# 1 Formalisation

A formal system is a set of consistent rules that are susceptible to mechanisation.

Formal systems typically have *configurations* which we move between according to the rules. Board games such as Chess, Go and Draughts are all formal systems, and in fact we might think of formalisation as making a *game* which models a problem domain. Mechanisation does not necessarily mean that judgement becomes superfluous: to win we must choose the best move.

# It's languages all the way down

In *A Brief History of Time*, Stephen Hawking wrote:

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the centre of a vast collection of stars called our galaxy.
At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise."
The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

This *infinite regression conundrum* appears in many forms throughout history - see `https://en.wikipedia.org/wiki/Turtles_all_the_way_down` for further discussion.

Our topic is programming languages (not metaphysics) but we have own version of the conundrum.

# The Java translation stack

To run a Java program we:

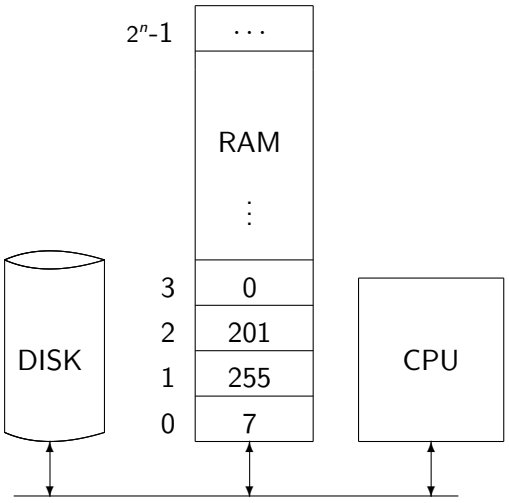
1. Translate the human-friendly Java source code into instructions for a pseudo-computer called the *Java Virtual Machine* using the `javac` compiler
2. Interpret the JVM instructions using the `java` JVM interpreter.

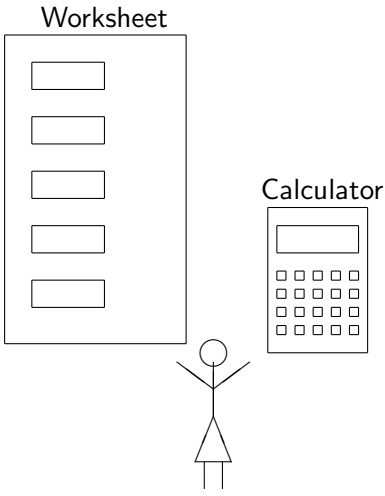
The JVM interpreter is itself a program, not an actual computer. The JVM has its own source code, and might even be written in Java...

Let's assume that it is written in C. Then the JVM source code will have been translated into the instructions for some real computer by a C compiler.

How do the instructions for the physical computer get executed? (Or, where do the turtles stop?)

# Real computers, automatic and manual



# The essence of computing

The human computer uses a four function calculator and a worksheet with formulae, jumps and boxes into which the computed results of formulae may be written.
The automatic computer has a memory into which both the computed results of formulae *and a representation of jumps and of the formulae themselves*, along with a central processor which can execute the calculator's functions and keep track of where we are in the program using a special variable called the *Program Counter* (PC).
Let's define these six instructions which capture the notions of typing a number into the calculator, jumps, and specifying one of the four functions the calculator provides.

| Action and code | | Syntax | | | | Effect on configuration |
|---|---|---|---|---|---|---|
| Set value | 1 | set | dst, | k | | $mc(dst) \leftarrow k,\ PC \leftarrow PC + 1$ |
| Branch if equal | 2 | beq | k | src1 | src2 | **if** $src_1 = src_2$ **then** $PC \leftarrow k$ **else** $PC \leftarrow PC + 1$ |
| Addition | 3 | add | dst, | src1, | src2 | $mc(dst) \leftarrow mc(src_1) + mc(src_2),\ PC \leftarrow PC + 1$ |
| Subtraction | 4 | sub | dst, | src1, | src2 | $mc(dst) \leftarrow mc(src_1) - mc(src_2),\ PC \leftarrow PC + 1$ |
| Multiplication | 5 | mul | dst, | src1, | src2 | $mc(dst) \leftarrow mc(src_1) \times mc(src_2),\ PC \leftarrow PC + 1$ |
| Division | 6 | div | dst, | src1, | src2 | $mc(dst) \leftarrow mc(src_1)\ /\ mc(src_2),\ PC \leftarrow PC + 1$ |

# Programs

The automatic computer's memory is a sequence of pigeonholes, each of which can hold a number which may be changed during the program's execution. The sequence number for each pigeonhole is called it's *address* and never changes.
We can represent programs as sequences of numbers held within the pigeonholes. For instance, here is Java (or C) fragment and its six-instruction-code program

```
a = 3; c = b = 2; while (a != 0) { c = c * b; a-- };
```

```
200: 1, 100, 0 // set location 100 to constant 0
203: 1, 101, 1 // set location 101 to constant 1
206: 1, 102, 3 // set variable a to 3
209: 1, 103, 2 // set variable b to 2
212: 1, 104, 2 // set variable c to 2
215: 2, 231, 102, 100 // branch if a=0 to location 231
219: 5, 104, 104, 103 // multiply b by c and store in c
223: 4, 102, 102, 101 // subtract 1 from 1 and store in a
227: 2, 215, 100, 100 // branch always to location 215 (since 0 = 0)
231: // The rest of the program...
```

# Is the six-instruction computer sufficient?

I claim informally that this six instruction computer can execute any Java program.

Logic operations such as `a && b` may be implemented by using constant zero to represent false, and any other value to represent true. We then use subtraction and branching instructions to compute the elements of the logic operation's truth table.

Shifts may be achieved by multiplying or dividing by two.

Array accesses need nasty trickery. Since all of our instructions have constant addresses built into them, when we compute an array access such as `x[y*2] = 7;` we need to calculate the address of the requested array element and then *write that address* into the set instruction `1, ?, 7`. This is *self modifying code* which was once common, but is now thoroughly deprecated because it makes programs very hard to read, and also disrupts the smooth execution of instructions on modern pipelined architectures.

