

# What is a programming language?

**COMP3610 — Principles of Programming Languages**

Ben Lippmeier

Australian National University

Semester 2, 2009

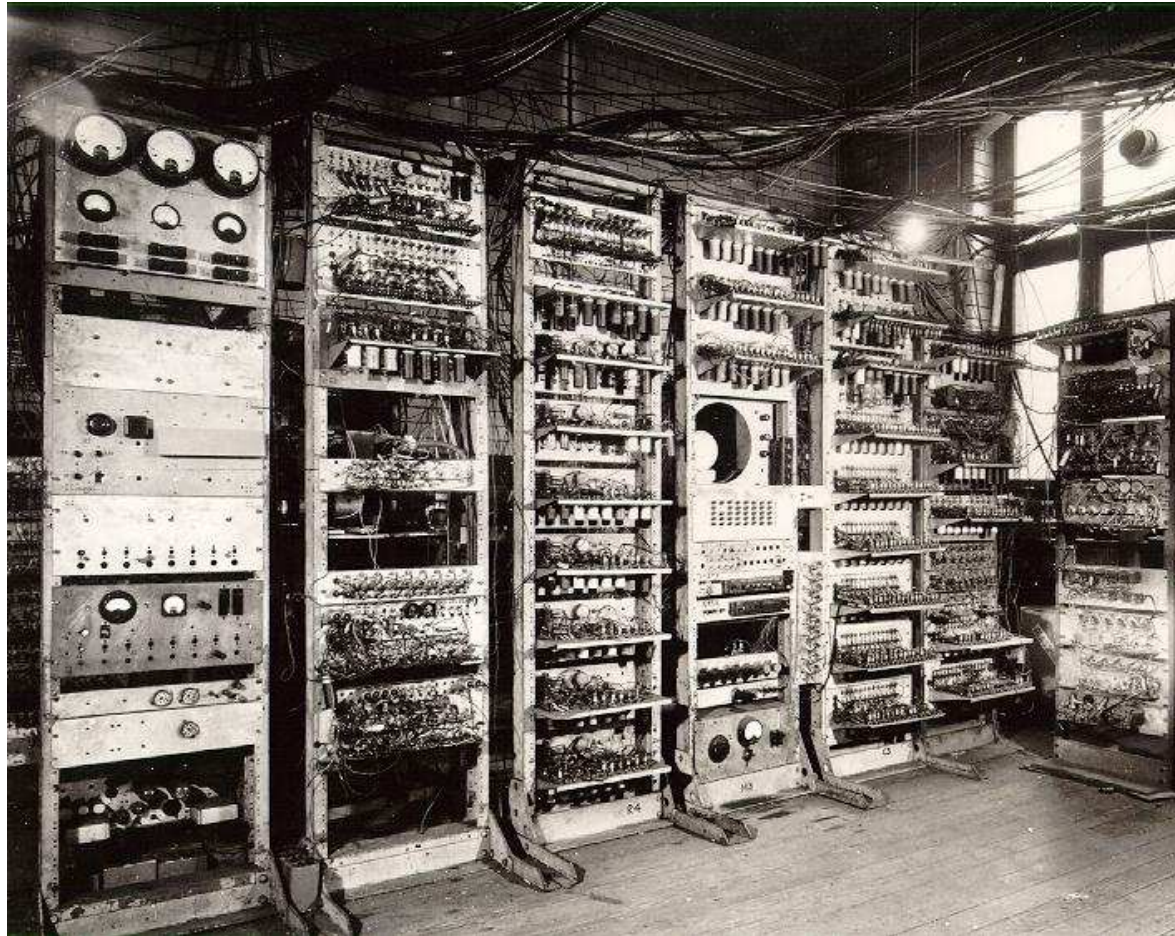
## von Neumann computers

The essential feature of what we call a computer today is that the program resides in the machine store, along with the data.

The program is a sequence of instructions coded in binary – it is therefore also data.

Is that convenient for humans? No!

## The Manchester Mark-1, circa 1949



## Step 1: Assemblers

An assembly language is a textual program notation with (roughly) a 1:1 correspondence between machine instructions and some “readable” symbolic representation.

