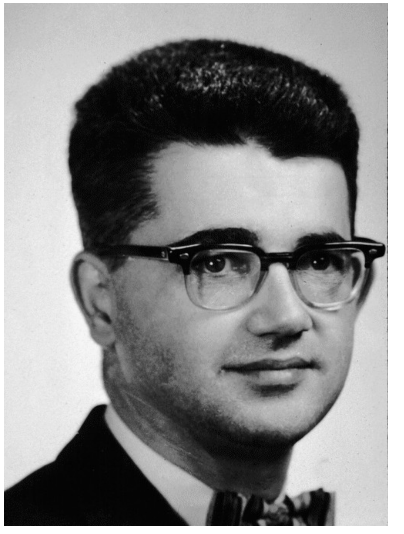
Lisp

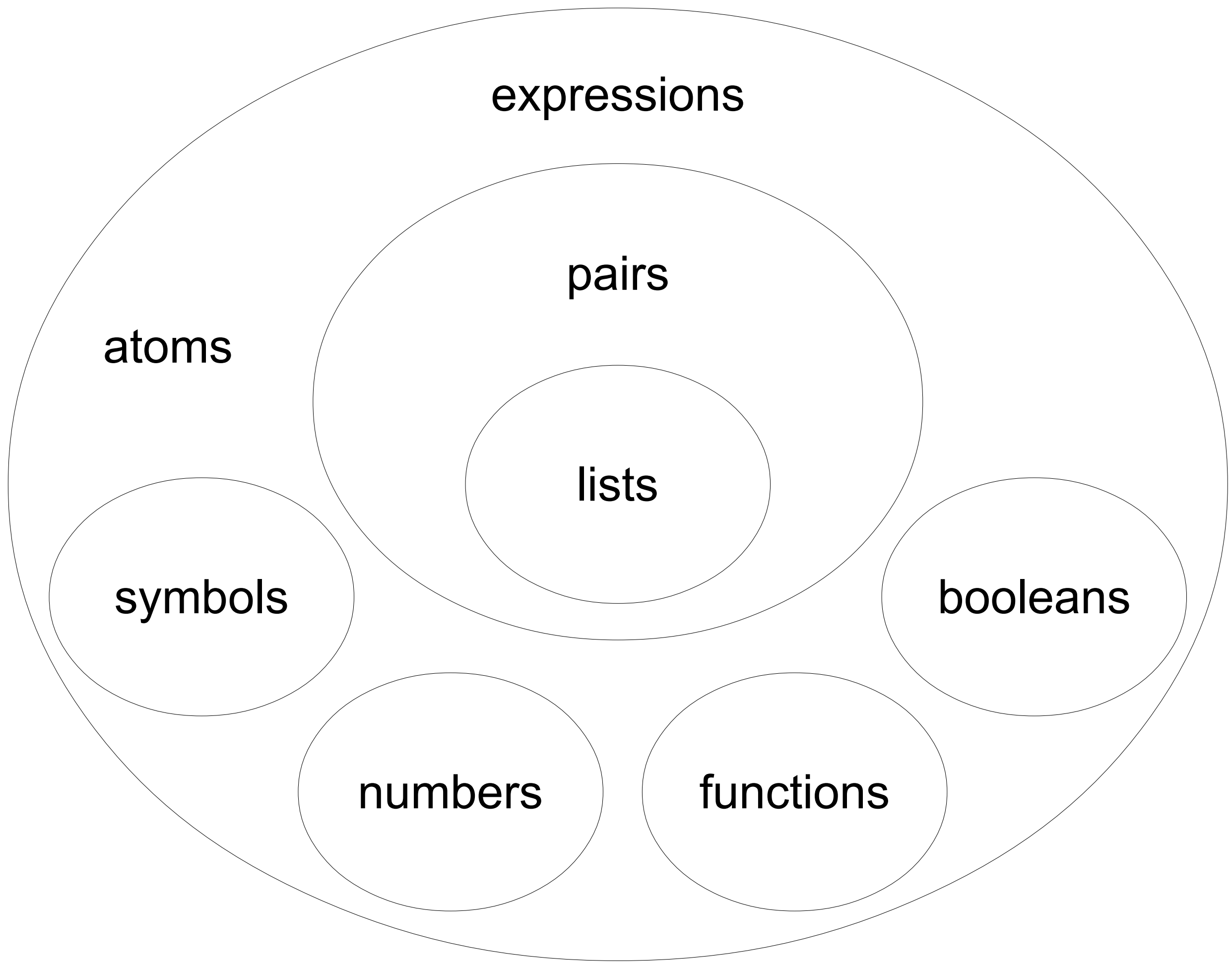
- Lisp was invented in 1958 at MIT by John McCarthy.
- Lisp stands for *list processing.*
- Its intended use was research in *artificial intelligence.*
- It is based on the λ calculus which was invented by Alonzo Church in the 1930's.
- Many features of modern programming languages such as garbage collection, first class functions and compiling to virtual machine bytecode first appeared in Lisp.



