Programming Languages

There are many programming languages available.

Formally, most of them are “equivalent" – i.e. you can do in B what you can do in A and vice versa.

However some Languages are more “useful" than others

Desiderata for Programming Languages

- Speed
- Ease of Coding
  - Powerful Language (many functions)
  - Existing Code Libraries
  - Reusability
- Portability
- Maintainability (can someone else understand the code a year later)?

Aspects for language choice

- What will be the input?
- How often will the program run?
- How much programming time can be devoted?
- Is there existing code/libraries that can be (re)used. Is cost an issue?
- For how long has the program to be maintained? Will there be other users?
- What languages are known/available?
- How long does programming take, how long will the program run? (Is speed an issue)?

Assembler

Assembler (Mnemonics for machine code) is the code the processor runs on. It is fast, but very inconvenient to code.

Code portability is almost neglegible.

On modern processors, hand-optimization is almost impossible.

Unless you write hardware drivers or compilers, you are unlikely to use assembler ever.

Running other Languages

Compilation translates high language code into machine language. This translation has to be done after every change in the program.

Variables often are kept in processor registers.

Advantages:

- Speed tends towards Assembler
- Compiler optimization can obey complicated optimization rules

Disadvantages:

- When debugging, the translated code is running (variable inspection can be tedious).
- Delay due to compilation step.
- The compiled code is machine dependent (only sourcecode portability).

Interpretation

An *Interpreter* is a program that reads code in some language and performs commands according to the language definition. It keeps variables in memory areas that are reserved for the variable.

Advantages:

- No delay for compilation
- The running code is the source: Portability
- Convenient debugging
- Dynamic data structures, Garbage Collection

Disadvantages:

- Not as fast. (Remedy: Interface to compiled functions for time-critical routines)

Pseudo-Code

To overcome the portability problem of compiled code and improve the runtime over interpreted code, sometimes code gets compiled in the machine language of some “virtual” processor. An interpreter then simulates this virtual processor.

The virtual machine might also provide more convenient memory managment (GC)

If the “virtual” machine is cleverly designed, the runtime overhead is low (but typically a compiled language will still be faster).

Prominent example: Java

Object Orientation

Suppose you want to write a library routine that multiplies matrices. Users might want to use this routine for matrices containing integers or various floating point numbers.

In a classical programming language you have to write different (though almost identical) routines for the various types of matrix entries.

In an object oriented language, one can write a *generic* matrix multiplication routine that only calls +,-„/. The programming language then will automatically call the correct subroutines.

The idea behind object orientation is that one should not look inside data types but only have *accessor functions* for certain tasks. These functions then can be used for different types of objects.

Routines that only use these documented accessor functions thus are independent of the type of objects.

Drawback: Some languages got object orientation added as an afterthought.

Some common languages

Fortran Old war-horse of numerical computation. Current version Fortran90 contains newer concepts, but the language still carries outdated constructs. Very large legacy library of code.

```
      PROGRAM SQUARE
      DO 15,I = 1,10
        WRITE(*, *) I*I
15    CONTINUE
      END
```

Lisp

Very high-level interpreted language. Rather AI-type applications than scientific computation.

```
(let ( (a 1) )
  (while (<= a 10)
    (princ (* a a)) (princ " ")
    (setq a (+ a 1))
  )
)
```

Algol Family

(Pascal, Ada) Well structured, sometimes too rigid.

```
PROGRAM Squares;
VAR i:integer;

BEGIN
  for i:=1 to 10 do BEGIN
    writeln(i*i);
  END;
END.
```

C

High-level language that gets as close to machine code as possible. Probably (at the moment) *the* general-purpose implementation language.

Object-oriented version: C++

```
#include <stdio.h>
main()
{
  int i;
  for (i=1;i<=10;i++)
    printf("%d ",i*i);
}
```

Java

Object-oriented, interpreted, architecture-neutral. Closest rival to C.

```
Class Squares {
  for (int i=1;i<=10;i++) {
   System.out.println(i*i);
  }
}
```

Matlab

Good (and convenient) matrix functions. Can be good place for numerical prototyping. (Open-Source clone: `Octave`)

```
>> l=[1:10]
l =
     1     2     3     4     5     6     7     8

>> l.^2
ans =
     1     4     9    16    25    36    49    64
```

Maple

Algol-family language with extra (slightly weird) list data types.

Strong symbolic computation facilities.

```
seq(x^2,x=1..10);
```

GAP

"Maple for discrete Mathematics". Very convenient list functions, interpreter convenient for debugging. No floats.

```
for i in [1..10] do
  Print(i^2);
od;

List([1..10],i->i^2);
```

PERL

"Practical Extraction and Report Language": A language designed to permit easy text/file manipulation. Many string/list commands. Many library routines for "web" tasks.

```
print join(" ", map { $_ ** 2 } 1..10), "\n";
```

Example: Matrix conversion

A MATLAB matrix file contains each row in a line entries separated by spaces.