- You don't have to remember the binary code for each of the instructions, but program is not portable between architectures.
- A two-pass assembler allows forward references in instructions, so you don't have to manually re-compute branch offsets when the program changes.
- Some programs were written entirely in assembly into the early 1990's. Doing so is rarely necessary (or worthwhile) in 2009.

## From Digdug for the Atari 7800 (MOS 6502), circa 1982

```
FINMOV                                ;DO THIS STUFF ONLY IF DIGDUG MOVES
    LDA    #0
    STA    DIGREST
    LDY    NEEDDIR                    ; DIGDUG HOLDING UP ROCK?
    BEQ    CHKFALL
    BMI    STRTFALL                    ; FF MEANS ANY MOVE WILL CAUSE FALL
    DEY                                ; TRUE DIRECTION NEEDED
    CPY    LASTMOVE                    ; WILL CURRENT DIR CAUSE FALL?
    BNE    CHKFALL
STRTFALL LDX    HITROCK
    LDA    #ONEDGE
    STA    TUMBLE,X
    LDA    #0
    STA    NEEDDIR
```



## Step 2: Programming languages

- Higher level (whatever that means)
- General arithmetic expressions
- Named variables
- Control structures
- Data structures
- Functions, Recursion
- Modules, etc.

Negative Consequence: more complex correspondence with machine code.

The program that does the translation (compilation) is much more complex.

## 1950s: Autocode → Fortran

The first programming languages.

- Very odd syntax by today's standards.
- The language designers and compiler developers had no knowledge of formal languages, text processing, or translation.
- Maintained quite a close correspondence with machine code e.g. computed goto, statements to check for register overflow, turn front-panel lights on etc.
- Later versions of Fortran removed machine dependent features, and added “missing” features. Fortran is still used today for scientific and engineering applications.

## From an Autocode program by Stuart Stong

```

JV I
TEXT
FEED COMPOSITION

*n20=TAPE _____ number of components
v213=TAPE _____ F
v240=TAPE n20 _____ Xi
v140=TAPE n20 _____ Ki
v3=0
n0=4
n1=0
v0=4 _____ V/L= 4
43) v215=v0+i*
v215=v213/v215 _____ L
v218=v213-*v215 _____ V
n1=0
42) v(20+n1)=v(240+n1)xv213 — FXi
n1=n1+i
→42, n1≠n20
v50=0
v3=0
n1=0
44) v10=v(140+n1)xv0
v10=i+v10

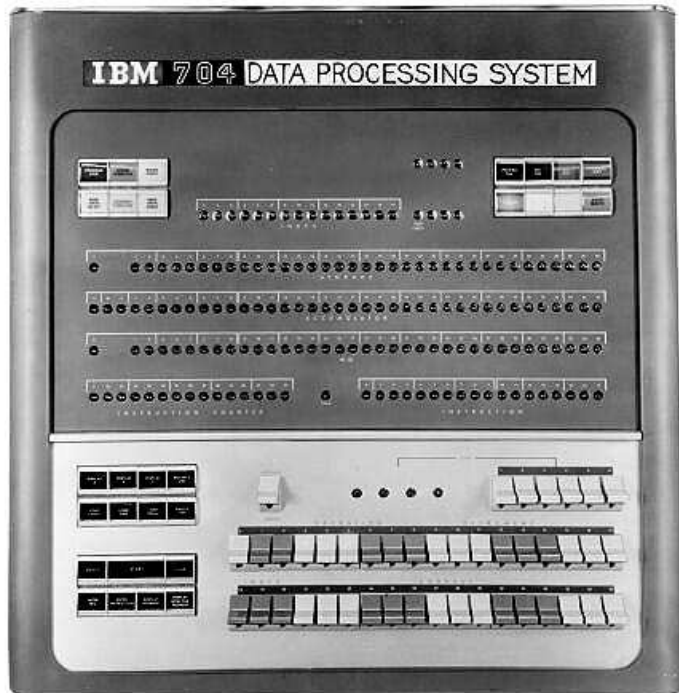
```