# Are all six instructions necessary?

It is not hard to implement multiplication and division using only addition and subtraction, and in fact most computers sold before the mid-1980s did not have hardware multipliers or dividers.

We do not need the **set** instruction since zero is the identity under addition and subtraction, so instead of `set x, k` we could write `sub x, k, zero` where `zero` is the address of a location containing a constant zero (such as location 100 in the earlier example)

More interestingly, we do not need the add instruction either since

$$a + b = a - (-b) = a - (0 - b)$$

So, assuming that we have a spare location `tmp` available, we can replace every occurrence of `add x, a, b` with the sequence `sub tmp, zero, b; sub x, a, tmp`. Now we only have subtraction and branching left.

# Cheap computing

If resources were very scarce (perhaps we are a 1940s computing pioneer) or if we enjoy intellectual parlour games, we can go one step further and make the program counter appear as one of the pigeonholes in our memory.

This would allow calculations to directly write the address of the next instruction to be executed into the program counter and so we do not even need a branch instruction. Now we have a single-instruction computer: everything is a subtraction.

Many early computers were *bit serial*. That means that, say, a 16-bit subtraction was computed one bit at a time, working across the operands from least significant to most significant bit.

I know how to make such a CPU using only 30 switching elements and some one bit memories, but that is a story for another module.

# Expensive computing

These minimal computers are fun to design, but not much fun to program. Broadly speaking, the first forty years of computer architecture research focused on closing the so-called *semantic gap* between human-friendly high-level programming languages, and the low level *machine code* by adding new, expressive hardware instructions.

An important part of those developments was the use of *addressing modes* that allowed, for instance, an entire array index calculation to be performed during the fetching of an instruction's operands – the constant address of our instruction sets was expanded to allow quite baroque expressions.

In the mid-1980s a revolution occurred. Called *Reduced Instruction Set* (RISC) architecture (though I think *Reduced Addressing Mode* (RAMP) would have been a more accurate acronym), this new style eschewed any feature which made it hard to *pipeline* operations. Examples include MIPS and ARM: the only pre-RISC architecture being commercially developed is the Intel ISA x86 line.

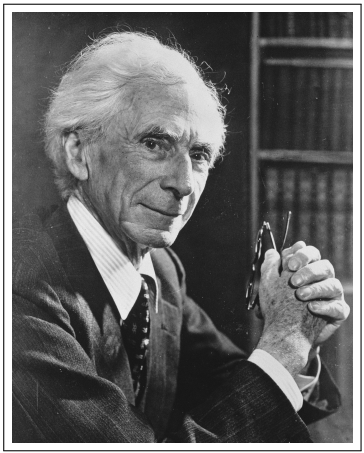
# Languages all the way down... with the hardware as a base case

As computer scientists we know that the solution to recursive conundrums like the turtle model of reality is to have a *base case*

A hardware instruction set like the ones we have just discussed forms the base case for our stack of translations. Computer architecture research is about finding the best base case. The definitive textbook is Hennessy and Patterson[1] but that is a story for a different module.

What we have seen, though, is that we can expand and contract the capabilities of the hardware without losing generality, and that we can translate programs from, say, our six instruction architecture to our four, two or one instruction architectures, easily and *without changing the upper translation layers at all*.

[1] https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1



Bertrand Russell 1872–1970

https://en.wikipedia.org/wiki/Bertrand_Russell

# Utility and power in languages and their processors

We now have a suspicion that our programming languages might somehow all be the same, in that we can translate between them. If that is so (and we shall return to that question later), why are there so many languages? I think there are four main drivers.

**Culture** Human (natural) languages seem to develop alongside cultures - groups of people working and living together share a common language which over time can develop into a separate dialect. ANSI C, ANSI C++ and even Java might be seen as derivatives of the original C.

**Commerce** Sometimes manufacturers attempt to lock customers in with proprietary languages. This was common in the early years, and still happens at the fringe: for example Nvidia's CUDA libraries require their hardware.

**Coolness** Fashion can dominate decisions - programmers love the latest cool thing, especially if it is not completely different to what they already know.

**Conciseness** Verbose programs are tiring programs, and much of programming language history can be seen as the pursuit of elegant notations to express complex notions. Of course, extremely concise notations can be tiring too (which is perhaps why many folk are put off mathematics).

# Reuse encourages layering

New software products are built on the foundations of existing systems, since *ab initio* implementation of, say, network protocols, encryption standards and graphics systems in machine code would be immensely costly.

This reuse might be in the form of libraries for use with an existing language, or it might involve a complete new *Domain Specific Language* (DSL)

For instance, a game engine might have its own scripting language. The script interpreter could be written in one architecture's machine code, but then it would only run on that architecture. A portable script engine could be written in Java. Now we have an interpreted language being executed by a program written in Java, which is itself being interpreted...

That game might also save configuration data in XML, access a database using SQL, use HTML5 graphics in the browser, and have a manual written (like these slides) in a typesetting systems such as LaTeX.

We have an entire ecosystem of interacting DSLs

# Internal and external DSLs

Libraries such as the Java FX graphics subsystem or the Java MIDI synthesizer offer a well defined set of capabilities that address specific domains - graphics and music-making in these cases.

We can think of each library as a Domain Specific Language in which the features are presented as method calls to Java objects, and we can build applications by writing Java programs that string together lots of calls to these libraries with general purpose Java control flow code.

Alternatively, we could make up a new syntax with its own control flow and data declarations, and hide all the (sometimes verbose) details of the Java scaffolding.

The first approach – a library called from a general purpose language - is sometimes called an *internal* DSL. The full-fat version with its own syntax and processor is called an *external* DSL.

We can view the external DSL as being a wrapper around the underlying internal DSL: the purpose of the external DSL's special language is to provide *concise* and sometimes *cool* ways of specifying applications.

# The DSL maintenance trap

I left out one common source of new languages in the first slide: ego.

Programming languages are fascinating artifacts, and many programmers have explored creating their own language. The Python and C++ languages emerged from experiments by individuals.