Scheme

- Scheme is a dialect of Lisp invented at MIT in 1975 by Guy Steele and Gerald Sussman.
- It corrected some design flaws in Lisp
  - Single name space for functions and data
  - Lexically (not dynamically) scoped variables
  - Optimization of tail calls
- No programming language matches Scheme in ratio of expressive power to implementation size.

Scheme Datatypes



Expressions

- In Scheme all values are *expressions.*
- Expressions are either *pairs* or *non-pairs.*
- Pairs are the building blocks of *lists.*
- Non-pairs are also called *atoms.*
- Atomic types include *symbols, numbers, booleans* and *functions.*
- Lists can be nested and can contain expressions of different types.

Programs

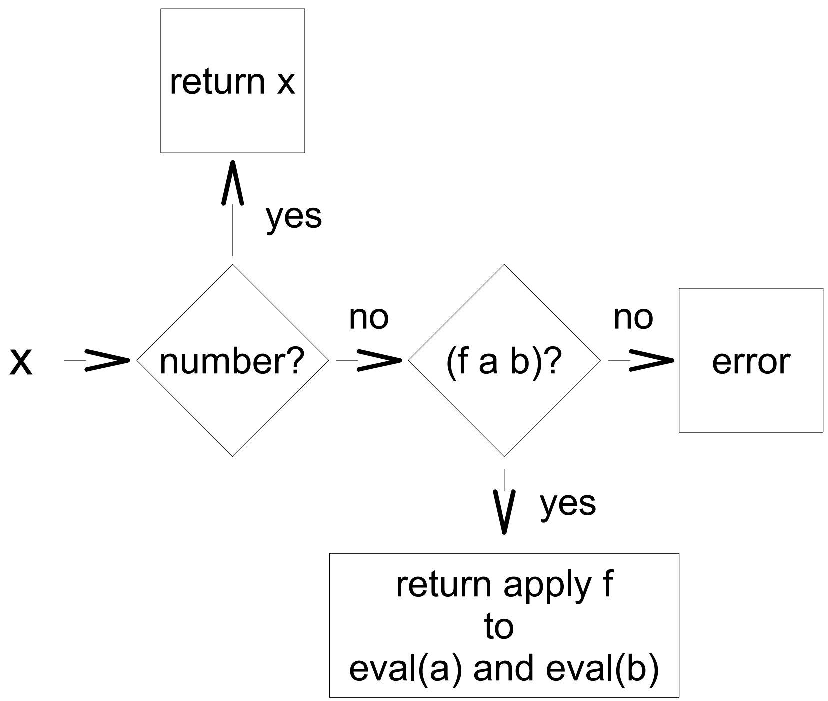
- Both programs and data are expressions.
- Programs are represented as nested lists.
- For example,

  (/ (+ (- b) (sqrt (- (* b b) (* 4 (* a c))))) (* 2 a)).
- Function calls in Scheme are written in *prefix notation.*
- O*perators* appear before *operands* in the list representing the function call, *e.g.,* (+ 1 2).