## From the Colossal Cave Adventure, Fortran IV

```
C  READ INFO ABOUT AVAILABLE LIQUIDS AND OTHER CONDITIONS,  
C  STORE IN COND.
```

```
1069  FORMAT(I3,I6,10I3)  
1070  READ(IPDATA,1069)K,(TK(J),J=1,10)  
      IF(K.EQ.-1)GOTO 1002  
      DO 1071 I=1,10  
        LOC=TK(I)  
        IF(LOC.EQ.0)GOTO 1070  
        IF(BITSET(LOC,K))CALL BUG(8)  
1071  COND(LOC)=COND(LOC)+2**K  
      GOTO 1070
```

## The IBM 704, for which Fortran was developed



# FORTRAN STATEMENT

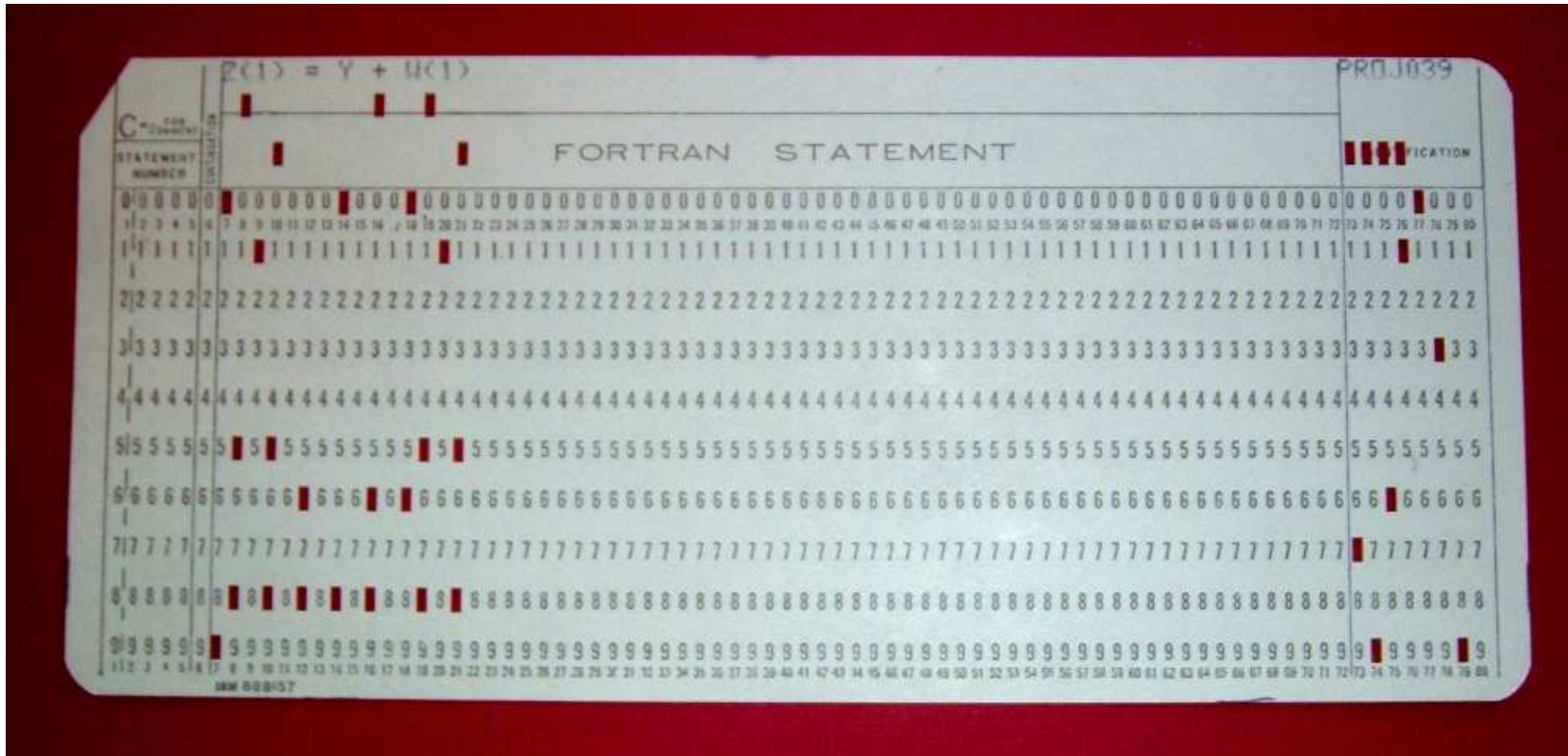


image: Arnold Reinhold CCSA 2.5

## Late 1950s: Algol (Algorithmic Language)

*“A great improvement on all its predecessors, and most of its successors.”*

– Tony Hoare

Laid the foundations of syntax definition and processing that persists today.