Reasoning about the interaction of language features is *hard*, especially when trying to extend an existing design.

Sometimes a hot-shot programmer within an organisation is a bit bored and decides to pressure management to allow a new DSL to be developed in which to encode their business processes.

Management wants to retain their expensive hot shot programmer who is thus allowed to build something, usually using tools and techniques that were optimised for 1970s architectures with tiny memories, and with a sticky mess of glue logic to paper over the cracks.

Eighteen months later the hot-shot programmer leaves anyway, and the company is left with an opaque system at the heart of their business that other programmers are frightened to meddle with.

# The DSL maintenance solution

There is another way

This course will teach you how to specify languages and their processors with concise very-high-level notations that allow you to:

- experiment with, and reason about, the interactions of language features,
- provide clear specifications to the engineers that have to maintain your work after you have moved on, and
- as a bonus, allow complete language interpreters to be generated automatically from your specifications which you can use for testing, and which might be fast enough for production use.

We hope for concise, complete and directly executable language specifications

Alfred North Whitehead 1861–1947

https://en.wikipedia.org/wiki/Alfred_North_Whitehead

# What do we mean by *formal*?

The history of science across the twentieth century is one of increasingly rigorous mathematical modelling.

In Physics we have the development of the Bohr atomic model, leading to quantum theory, Quantum Electrodynamics and the standard model of particle physics.

In Biology we have the development of chemical bond modeling leading to the elucidation of the structure of DNA and molecular biology.

In the future, medicine will increasingly be based on the application of statistical machine learning techniques to the unravelling of biochemical pathways, and their impact on the organism leading to new understanding of disease and opportunities for therapeutic intervention.

# Formalising mathematics

Mathematics itself is a discipline that benefits from mathematical analysis. Up until suprisingly recently, all mathematics was expressed in prose which is hard to analyse. For instance, the now universally-understood symbol for equality $=$ was only introduced in the mid-sixteenth century, during the reign of the English Queen Elizabeth I, and it was a further 150–200 years before it's use became universal.

As mathematical language moved from (rather fuzzily defined) words to symbols, some people began to wonder whether mathematics is in fact anything more than the manipulation of symbols.

The so-called formalist school of thought has it that all we have are strings of symbols which can be manipulated according to certain rules. The idea is that mathematical statements are simply syntactic forms which have no meaning unless they are given a semantics.

Philosophers of mathematics do not all agree. You can read about some of the debates here `https://en.wikipedia.org/wiki/Formalism_(philosophy_of_mathematics)`

# Formalising the notion of computation

Attempts to define computational tasks as well-defined procedures are very old. We shall study Euclid's 2,300 year old Greatest Common Divisor algorithm in a later section. Attempts to actually automate computation (as opposed to designing aids to for human computers) date to Babbage's multiple implementations of the Method of Differences, beginning 200 years ago.

*Limits* to computation became a topic of great interest in the 1920's. Hilbert, one of the most prominent of the formalists, set a challenge in 1928 called the *Entscheidungsproblem* or *decision problem*. One interpretation of this problem is that it asks whether there are statements in a logic that cannot be deduced from the axioms by any algorithm.

To answer the question, a formal model of 'algorithm' was needed. Alonzo Church's 1935 $\lambda$-calculus and Turing's 1936 *Turing machine* are equivalent minimalist models of computation which were used to show that *undecidable* decision problems exist.

More at https://en.wikipedia.org/wiki/Entscheidungsproblem

# Programming languages and the Entscheidungsproblem

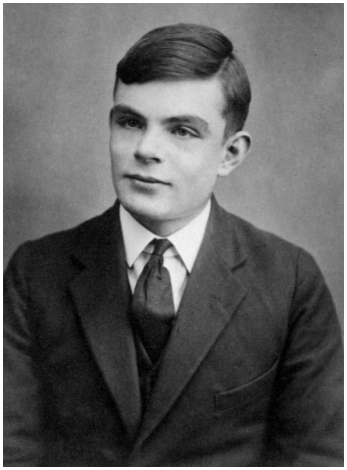
So, a Turing machine is a formal *game* with a configuration and transition rules that, according to the Church-Turing thesis, can perform any computation that a more complex device can. We could view it as an *abstraction* of our real computers, though of course it predates their design.

A key result of Turing's is that there exist *Universal Turing Machines* which can simulate any other Turing machine. They work by essentially reading in a description of the machine to be simulated, and then interpreting the input under those rules.

Many authorities believe that this notion directly inspired Von Neumann description of computers. Others recognise its significance, but believe that pragmatic, engineered stored-program computers developed independently of these theoretical formalisms – certainly Maurice Wilkes told me over dinner one night that his team knew nothing of Turing's formalisms.

Whatever the truth, it is the formal proof of existence of UTMs that 'explains' our insight that programming languages equipped with basic flow control are equivalent, in that one can be translated into another, or simulated in another.

More at https://en.wikipedia.org/wiki/Universal_Turing_machine



Alan Turing 1912–1954

https://en.wikipedia.org/wiki/Alan_Turing

# Conway's Game of Life, formalised

Conway's Game of Life (CGL) was popularised by Martin Gardner in the October 1970 edition of Scientific American

It is a game with no 'players'. The board is ordinary square ruled graph paper. Each square on the paper may be filled or it may be empty.

The game proceeds in generations: an entire board is evaluated, and the rules of the game specify which squares will switch from filled to empty, which from empty to filled, and which will stay the same.

There is no end point: the generations continue to evolve without limit, although they may converge to constant or repeating patterns that will never develop new behaviour.

CGL is interesting because a set of very simple rules generates complex, semi stable populations. It is an example of *emergent behaviour*

# Cellular automata

CGL is an example of a cellular automaton. We'll now give some more precise technical language for talking about such automata.

A cellular automaton is a formal system comprising a set $\Gamma$ of cells, each of which must be labeled with an element from a finite set of states $\Sigma$, and a transition rule which specifies the evolution of the cell labels in discrete time steps $\Gamma_0, \Gamma_1, \ldots$ called generations.