Programs

- Expressions representing programs are *evaluated* by *recursively* evaluating subexpressions.
- All atomic types except *symbols* evaluate to themselves.
- Recursive evaluation bottoms out on these *self-evaluating* types.

eval 0.0


return x
yes
x
number?
no
(f a b)?
no
error
yes
return apply f
to
eval(a) and eval(b)

Evaluating an Expression

(/ (+ 7 (- 3 2)) 2) → (/ (+ 7 1) 2) → (/ 8 2) → 4

Referential Transparency

- When a function is called second and subsequent times on given values, the result is always the same.
- Referential transparency simplifies analysis of program behavior.
- When a programming language is referentially transparent, code can be transformed algebraically without breaking it.
- These transformations can result in significant gains in *efficiency* and *parallelism*.

No Referential Transparency

- Consider the following C expression

  $((x{+}{+}) * y) + (x * (y{+}{+}))$

- Now let x = 1 and y = 2.
- If the left subexpression is evaluated first

  $((x{+}{+}) * y) + (x * (y{+}{+})) \rightarrow 2 + (x * (y{+}{+})) \rightarrow 2 + 4 \rightarrow 6$

- However, if the right is evaluated first

  $((x{+}{+}) * y) + (x * (y{+}{+})) \rightarrow ((x{+}{+}) * y) + 2 \rightarrow 3 + 2 \rightarrow 5$

- The program's behavior depends on the order of evaluation of its subexpressions!

First Class Datatypes

- A datatype is *first class* if functions can take values of that type as arguments and return values of that type as results.
- If a datatype is first class then new values of that type can be created at run time.

First Class Functions

- In Scheme, *functions* are first class.
- Functions which take functions as arguments and return functions as results are called *higher-order functions.*
- Higher-order functions can be used to abstractly represent sets of similar functions.
- The use of higher-order functions as a method of abstraction leads to increased program *modularity*.

