Order of evaluation matters for statements, and for expressions with side effects. Ordering for statements is control flow.

[Expressions have values; statements don’t. Statements are evaluated for their side effects. Expressions may have side effects, but don’t necessarily (and it’s often considered bad style).]

Principal paradigms for control flow:
- sequencing
- iteration
- subroutine, recursion (and related control abstractions, e.g. iterations)
- nondeterministic
- concurrency

Expression evaluation

Operators are built-in functions with often
non-standard identifiers (e.g. +,-,=).

Terms are possibly used as operators for most functions.

Precedence and associativity

C has 15 levels — too many to remember.

Pascal has 3 levels — too few for good semantics.

Fortran has 8; Ada has 6.

I don’t like the rules in any of these (Fortran probably closest).

Ada puts +, and, or at same level.

Pascal magma: if a + b == c and d then

Lesson: when unsure, use parentheses!

expression evaluation order matters

Consider f(x), x, x+y, x+y, x+y,

d might have side effects

reordering might improve register allocation

some arguments might already be available in registers on old machines with old compilers, last-to-first would optimize

stack access

Most languages say order of evaluation of arguments is undefined.

Likewise, most leave operand evaluation order undefined.

e.g., in f(x, a, b) = 9, b, b

arithmetical identities work in math class; not always on a computer

commutativity of + and * is usually safe

associativity is not

 murdering arguments — e.g., in Haskell

Expressions may have side effects, but don’t necessarily.

Statements are evaluated and may or may not be visible!

some languages evaluate arguments in a more general sense, assignment statements provide the ultimate

in a more general sense, assignment statements provide the ultimate

effect of call other than return value.

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SML/OCaml)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SM/OCaml/ Haskell)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SML/OCaml/ Haskell)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SML/OCaml/ Haskell)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SML/OCaml/ Haskell)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.

A side effect is some permanent state change caused by execution of

a function

but has costs

and also avoids some races in parallel programs

requiring (or defaulting) initialization avoids use of garbage

globals

C, C++, Python

Java $v = a + b$

Smalltalk

Vice versa, though user can choose which object variables are values

("expanded", in Eiffel terminology)

and which are references

Assignment statement (or expression) executed for its side effects;

operator assignments ($=, +=$ etc)

should avoid redundant work (or need for optimization)

perform side effects exactly

avoid redundant work (or need for optimization)

Handy

Infix syntax (as opposed to prefix for most functions)

special non

C, C++, Ada

most functional languages (Lisp/Scheme, SML/OCaml/ Haskell)

Smalltalk

Java deliberately in-between

in-built types are values

user-defined types are objects — references

In (pure) functional, logic, and dataflow languages, there are no

side effects.

They change the value of a variable.