- Introduction of BNF (Backus-Naur Form) notation for describing syntax.
- Parameter passing techniques
- Local declarations
- Block control structures
- Inspired C, Pascal, PL/I etc etc.

## Algol code sample (from Wikipedia)

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of
    size n by m is transferred to y, and the subscripts of
    this element to i and k;
begin integer p, q;
    y := 0; i := k := 1;
    for p:=1 step 1 until n do
        for q:=1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
            end
        end
    end
end Absmax
```

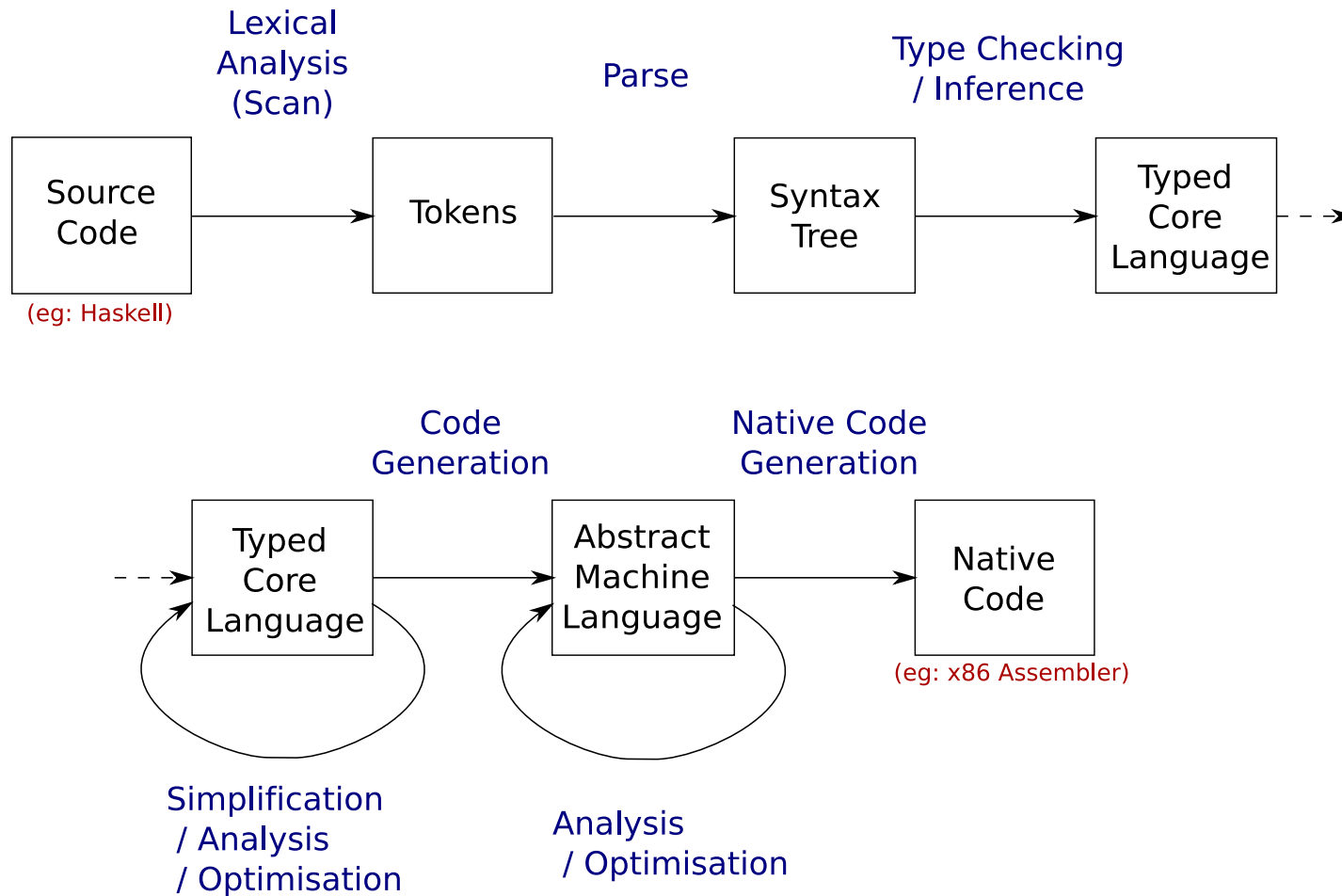
## Late 1970s, early 1980s: ML (Meta-Language)

“Well typed programs don’t go wrong” – Robin Milner.

Developed by Robin Milner and others at the University of Edinburgh, originally as part of their LCF theorem prover.

- Influenced by ISWIM (which was never implemented), also by LISP. ISWIM: If You See What I Mean. LISP: LISt Processor.
- ML includes the Hindley-Milner polymorphic type system. The compiler can reconstruct the type of every term in the program.
- A well typed program is guaranteed not to crash (“go wrong”) at runtime.
- Polymorphic type inference makes it practical to construct programs from higher order functions, known as “Combining Forms” by John Backus.
- Inspired O’Caml, Miranda, Clean, Haskell, Scala etc.

# Typical compiler structure

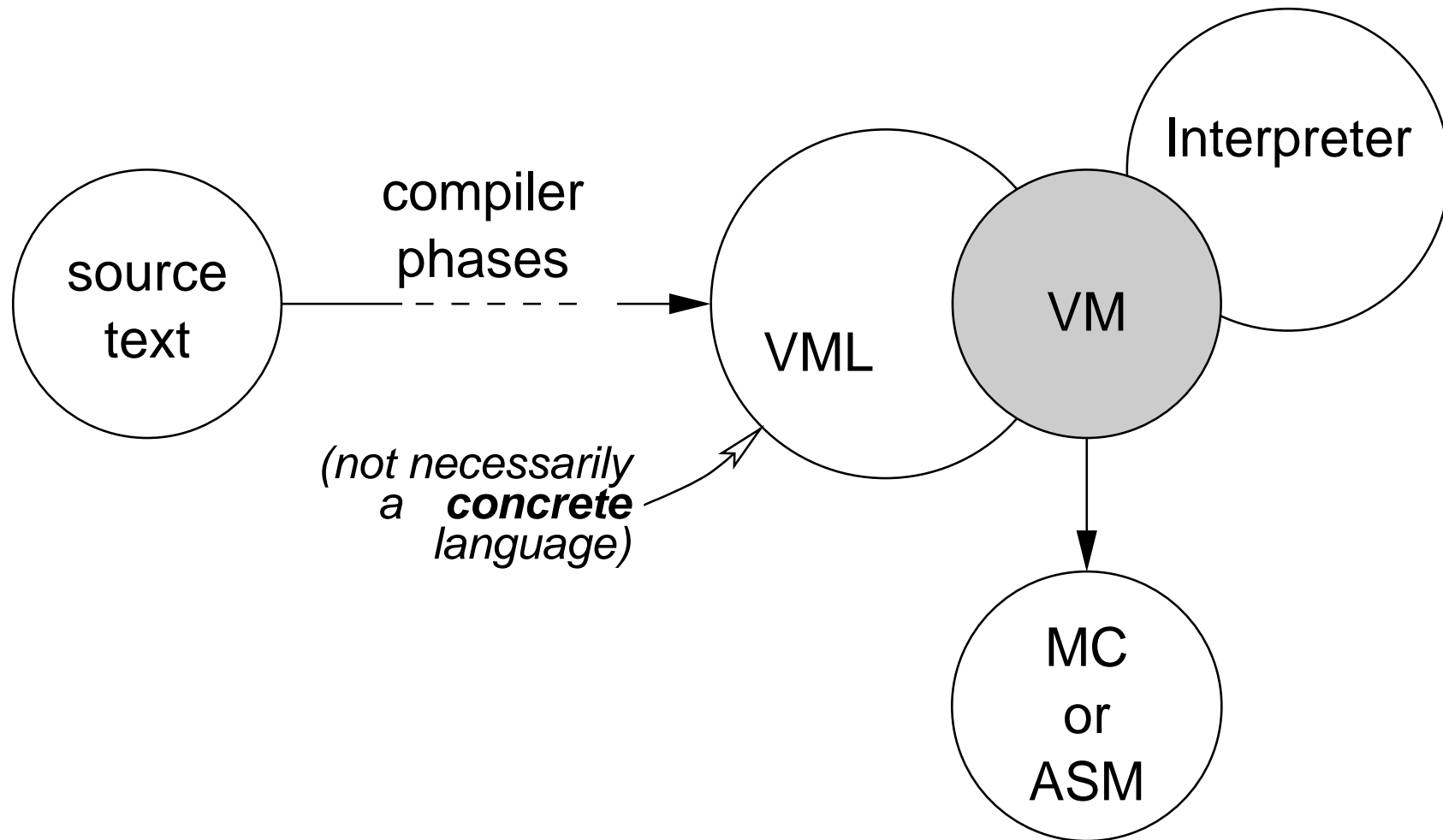


## Virtual machines

Compiling to machine code suits only one architecture. Porting a compiler to another architecture can be very hard work.