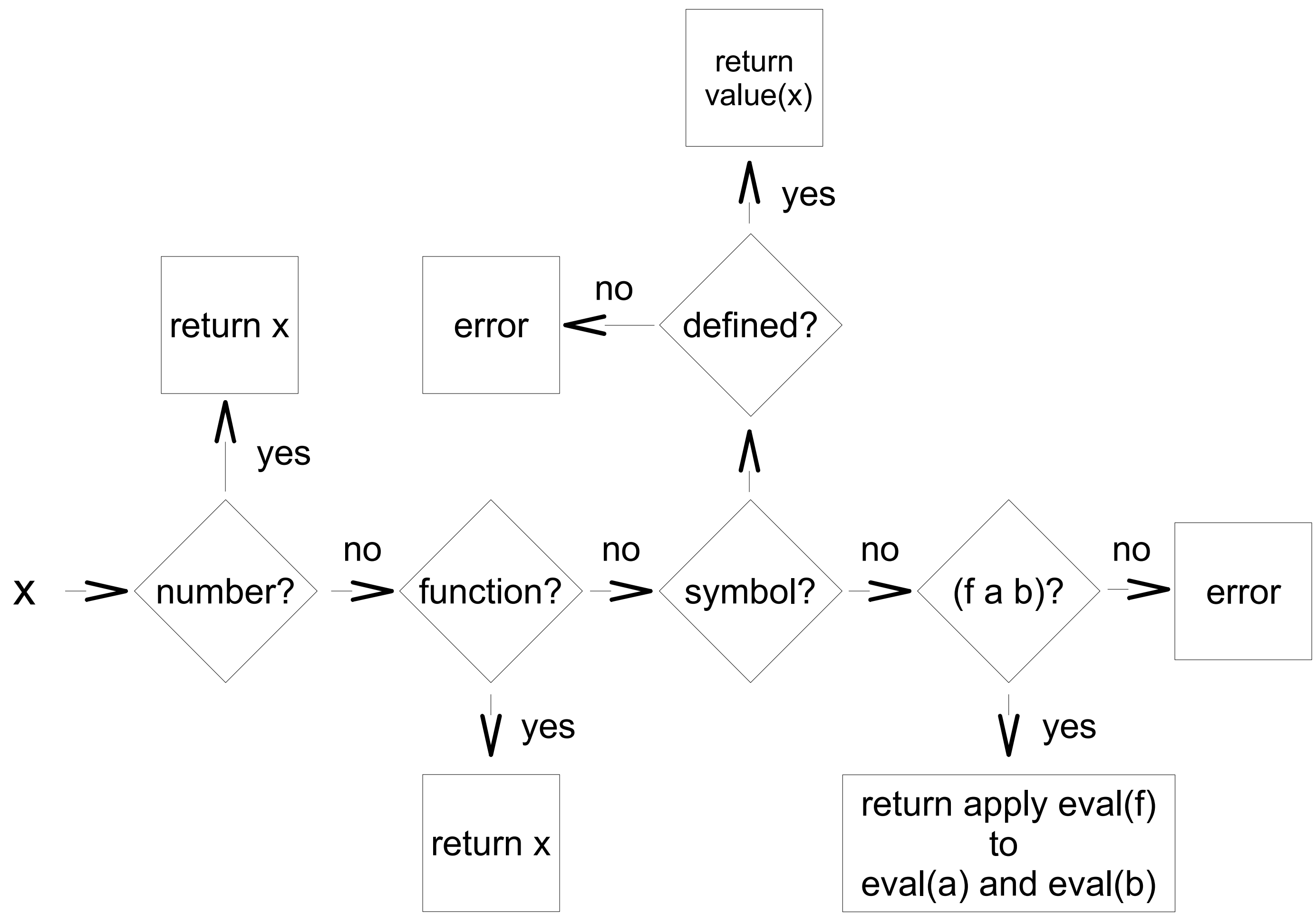
Symbols

- Symbols are the only atomic type that is not self-evaluating.
- Symbols serve as names for *other* values.
- They are different from *variables* in imperative programming because the values they represent never change.
- Functional programming languages are sometimes called *single-assignment* languages because once defined, a symbol's value never changes.

Symbols with Function Values

- In the expression (+ 1 2), the plus sign is not the function that adds two numbers, it is a *symbol.*
- The symbol's value is the function that adds two numbers.

