Introduction to programming languages and paradigms

Christoph Groth (CEA Grenoble)

MaiMoSiNE, 11 January 2012

Overview

Three parts:

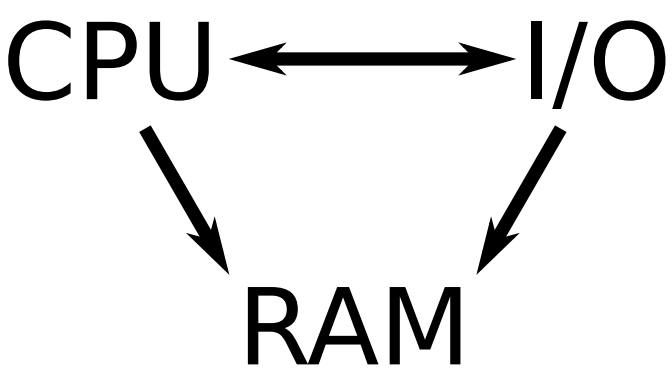
1. A short introduction to programming
2. Important programming paradigms
3. Advice on choosing the right programming language for a specific project involving numerical computation

Part 1: A short introduction to programming

- How does a computer work?
- How to tell it to do useful things?

How does a computer work?

(from a programmer's perspective)



CPU *Central Processing Unit*
Arithmetic and logic engine that executes user programs

RAM *Random Access Memory*
Very long (currently billions) numbered (=address) sequence of small integers: bytes. Can be interpreted in various ways.

I/O *Input and Output devices*
Hard drive, graphics, keyboard, mouse, printer, etc.

Machine code

Instructions in RAM executed directly by the CPU

Add registers 1 and 2, place result in register 6.

```
[  op  |  rs |  rt |  rd |shamt| funct]
    0     1     2     6     0     32     decimal
 000000 00001 00010 00110 00000 100000   binary
```

Jump to address 1024.

```
[  op  |          target address         ]
    2                 1024                decimal
 000010 00000 00000 00000 10000 000000   binary
```

(MIPS architecture)

Assembly language

Symbolic representation of machine code

```
        mov     cx, SIZE
@@3:    mov     al, [di + 640]
        mov     dl, [di]
        add     ax, dx
        mov     dl, [di + 321]
        add     ax, dx
        mov     dl, [di + 319]
        add     ax, dx
        shr     ax, 2
        jz      @@4
        dec     ax
@@4:    mov     [di], al
        inc     di
        loop    @@3
```

(Intel x86 assembly language)

Compiled languages

Same code as on previous slide, in C programming language:

```
for(i = 0; i < size; ++i)
{
    avg = (s[i] + s[i+319] + s[i+321] + s[i+640]) / 4;
    if (avg != 0) --avg;
    s[i] = avg;
}
```

- Easier to write and maintain programs.
- Can be almost as fast as assembly.
- Can be machine-independent.

Other compiled languages: FORTRAN, C++, Pascal

Interpreted languages

A program executes another program!

- Normally slower. (but can be fast)
- Platform-independent (e.g. JavaScript)
- Language can be *dynamic*.
- Easier to use, e.g. as a scripting language in a simulating framework.

Examples: MATLAB, Python, PHP, JavaScript, sh, ...

Part 2: Important programming paradigms

- Procedural versus object-oriented
- Imperative versus declarative
- Static versus dynamic
- Sequential versus parallel

Examples will be in *Python*, a flexible programming language which should be readable quite easily.

Procedural versus object-oriented

Procedural Programming in terms of procedures (=portions of code within that perform a specific task).

Object-oriented Programming in terms of objects which have user-defined types.

A procedural random number generator

(method: linear congruential)

```
a = 1664525
c = 1013904223
state = 0

def generate_next():
    global a, c, state
    state = (a * state + c) % 2**32
    return state

for i in range(5):
    print generate_next()
```

More than *one* random number generator?