It can be useful to think of $\Gamma$ as having structure, in the sense that some cells are 'adjacent', and for the transition rule to be a function over a 'neighbourhood' $f(p_1, p_2, \ldots, p_k)$ with signature $f : \Sigma \times \Sigma \times \ldots \times \Sigma \to \Sigma$ where the $p_k$ are the elements of the neighbourhood.

# The CGL transition rule

The blue paragraph on the previous slide is *woolly*: we have not explained how the neighbourhoods are defined or used, and as a result we could not automatically generate an implementation.

In CGL, the cells in $\Gamma$ are arranged as an unbounded two-dimensional rectangular array; $\Sigma$ is the set $\{0, 1\}$ and the transition rule defines the new state $\sigma'$ of a cell in terms of the its existing state $\sigma$ and the population $P$ as the sum of the states of its eight-connected neighbours according to this function:

$$\sigma' = \begin{cases} 1, & \sigma = 0, P = 3 & \text{(birth)} \\ 0, & \sigma = 1, P < 2 \vee P > 3 & \text{(death through loneliness or overcrowding)} \\ \sigma, & \text{otherwise} & \text{(stasis)} \end{cases}$$

This rule induces a *transition relation*: the set $\{(\Gamma_i, \Gamma_j)\}$ such that the transition rule can construct $\Gamma_j$ from $\Gamma_i$.

The CGL transition relation is a *function* because only one new board can be generated from each $\Gamma_i$. Contrast that with, say, chess in which there may be hundreds of moves from each position.

# Neighbourhoods

The transition rule is still woolly: what is the meaning of *eight-connected neighbours*? Geometrically, we mean that given the plane tesselated by rectangles, the 8C-neighbours of cell $C$ are the eight cells that touch $C$, including four that only touch at their corners.

So if we are looking at the cell at coordinates $(x, y)$, then we want to count the total population of the cells at locations (x-1,y-1), (x,y-1), (x+1,y-1), (x-1,y), (x+1,y), (x-1,y+1), (x,y+1), (x+1,y+1)

A more concise way of saying this is

$$P_{x,y} = |\Gamma_{x,y}|$$

where

$$\Gamma_{x,y} = \{(x+i, y+j) \mid (x,y) \in \Gamma, -1 \leq i,j \leq 1, (i \neq 0 \wedge j \neq 0), (x+i, y+j) \in \Gamma\}$$

# Translation to Java

Substituting $\Gamma_{x,y}$ into the expression for $P_{x,y}$, we have

$$P_{x,y} = |\{(x+i, y+j) \mid (x,y) \in \Gamma, -1 \leq i,j \leq 1, (i \neq 0 \wedge j \neq 0), (x+i, y+j) \in \Gamma\}|$$

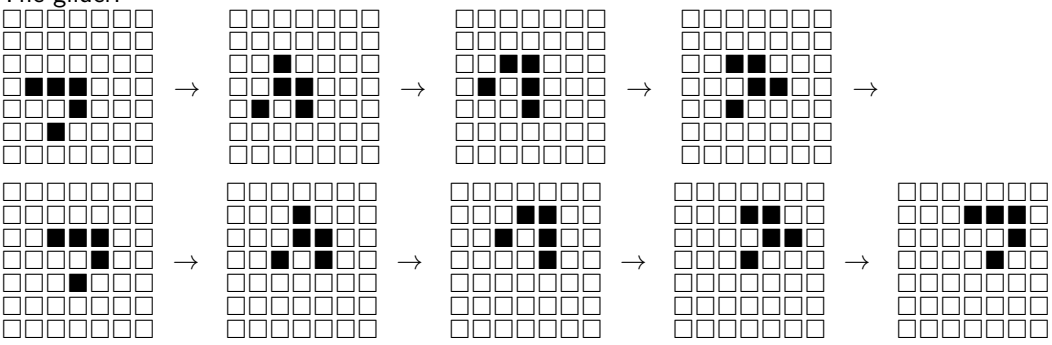
and here is a fairly direct translation from the formal language of set theory, to a formal language of programs (using Java in this case).

```
1  int P(x,y) {
2    int ret = 0;
3    for (i = −1, i <= 1, i++)
4      for (j= −1, j <= 1, j++)
5        if (i != 0 && j != 0) ret += gammaCellState(x + i, y + j);
6    return ret;
7  }
8
9  int gammaCellState(x,y) {
10   if ( /* The cell (x,y) is in Gamma */)
11     return 1;
12   return 0;
13 }
```

# Some CGL patterns

Block:  Beehive:  Blinker: 

The glider:



# Naïve implementation of CGL using an array

So far, our `gammaCellState()` function remains undefined. It depends on the representation we use for the Life board.

An obvious, though flawed, way to implement Conway's Game of Life is to use a two dimensional array of Boolean values or integers. Section 1.7.3 of the book gives a full implementation that you should study.

Here are some weaknesses of the array-based approach

- The size of an array is fixed at the time it is created
- The number of calls per generation-transition to the population function $P$ is the number of cells in the array. This means that the computational demand is $O(XY)$ where $X$ and $Y$ are the board has $X \times Y$ cells.
- Arrays have *edges* because the allowed values of $x$ and $y$ have finite bounds. However, our transition function is defined over the integers, and so we do not know what to do if the neighbourhood function wants to look up an element that is outside of the array. This this implementation is *incorrect*. Consider two gliders that fly outside of the finite bounds before colliding and sending a fragment back in: our implementation will diverge from the formal definition.

# A set based implementation without 'edges'

Life grids are often sparsely populated. We would like an implementation where the processing times is $O(|L|)$ where $L$ is the set of cells that are *live*.

We also want an implementation that avoids the finite bounds of the array. Of course, all real computer memories are finite so eventually we shall run out of memory, but we can do much better then the array representation.

In general, cellular automata can have multiple states, not just 'filled' and 'empty', so let us represent one generation of an automaton in the plane as a set of triples $\Gamma_i = \{(x, y, \sigma)\}$ with $\sigma \in \Sigma$, the set of possible states. We choose a particular element of $\Sigma$ to be the 'background' and omit those triples from our representation of $\Gamma_i$

For CGL, this reduces to representing the state of the board with the set of coordinate pairs of each live cell. You will find an implementation in this style in the book, which you should study. We extract an element from the current $\Gamma$, then check its 8-connected neighbours to see if it should live, and if any neighbour births occur.