eval 1.0

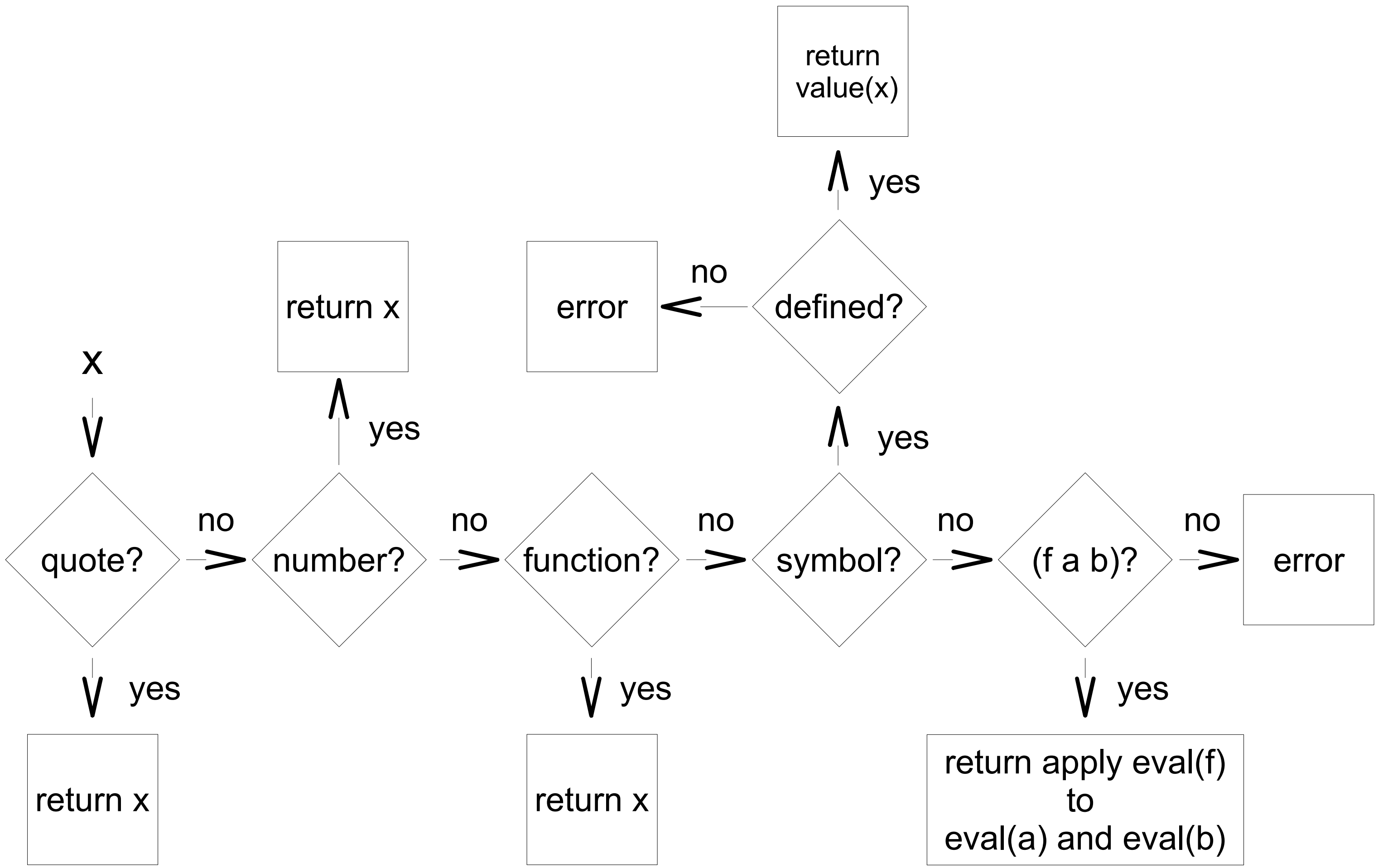

return
value(x)
yes
return x
error
no
defined?
yes
x
number?
no
function?
no
symbol?
no
(f a b)?
no
error
yes
yes
return x
return apply eval(f)
to
eval(a) and eval(b)

Program Data Equivalence

- If programs and data are both expressions, how does the evaluator know which is which?
- How does it know what expressions represent programs and should be evaluated and what expressions represent data and should be left alone?
- Expressions which are data are *quoted.*
- For example

'(+ 1 2) → (+ 1 2)

eval 2.0


return
value(x)
yes
return x
error
no
defined?
x
yes
yes
no
no
no
no
no
quote?
number?
function?
symbol?
(f a b)?
error
yes
yes
yes
return x
return x
return apply eval(f)
to
eval(a) and eval(b)

True and False

- Boolean true and false are typed #t and #f.
- However, #t is sort of useless, because any value that is not #f is considered true in Scheme.