Starting(?) with BCPL (the forerunner of C) in the early 1960s, some compilers targeted a virtual machine (VM) rather than a concrete one.

- The compiler translates from the source program to the language of a virtual machine (VML), which is designed to fit between the source language and a range of target machines.
- Why? Because porting the VM can be a lot simpler than the whole system.
- The Pascal P-compiler (to the virtual P-machine) was the reason for Pascals popularity in the 1980s. Now we have the Java VM (Sun), CLR for .NET (Microsoft), LLVM (now backed by Apple) etc.



## Just In Time (JIT) Compilation

Web-deployed software must run on machines of different architectures. JIT compilation systems compile the source code just before it needs to be run on the target machine.

- Compilation to native code avoids the overhead of using an interpreted language.
- We can deploy code written in a VML, then do only the VML → object code translation before execution. This can be much faster than full compilation.
- Another option is to only translate commonly used blocks to object code, and interpret the rest. This provides reasonable runtime performance with low start-up delay. Used by Sun's HotSpot Java compiler.

## Bottom Line: Programming is a *Human Activity*

High level languages are more convenient for humans to use than programming directly in binary code.

- A good programming language lets us work closer to the problem domain rather than machine-oriented concrete representations.
- Some languages are better than others.
- Most are arguably general purpose languages.
