&(b<c)
:<native function true>
```

transcript

Example 1: the greatest common divisor ...

fundamental property: "if g is the greatest common divisor of p and q, then g is also the greatest common divisor of the minimum of p and q on one hand, and of the difference of the maximum and the minimum of p and q on the other hand"*

$$gcd(p,q) = \begin{cases} gcd(p-q,q) & \text{if } p>q>0 \\ gcd(p,q-p) & \text{if } q>p>0 \\ p & \text{if } p=q>0 \end{cases}$$

* this is not an optimal algorithm...

Example 1 (cont'd): the greatest common divisor ...

```
a:121
:121
b:33
:33
gcd(x,y):if(x>y,gcd(x-y,y),
                    if(x<y,gcd(x,y-x),
                           x))
:<function gcd>
gcd(a,b)
:11
```

transcript

Example 2: factorials ...

fundamental property: "if N is the factorial of n, then N is the product of n and the factorial of n-1"

$$fac(n) = \begin{cases} n*fac(n-1) & \text{if } n>1 \\ 1 & \text{if } n\leq 1 \end{cases}$$

Example 2 (cont'd): factorials ...

```
fac(n): if(n>1,
           n*fac(n-1),
           1)
:<function fac>
fac(10)
:3628800
fac(12)
:479001600
```

greater than 12
⇒ *overflow!*

Example 3: Fibonacci...

fundamental property: "a Fibonacci number is the sum of its two predecessors"

$$fib(n) = \begin{cases} fib(n-1)+fib(n-2) & \text{if } n>1 \\ 1 & \text{if } n\leq 1 \end{cases}$$

Example 3 (cont'd): Fibonacci...

```
fib(n):if(n>1,
          fib(n-1)+fib(n-2),
          1)
:<function fib>
fib(5)
:8
fib(10)
:89
fib(20)
:10946
fib(30)
```

time $\rightarrow \infty$
because: fib(n) takes double the amount of time of fib(n-1)!

transcript

Example 3 (cont'd): Fibonacci...

```
fib(p,q,r):if(r>1,
                fib(q,p+q,r-1),
                q)
:<function fib>
fib(1,1,5)
:8
fib(1,1,10)
:89
fib(1,1,20)
:10946
fib(1,1,30)
:1346269
```

time is ok now because: fib(n) takes double the amount of time of fib(n//2)!

transcript

Example 4: quadratic equations ...

```
L(a,b):
   {display('solution: ');
    if(a=0,if(b=0,'IR','none'),-b/a)}
<function L>
L(0,0)
solution: IR
L(1,2)
solution: -2
L(0,1)
solution: none
```

first step: coefficient of x^2 is zero

Example 4 (cont'd): quadratic equations ...

```
Q(a,b,c):
    if(a=0,
       L(b,c),
       {D: b^2-4*a*c;
        display('solution: ');
        if(D<0, display('none'),
           if(D=0,-b/(2*a),
                  [(-b-sqrt(D))/(2*a),
                   (-b+sqrt(D))/(2*a)]))})
:<function Q>
```

next step: coefficient of x^2 is not zero

transcript

Example 4 (cont'd): quadratic equations ...

```
Q(1,0,0)
 :solution = 0.000000
Q(1,2,1)
 :solution = -1.000000
Q(1,-4,3)
 :solution =  [1.000000, 3.000000]
Q(0,0,0)
 :solution = IR
```

transcript