Special-Forms

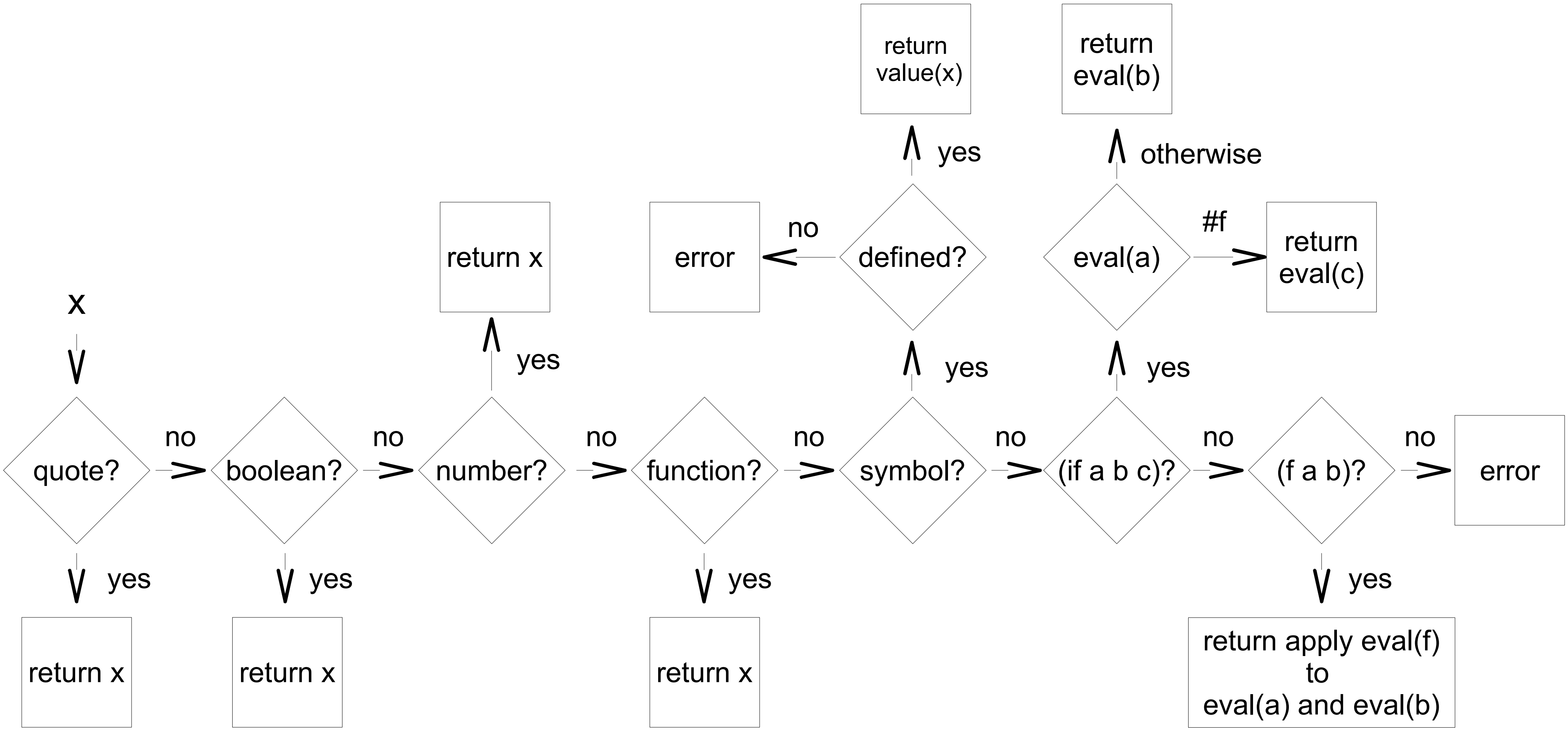
- A *special-form* is an exception to the normal rules of evaluation.
- Normally, all of a function's arguments are evaluated before the function is applied to them.
- Expressions can be conditionally evaluated in Scheme using the *if* special-form.

if special-form

- The if special-form evaluates its first argument.
- If the first argument is not #f, it evaluates and returns the value of its second argument.
- Otherwise, it evaluates and returns the value of its third argument.
- For example,

    (if (= 1 1) 0 1) → 0

eval 3.0



return
value(x)
return
eval(b)
yes
otherwise
return x
error
no
defined?
eval(a)
#f
return
eval(c)
x
yes
yes
yes
quote?
no
boolean?
no
number?
no
function?
no
symbol?
no
(if a b c)?
no
(f a b)?
no
error
yes
yes
yes
yes
return x
return x
return x
return apply eval(f)
to
eval(a) and eval(b)

"If ever the time should come, when vain and aspiring men shall possess the highest seats in government, our country will stand in need of its experienced patriots to prevent its ruin."
Samuel Adams