Output

```
1013904223
1196435762
3519870697
2868466484
1649599747
```

Same generator, expressed as a *class*

```
class Generator(object):
    def __init__(self, state=0, a=1664525, c=1013904223):
        self.state = state
        self.a = a
        self.c = c

    def generate(self):
        self.state = (self.a * self.state + self.c) % 2**32
        return self.state

g1 = Generator()
g2 = Generator(555)
for i in range(5):
    print g1.generate(), g2.generate()
```

Same generator, expressed as a *class* (improved)

```
class Generator(object):
    def __init__(self, state=0, a=1664525, c=1013904223):
        self.state = state
        self.a = a
        self.c = c

    def __call__(self):
        self.state = (self.a * self.state + self.c) % 2**32
        return self.state

g1 = Generator()
g2 = Generator(555)
for i in range(5):
    print g1(), g2()
```

Output

```
1013904223 1937715598
1196435762 2659257237
3519870697 3576096752
2868466484 910318223
1649599747 1171883682
```

Procedural versus object-oriented: summary

Object-oriented programming:

- Define new types (classes), which behave like the build-in ones.
- Relations between types can be expressed and used, e.g. a "Circle" is a "Shape".
- Very useful for some applications: some simulations, computer graphics
- BUT: Procedural model is sometimes good enough, and can be simpler.
- In practice often a mixture is used.

Imperative versus declarative

Imperative Specify the sequence of operations for solving the problem.

Declarative Specify the problem.

Example: Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_n = F_{n-1} + F_{n+2}, F_0 = 0, F_1 = 1$$

Imperative Fibonacci numbers

```
def fibonacci(n):
    if n < 2:
        return n
    a, b = 0, 1
    for i in range(n - 1):
        a, b = b, (a + b)
    return b
```

- We specify the sequence of operations.
- *Order of execution* is important.
- *State* is important.
- Runtime is $O(n)$: efficient

Declarative (functional) Fibonacci numbers

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

- We specify the problem.
- Order of execution is unimportant.
- No state (=no assignments or side-effects).
- Runtime is $O(n^2)$, unless we have a very clever implementation.

Imperative versus declarative: summary

- Declarative approach is more elegant, but there are problems in practice, i.e. efficiency.
- There exist many purely functional programming languages, but they never became mainstream. Too abstract?
- Some algorithms are better expressed in an imperative style, some in a functional style.
- Advice: be aware of the functional approach and try to use it whenever it makes the code clearer, but don't be dogmatic about it.

Static versus dynamic

Static programming languages Properties (types of variables, definitions of types, code of functions) are fixed when the program is compiled.

Dynamic programming languages The things mentioned above can be changed during runtime. This implies an interpreted language.

Example 1: dynamic typing

```
def first(seq):
    a = seq[0]
    for b in seq[1:]:
        if b < a:
            a = b
    return a

print first([2, -7, -2])
print first(['Chambery', 'Annecy', 'Grenoble'])
```

The function *first* works for any sequence of any type which supports the operation "smaller then".

Example 2: “scriptability”

```
from math import *
import sys

func = sys.argv[1]
x = float(sys.argv[2])
dx = 0.00001
f0 = eval(func)
x = x + dx
f1 = eval(func)
print (f1 - f0) / dx
```

This is a complete program to numerically approximate the derivative of any given function. For example:

```
$ python test.py 'cos(x)**2' 0.5
-0.841476387781
$ python test.py 'exp(x)' 1
2.71829541996
```

Example 3: first class functions

Functions are objects like everything else.

```
def make_is_inside(radius):
    rr = radius**2
    def is_inside(point):
        return sum(elem**2 for elem in point) < rr
    return is_inside

is_within = make_is_inside(2)
print is_within([1, -1])      # prints "True"
is_within = make_is_inside(1)
print is_within([1, -1])      # prints "False"
```

Static versus dynamic: comparison

- Dynamic languages are more expressive.
- The user can directly interact with the program without recompiling.
- BUT: Static checking can help to find errors.
- AND: Dynamic languages are interpreted and usually significantly slower.
- metaprogramming (e.g. C++ templates): "dynamic typing at compile time". Very powerful, but can get complicated.

Sequential versus parallel

- Sequential is the traditional, easier approach.
- It's fundamentally more difficult to express algorithms in a parallel way.
- *Amdahl's law*: the possible speedup is limited by the time needed for the sequential part of the program.
- BUT: increase of performance mostly due to parallelization. (Current supercomputers have > 100000 CPU cores.)
- Luckily, some problems are *embarrassingly parallel*.