A Maple matrix is created in the form:

```
a:=array(1..2,1..2,[[1,2],[3,4]]);
```

We want to write a program that converts a MATLAB matrix file into a Maple matrix file.

```
#!/usr/local/bin/perl
sub convmat {
  my ($name)= @_;
  open(IN,$name);
  $outnam=">".$name.".map";
  open(OUT,$outnam);
  $cnt=0;
  $first=1;
  while ($line = <IN>) {
    $row="[";
    $startrow=0;
    while ($line =~ "( +)([-]*[0-9]+)(.+)") {
      $cnt=$cnt+1;
      if ($startrow==1) {
        $row=$row.","; }
      else { $startrow=1; }
```

```
      $row=$row.$2;
      $line=$3; }
    if ($first==1) {
      print OUT $name.":=array(1..".$cnt.",1..".$
      $first=0; }
    else { print OUT ",\n"; }
    print OUT $row."]";
  }
  print OUT "]);\n";
  close IN;
  close OUT;
}
while ($#ARGV >=0) {
  $file=shift; # get argument
}
convmat($file);
```

Numbers

A computer natively can represent numbers only by bit sequences, typically for as many bits as the size of registers:

8 bit	0–255	-128–127
16 bit	0–16383	-8192–8191
32 bit	0–4294967295	-2147483648–2147483647
64 bit	0–18446744073709551615	-9223372036854775808–9223

The first bit might be interpreted as sign bit.

Arbitrary precision Integers/Rationals

All other numbers have to be composed from such numbers, for example by considering them as "digits" and using the arithmetic methods we learned in school.

From this one can build an arithmetic for large integers (or rational numbers).

Problems:

- How to represent transcendental numbers?
  ($\longrightarrow$ interval arithmetic)
- Denominator growth $\frac{1}{41} + \frac{1}{715} = \frac{756}{29315}$.

Floating Point Numbers

A floating point number (FPN) is of the form

$$x = \pm(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}) \cdot \beta^e$$

with

- Base β
- Precision p
- $0 \leq d_i \leq \beta - 1$ ($i = 0..p-1$) "Digits".
- $L < e < U$ Exponent

We call $d_0 d_1 d_2 \cdots d_{p-1}$ the *mantissa* or *significand* and $d_1 d_2 \cdots d_{p-1}$ the *fraction* of x.

Examples

Represent $a = 0.1$:

$\beta = 10$, $p = 3$: $a = 1.00 \times 10^{-1}$.

$\beta = 2$, $p = 24$: No exact representation,
$a \sim 1.10011001100110011001101 \times 2^{-4}$.

Typical floating point systems

name	β	p	L	U
HP calculator	10	16	-499	499
IEEE single	2	24	-126	127
IEEE double	2	53	-1022	1023
Cray	2	48	-16383	16384
IBM mainframe	16	6	-64	63

Today most computers use the IEEE standards.

Unique representation

If we ensure $d_0 \neq 0$ (except $x = 0$) the system is called *normalized*. Then $1 \leq m < \beta$.

Machine numbers

There are $2(\beta-1)\beta^t(U-L+1)+1$ floating point numbers (FPN).

The smallest positive number is $ufl = \beta^L$, the largest is $ofl = \beta^(U+1)(1-\beta^{-t})$.

FPN are equal spaced only between consecutive powers of β.

If x can be represented exactly as an FPN, we call it a *machine number*.

Rounding

If x cannot be represented exactly, it will be approximated by

truncation (chop off tail) or

rounding to nearest number, tiebreak: last digit even.

(rounding is better, is default for IEEE.)

The resulting error is called *rounding error*.

Example: $\beta = 10$, $p = 3$. $a = 0.0314159$ is represented as 3.14×10^{-2}. The error is $.159$ units in the last place (ulp).

A correctly rounded number has an error of up to $\frac{1}{2}$ ulp.

Relative Error: $.0000159/.0314159 \cong 0.000506$

Relative error corresponding to $\frac{1}{2}$ulp

Number $d.ddddd \times \beta^e$, absolute error (with $\beta' = \beta/2$):

$$0.000000\beta' \times \beta^e = ((\beta/2)\beta^{-p}) \times \beta^e$$

All such numbers have (potentially) same absolute error byt values between β^e and $\beta \times \beta^e$. Thus relative error:

$$\frac{1}{2}\beta^{-p} = le\frac{1}{2}\text{ulp} \leq \frac{\beta}{2}\beta^{-p}.$$

Maximal relative error is thus:

$$\epsilon = (\beta/2)\beta^{-p}.$$

In the example: $\epsilon = .005$. Do not confuse ϵ with ufl.

Customary, errors are given in units of ϵ: Relative error 0.0005 in example corresponds to $.1\epsilon$.

Example: $x = 12.35$ approximated by 1.24×10^1. Error is 0.5ulp, relative error 0.8ϵ.

Calculate $8x = 99.8$. Computed value is 9.92×10^1. Error is 4.0ulp, relative error is still 0.8ϵ.

In IEEE single: $\epsilon = 2^{-24} \sim 10^{-7}$,
In IEEE double: $\epsilon = 2^{-53} \sim 10^{-16}$.

Typically: $0 < ufl < \epsilon_m < ofl$.