# Using a visited set to avoid recomputations

We can go further. The set based implementation computes $\Gamma_{i+1}$ by extracting each element $(x, y)$ from $\Gamma_i$ and checking its eight connected neighbours for two purposes:

1. To decide if $P(x, y)$ is in the range $2 - 4$ in which case $(x, y)$ is entered into $\Gamma_{i+1}$
2. To check the eight connected neighbours of cell $(x, y)$ to see if they are not in $\Gamma_i$ and if so, if their neighbour population is 3 in which case a new cell is 'born'.

Now, an empty cell can be a neighbour of up to eight filled cells, and step 2 will be executed for each of those filled neighbours

We can save this computation by keeping a set $C$ of 'checked' cells and only executing step 2 on cells that are not in $C$. You will find a full implementation in the book.

This is an example of the technique called *memoisation* in some sources, in which we cache the result of calculations that might otherwise be performed multiple times.

# How is this relevant to programming language design and implementation?

There are two lessons from this section.

Firstly, *thinking about things in an abstract way can reveal insights*. By moving away from a geometric understanding, and instead representing each CGL configuration (generation) as a set of tuples we were able to find implementations that were more efficient in both time and space.

Secondly, the idea of a system that has *configurations* which we step between under the control of a *transition relation* is a widely applicable formal notion that is susceptible to mechanisation.

We can use the transition relation idea to think about automata, such as the deterministic and nondeterministic finite automata that are studied in the first year of most computer science degrees, as well as network protocols and other more complex patterns of behaviour.

# Programming languages as transition relations

There is a sense in which a programming language, such as Java, defines a transition relation between *configurations* of a computer.

We think about these configurations as the combined state of *all* of the mutable memory elements of the computer by which all of the main RAM memory, the complete state of the disks, the program counter and every other mutable memory element within the system - its a bit like an enormous CGL board with one cell for each memory bit.
As we execute lines of a Java program, the machine configuration changes, and in fact since our configuration includes *all* of the mutable memory elements, the evolving sequence of configuration *completely* defines the behaviour of the machine.
The Java transition relation includes $(C_i, C_j)$ iff $C_i$ is a machine configuration generatable by some Java program, and $C_j$ may be reached from $C_i$ by executing one Java 'step' – say the evaluation of a single operator.

If a program is completely sequential, then there is only ever at most one successor configuration available, that is the transition relation is a partial function. For a concurrent program or a non-deterministic sequential program, then there may be multiple successors. So sequential programs have transition functions as does CGL, and concurrent programs look more like chess with multiple possible 'moves' to multiple possible successor configurations.

John Horton Conway 1937–2020

https://en.wikipedia.org/wiki/John_Horton_Conway

# The rewrite model: Euclid's Greatest Common Divisor

In volume 2 of his *magnum opus* The Art of Computer Programming, Donald Knuth calls Euclid's Greatest Common Divisor the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.

The algorithm appears in Book VII of Euclid of Alexandria's *Elements* which was probably written around 2,300 years ago. It is not clear if Euclid collected previous results, or originated them, and in fact it is not entirely clearly that Euclid was a real person... but *Elements* set a pattern for formal presentation of mathematical arguments that still is widely used today.

For our purposes, it has the great merit of being the shortest program I know of that actually does something useful and has non-sequential control flow. We shall use it as a running example.

# The text of *Elements*

https://www.claymath.org/library/historical/euclid/files/elem.7.2.html

**Book VII, Proposition 2**

Given two numbers not prime to one another, to find their greatest common measure.

| | |
|---|---|
| Δύο ἀριθμῶν δοθέντων μὴ πρώτων πρὸς ἀλλήλους τὸ μέγιστον αὐτῶν κοινὸν μέτρον εὑρεῖν. Ἑστωσαν οἱ δοθέντες δύο ἀριθμοὶ μὴ πρῶτοι πρὸς ἀλλήλους οἱ ΑΒ, ΓΔ. δεῖ δὴ τῶν ΑΒ, ΓΔ τὸ μέγιστον κοινὸν μέτρον εὑρεῖν. Εἰ μὲν οὖν ὁ ΓΔ τὸν ΑΒ μετρεῖ, μετρεῖ δὲ καὶ ἑαυτόν, ὁ ΓΔ ἄρα τῶν ΓΔ, ΑΒ κοινὸν μέτρον ἐστίν. καὶ φανερόν, ὅτι καὶ μέγιστον: οὐδεὶς γὰρ μείζων τοῦ ΓΔ τὸν ΓΔ μετρήσει. Εἰ δὲ οὐ μετρεῖ ὁ ΓΔ τὸν ΑΒ, τῶν ΑΒ, ΓΔ ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεταί τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται: εἰ δὲ μή, ἔσονται οἱ ΑΒ, ΓΔ πρῶτοι πρὸς ἀλλήλους: ὅπερ οὐκ ὑπόκειται. λειφθήσεταί τις ἄρα ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. καὶ ὁ μὲν ΓΔ τὸν ΒΕ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΕΑ, ὁ δὲ ΕΑ τὸν ΔΖ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΖΓ, ὁ δὲ ΓΖ τὸν ΑΕ μετρείτω. ἐπεὶ οὖν ὁ ΓΖ τὸν ΑΕ μετρεῖ, ὁ δὲ ΑΕ τὸν ΔΖ μετρεῖ, καὶ ὁ ΓΖ ἄρα τὸν ΔΖ μετρήσει: μετρεῖ δὲ καὶ ἑαυτόν: καὶ ὅλον ἄρα τὸν ΓΔ μετρήσει. ὁ δὲ ΓΔ τὸν ΒΕ μετρεῖ: καὶ ὁ ΓΖ ἄρα τὸν ΒΕ μετρεῖ: μετρεῖ δὲ καὶ τὸν ΕΑ: καὶ | Given two numbers not prime to one another, to find their greatest common measure. Let AB, CD be the two given numbers not prime to one another. Thus it is required to find the greatest common measure of AB, CD. If now CD measures AB—and it also measures itself—CD is a common measure of CD, AB. And it is manifest that it is also the greatest; for no greater number than CD will measure CD. But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it. For an unit will not be left; otherwise AB, CD will be prime to one another [VII. 1], which is contrary to the hypothesis. Therefore some number will be left which will measure the one before it. Now let CD, measuring BE, leave EA less than itself, let EA, measuring DF, leave FC less than itself, and let CF measure AE. Since then |

