IMPERATIVE LANGUAGES

Most commonly-used programming languages are imperative: they consist of a sequence of actions that alter the state of the world.

State includes the values of program variables and also the program’s external environment (e.g. files the program reads or writes).

Imperative programming is a good match to underlying Von Neumann machine programs, which are sequences of instructions that modify the contents of registers and memory locations.

- User-program variables are mapped to machine locations.
- User-program operations correspond to primitive machine instructions.

Imperative languages are also suitable for writing reactive programs that interact with the state of the “real world.” Examples:

- Reading mouse clicks and modifying the contents of a display.
- Controlling a set of relays in an external device.

Imperative programming is the dominant paradigm, but there are alternative “declarative” paradigms too...

STATEMENTS AND EXPRESSIONS

Many languages put have a separate syntactic category of statements (or commands) that includes stateful operations which don’t produce a result value.

But in some languages, certain expressions can also affect the state (in which case they are said to have side-effects) in addition to returning a result.

Also, most languages support user-defined functions, which contain statements but return a value and are invoked in an expression context; this is another way expressions can have side-effects.

ASSIGNMENT

The basic primitive stateful operation is typically assignment, which alters a value stored in a location.

Depending on language, assignments are statements (with no result value), or expressions (maybe with result value).

In the simplest form, the location is associated with a simple variable, e.g.,

\[ a := a + 2 \]

(Will use := for assignment, = for equality relational operator. C/C++/Java use =, == respectively: a bad idea, because both form expressions.)

In most languages, the variable name \( a \) means different things on the left-hand and right-hand sides.

On the LHS, \( a \) denotes the location of the variable \( a \), into which the value of the RHS expression is to be stored.

On the RHS, \( a \) denotes the value currently contained in \( a \), i.e., it indicates an implicit dereference operation.
**Order of Evaluation**

Order of stateful operations affects program semantics (behavior).

**Statements** are always explicitly ordered, making these differences obvious.

**Expressions** can also have side-effects, but order of evaluation is often **under-specified** (precedence and associativity don’t always fix order).

ANSI C example:

```c
a = 0;
b = (a = a + 1) - (a = a + 2);
```

Result (1-3 = -2 or 3-2 = 1 ?) depends on compiler whim.

**Hidden Side Effects**

Side-effects are not always obvious:

```c
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2)); // = 0 or 2 ?
```

Keeping expression evaluation order or argument evaluation order undefined sometimes lets compiler generate more efficient code.

But most modern languages have moved towards precise definition of evaluation order within expressions (e.g., left-to-right).

**Structured Control Flow**

All modern higher-level imperative languages are designed to support **structured programming**.

Loosely, a structured program is one in which the syntactic structure of the program text corresponds to the flow of control through the dynamically executing program.

Originally proposed (most famously by Dijkstra) as an improvement on the incomprehensible “spaghetti code” that is easy to produce using the labels and jumps supported directly by hardware.

More specifically, structured programs use a very small collection of (recursively defined) **compound statements** to describe their control flow.

**Kinds of Compound Statements**

- **Sequential composition**: form a statement from a sequence of statements, e.g.
  (Java) `{ x = 2; y = x + 4; }
  (Pascal) begin x := 2; y := x + 4; end

- **Selection**: execute one of several statements, e.g.,
  (Java) `if (x < 0) y = x + 1; else z = y + 2;

- **Iteration**: repeatedly execute a statement, e.g.,
  (Java) `while (x > 10) output(x--);
  (Pascal) `for x := 1 to 12 do output(x*2);

And that’s all!

But is this really a joke?

Even with a **GO TO**, we must examine both the branch **and** the target label to understand the programmer's intent.