- Some are designed for a specific application domains (DSL: Domain Specific Language)
- There are (vastly?) more programming languages than useful programming language features.

## Perversions...

Some people (typically graduate students) enjoy subverting the structure and point of a compiler.

A favourite idea is to make the source LESS readable and convenient for humans.

For example: Shakespeare, LOLCODE, Brainf\*\*k, Intercal.

One of my favourites is...

## Whitespace

A Whitespace program consists only on spaces, tabs and newlines. All other characters are ignored (i.e. treated as comments).

The virtual machine is a simple stack-and-heap structure.

The virtual machine language has instructions like:

- push n
- swap
- store
- fetch
- jump label

(Note that there is no need for a concrete representation of the VML.)

A Whitespace fragment like:

which you probably can't see, corresponds to:

```
<tab><tab><space>
```

and is translated to the instruction store.

The Whitespace language is fully described at

<http://compsoc.dur.ac.uk/whitespace/>

Essentially, the source code is like a ternary encoding of the VML.

That's similar to writing binary machine code, then translating it to assembler counter to the point of a compiler.

## Exercises

- Check out the [Fortran for the IBM 704](#) manual on the course website.
- On page 17 it says “The statement `ASSIGN 12 to N` and the arithmetic formula `N = 12` are *not* the same”. Why do you think the language designers made this distinction? Is it a good idea? Why/Why not?