# The GCD algorithm as a program

```
1  a := input();
2  b := input();
3
4  while a != b
5      if a > b
6          a := a − b;
7      else
8          b := b − a;
9
10 output(a);
```

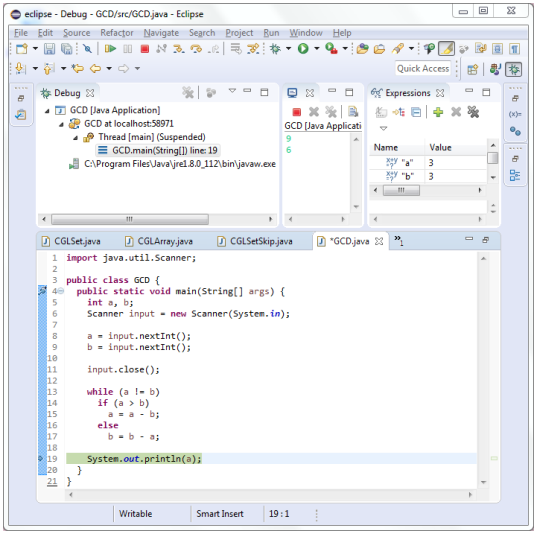
# GCD in Java

```
1  import java.util.Scanner;
2
3  public class GCD {
4     public static void main(String[] args) {
5        int a, b;
6        Scanner input = new Scanner(System.in);
7
8        a = input.nextInt();
9        b = input.nextInt();
10
11       input.close();
12
13       while (a != b)
14          if (a > b)
15             a = a - b;
16          else
17             b = b - a;
18
19       System.out.println(a);
20    }
21 }
```

# The fixed-code-and-program-counter interpretation

When we are developing software, we write code, load it into a development environment such as Eclipse and then run it to see if its behaviour matches our expectations. Here is a screenshot of the Java CGD program being run under the debugger within Eclipse.

We have the Java code, the input $[9, 6]$ (in green) and the state of the store with variables a and b, both presently mapped to the value 3. The program has stopped just before executing the output statement line 19 of the code window has a small pointer arrow on the left hand side and is highlighted.



# Rewriting – another way to think about program execution

Here is a short program.

```
1 output(3);
2 output(10+2+4);
```

The first thing the program does is output the value 3. We can represent this by constructing an output *list* [3] and then discarding the first line of the program (since we do not need it again). There is a sense in which the tuple

⟨"output(3); output(10+2+4);", []⟩

means the same thing as

⟨"output(10+2+4);", [3]⟩

because the externally visible effect of starting with an empty output and executing lines 1 and 2 above is the same as starting with the output [3] and only executing line 2. We can represent each step of a program's execution by a pair comprising the current output and a rewritten program that represents only what remains to be done.

```
1 output(10+2+4); [3]
```

Now we have to evaluate the expression 10+2+4 before we can execute the next output statement. In detail, the computer can only execute one arithmetic operator at a time; let us choose to execute 10+2, and rewrite it to the result 12.

```
1 output(12+4); [3]
```

Now we do the other arithmetic operation: 12+4 is rewritten to 16.

```
1 output(16); [3]
```

Finally we can execute the output statement, and add 16 onto the end of the output list.

```
1 [3, 16]
```

# Rewriting vs. using a program counter

The standard approach – using an unchanging program and a Program Counter that points to the next instruction to be executed – is efficient and effective, and basis of almost all hardware implementations of computers.

The rewriting approach is much more amenable to mathematical analysis. This is because when we want to, say, prove that a program terminates, we can use *proof by induction on the length of the program*. Induction proofs typically need some measure that is increasing or decreasing built into the induction hypothesis.

A closely related technique is *structural induction* in which we show that the effect of a nested structure (perhaps nested control flow in Java) is the union of the effects of its component parts.

We are not going to be proving program properties on this course. However we are going to build syntax analysers and interpreters *compositionally*, for which rewriting can support clear, uncluttered reasoning, producing short, maintainable specifications and for which structural induction proofs could, in principle, be constructed.

A reduction semantics for linear code is straightforward, but we need to think carefully about loops. One approach is to use this *program identity*

```
1 while booleanExpression do statement;
```

is *always* the same as

```
1 if booleanExpression { statement; while booleanExpression do statement; }
```

We have effectively unpacked the first iteration of the loop and are handling it directly with an **if** statement followed by a new copy of the `while` loop which will compute any further iterations. When we have completed all of the iterations we shall encounter a term like

```
1 if false { statement; while booleanExpression do statement; }
```

which rewrites to the empty subterm. This device, then, allows us to treat **while** loops using only **if** statements.

# A rewrite evaluation of GCD with input [6, 9]

Procedural programs typically have input, output and variables. We use lists of values to represent the input and output, as we have already seen. These lists are called *semantic entities*, and their rôle is to leep track of *side-effects* of the program whilst it is being rewritten away.

To represent the variables, we can use a *map*. A map is a set of bindings. In Java we assign, say, 6 to variable a by writing `a = 6;`. When we come to reduce that piece of code, we create a binding $a \mapsto 6$ in the store, giving the set of bindings $\{a \mapsto 6\}$

A *rewrite configuration* is a tuple comprising the program term to be rewritten and all of the semantic entities with their current values. The set of semantic entities is fixed for a given programming language. For a purely functional programming language, there are *no* semantic entities because program rewriting is all that they allow.

For our GCD program, configurations will be of the form $\langle \alpha, \sigma, \beta, \theta \rangle$ to represent the input, store, output and program term, respectively.

The 36 following steps completely execute GCD for inputs 6 and 9. Each step *roughly* corresponds to one machine instruction.

**Start of trace**

**Initialise variables from input**

$\langle [6,9], \{\}, [\,], \texttt{a:=input(); b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [9], \{a \mapsto 6\}, [\,], \texttt{b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

**Rewrite using** `while p s` $\rightarrow$ `if p { s ; while p s }`

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{while a!=b if a>b a:=a−b; else b:=b−a;output(a);}\rangle$

**Evaluate** $a \neq b$ **with store** $\{a \mapsto 6, b \mapsto 9\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$
`if a!=b { if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; } output(a);`$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$
`if 6!=b{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; } output(a);`$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$
`if 6!=9{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; } output(a);`$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$
`if true { if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; }output(a);`$\rangle$

**Evaluate** $a > b$ **with store** $\{a \mapsto 6, b \mapsto 9\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$
`if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);`$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **if** 6>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **if** 6>9 a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **if false** a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
**Evaluate** $b - a$ **with store** $\{a \mapsto 6, b \mapsto 9\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=b−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=9−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
**Rewrite using** `while p s → if p { s ; while p s }`
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **while** a!=b **if** a>b a:=a−b; **else** b:=b−a;output(a);$\rangle$
**Evaluate** $a \neq b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
**if** a!=b { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
**if** 6!=b{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
**if** 6!=3{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
**if true** { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
**Evaluate** $a > b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **if** 6>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **if** 6>3 a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **if true** a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
**Evaluate** $a - b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=a−b; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=a−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=6−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
**Rewrite using** `while p s` → `if p { s ; while p s }`
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **while** a!=b **if** a>b a:=a−b; **else** b:=b−a;output(a);$\rangle$
**Evaluate** $a \neq b$ **with store** $\{a \mapsto 3, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$
**if** a!=b { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$
**if** 3!=b{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$
**if** 3!=3{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$
**if false** { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ output(a);$\rangle$
**Evaluate output**
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ output(3);$\rangle$
**Empty term indicates normal (successful) termination**
$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [3],\ \rangle$
**End of trace**

Euclid of Alexandria (perhaps) 2300-2200 years ago?
https://en.wikipedia.org/wiki/Euclid

# Pragmatics

These books are useful compendia of topics in language design and compilation:

Programming Language Pragmatics
Michael L Scott, fourth edition, Morgan Kaufman, 2016

Modern compiler design
Dick Grune et al, Second edition, Wiley, 2016

Programming language design concepts
David A. Watt, Wiley, 2010

Design concepts in programming languages
Franklyn Turback and David Gifford, MIT Press, 2008

A now very dated, but in my opinion enjoyable and motivating book is:

The programming language landscape
Henry Ledgard and Michael Marcotty, Science Research Associates, 1981

# Further reading on machine level topics and software management

These two books are the standard texts on computer architecture and systems software:

Computer architecture: a quantitative approach
John Hennessy and David Patterson, sixth edition, Morgan Kaufmann,2017

Computer organisation and design
John Hennessy and David Patterson, sixth edition, Morgan Kaufmann,2020

Anybody who ever has to be a member of a software development should read this:

The mythical man-month
Fred Brooks, anniversary edition, Addison Wesley, 1995

# The architecture of language processors

We can identify four main parts of a language system

1. The *front end* (F) translates from external syntax to internal syntax
2. The *back end* (B) translates from external syntax to target syntax
3. The *middle end* (M) translates from internal syntax to internal syntax, transforming the program in some way
4. The *execution end* (X) executes the program, consuming data and generating the programs effects

There are at least three representations of the user program in use here: external (E), internal (I) and target (T). In the case of `javac`, the Java compiler, syntax E is the Java language itself, syntax T is Java byte code and syntax I is a property of the particular java compiler implementation. Some compiler toolkits, for instance the gcc toolchain, have multiple internal syntaxes, each optimised for a particular task.

# It's languages all the way along

We can visualise these four parts as a *translation pipeline*, each stage of which translates the user's initial program, perhaps producing some outputs as a side effect.

| Syntax | Stage | Side outputs |
|---|---|---|
| **E**xternal | *User program* | |
| | **Front end** | Syntax check |
| **I**nternal | *Internal representation* | |
| | **Middle end** | Call graph, cross refeference, lint and verify |
| **I**nternal | *Reworked internal representation* | |
| | **Back end** | Source-to-source transformations |
| **T**arget | *Translated program* | |
| | **Execution end** | Profile, coverage, program effects |

In detail, the middle end often involved multiple phases, each of which is an I-to-I translation

# Why four stages?

The four stage process arises naturally from a modularisation based on dependencies.

The front end is dependent only on the external syntax and outputs the internal syntax

The middle end is independent of external and target syntax

The back end is dependent only on the target syntax and the internal syntax

The execution end is dependent only on the target syntax

This separation of concerns allows external and target independent middle ends to be developed, which can then be connected to multiple front and back ends. So the GNU compiler collection includes front ends for C, C++, Go, Pascal, Modula-2, FORTRAN, Ada and others. GNU has supported over 50 different machine architectures over its development history.

Detailed code analysis and rearrangement so as to improve speed or compactness of the generated code resides in the middle end, so having a machine and programming language independent middle end allows reuse of significant intellectual property.

# Do we always need four stages? The single pass compiler

## In some use cases we do not need all four stages

It is perfectly possible (for certain restricted classes of programming language) to emit machine code for a real computer architecture as a side effect of the parsing process, with no discernible middle- or back-end. In fact nearly all languages designed before the mid-1990s support this so-called *single pass compiler* architecture.

The defining feature of 'single pass friendly' languages is that variables and procedures must be declared before use. ANSI-C and Pascal are good examples of this phenomenon. It cause particular problems with recursive functions, whose definitions cannot be completed before they are used. To handle such cases, both languages allow a signature to be declared without a corresponding function body.

Such compilers usually generate rather inefficient executable code, but they can be run on very small memory machines with only a few thousand memory locations. They can also be very fast - the 1980s Turbo Pascal compiler wasa very popular development environment for IBM PCs.

# Do we always need four stages? The assembler

Until highly portable languages such as Java appeared in the mid 1990s, most working software engineers would have had the skills to drop down into machine language if they wanted to speed up a critical piece of code. In our terms, this means writing programs directly in syntax T.

To a hardware engineer, syntax T comprises bit strings, and humans are not very good at distinguishing bit strings by eye, so very early on in the development of computers (around 1947 and 1948 in Birkbeck College London and at the Mathematical Laboratory in Cambridge) a programming style developed in which short mnemonics such as ADD and JMP were defined for each discrete machine instruction, along with a facility to associate an alphanumeric name with a machine address. This allowed programmers to write, for instance, `ADD X,Y,Z` which could easily be turned into a bit string by looking up the bit-string values of A, X, Y and Z, and then concatenating them together into a binary machine word. The style of programming language is called an *assembler*; and the reverse lookup provides a *dissassembly* when examining machine instructions.

Nearly all execution environments provide some sort of assembly and dissassembly. The `javap` command dissasembles JVM byte code from .class files.

# Do we always need four stages? The interpreter

Programs in languages other than assembly language first began to emerge in the mid-1950s. So-called *autocodes* were developed for the Manchester Mk 1 and later Ferranti machines also associated with Manchester University and in Cambridge for EDSAC. These systems typically allowed individual expressions to be compiled into subroutines, and were the beginnings of portable programs that could run on more than one machine architecture.

Language development then took two paths for two user communities. The FORTRAN language (1957) aimed to produce code which was fast enough to persuade assembly language programmers to learn the new technique.

Universities were interested in teaching programming to engineers and scientists, and needed simple languages that could run on low performance systems with instant feedback to the user. BASIC (Beginners All Purpose Symbolic Instruction Code) (1964) was designed to be executed line-by-line in an interactive environment. No syntax T program is produced: instead the middle end has a routine for each language feature which is activated as each line is parsed: this style is called an *interpreter*.

# Do we need more than four stages? The Just-In-Time compiler

One way to achieve high portability but still allow significant code improvements to be applied to the user's program is to compile to a *virtual machine*. The UCSD P-system (1977) was a response to the plethora of new microprocessor architectures being developed at that time. By compiling to *P-code* and then having an efficient interpreter which could be easily implemented in, say, assembly language one could quickly get Pascal running on a new architecture.

The same approach was taken by the designers of Java (1995) which was originally intended for embedded systems in which code would be compiled on a general purpose machine, but run on processors with very restricted resources.

Early Java systems were very slow compared to C code compiled to native instructions. Modern systems are often within a small integer factor of the performance of C due to the development of *Just-In-Time* techniques in which the interpreter monitors *hot spots* that are frequently executed, and can stop and compile those to native instructions. Note that this is a T to T' translation, where T is JVM byte code and T' is the native instruction set of the host architecture.

# Tooling use cases

1. Syntax checker (F)
2. Cross referencer, call graph generator, linter, 'bad smell' detector (FM)
3. Code verifier against some high level specification (FM)
4. Pretty printer, minifier, normaliser (FMB, but with syntax T=E)
5. Test case generator (FMB, but with syntax T=E)
6. Live editor with immediate feedback to the programmer (all of the above, running in the background of an editor)
7. Interpreter (FMX)
8. Single pass compiler (FX)
9. Optimising compiler (FMBX)
10. Just-in-time compiler (FMBX(B'X)*)
11. Reverse compiler (FMB, but with syntax E being low level (eg Java byte code) and syntax T being high level (eg Java))
12. Profiler (FMBX)
13. Coverage analyser (FMBX)

# The landscape of programming languages

Most languages evolve from earlier attempts, but occasionally radical new ideas emerge. FORTRAN was the first complete programming language that could run on more than one architecture. COBOL was the first attempt to converge programming language syntax with non-scientific prose (SUBTRACT A FROM B GIVING C). LISP (1958) was the first functional language (although LISP as we know it was intended only to be the internal syntax for a more baroque external syntax!)

Algol was the first widely used language to allow user defined types, recursion and block structured control flow. Simula-67 introduced object orientation. Smalltalk took the object idea to the limit - everything in Smalltalk-80 is an object including control flow structures.

SETL (1969) introduced *comprehensions* which now appear in many functional languages, and more recently Python. ML (1973)and its precursors introduced the notions of *type inference*, *type variables*, algebraic types and pattern matching. Rust (2015) provides a type system that guarantees certain safety properties.

# How not to design a successful language

Notions are more important than notations. Notation matters though!

When programmers start thinking about a new language there is a strong tendency to focus on external syntax by finding neat ways to capture common idioms. Now, it turns out that a clean and unified design is more likely to emerge from a consideration of abstract language features which are then fitted into an external syntax.

This is why on this course we have focused on internal syntax first – in our prefix style we only have to decide on set of features, their names and their arities and then we have the internal syntax defined. With it we can start to write semantic descriptions and discover unpleasant inconsistencies and interactions using a very simple regular syntax which supports reasoning. Once the design has settled down, we can 'decorate' the internal syntax by developing an external syntax that admits straightforward translation to our internal semantics-friendly notation.

# The perfect language may not exist

It is conceivable that at some point in the future, all programs will be written in a single language. After all, we are still quite early on in the evolution of computing systems, and are only just emerging from the frenetic era of exponential hardware improvement during which Moore's 'law' has delivered a doubling of speed and memory density every 18 months for many decades.

However, at the moment new ideas and ways of implementing them continue to surface in experimental languages.

Programming languages and their ecosystems have lifecycles, just like any other kind of software. Many modern features such as generics, lambdas and limited type inference have been retro-fitted into Java's initial core, just as object oriented programming was retro-fitted to ANSI-C. It is likely that in the end these languages will be superceded by more elegant, integrated designs

# Programming language facets

1. Names
2. Atomic values and types
3. Binding time and scope regions
4. Expressions
5. Sequencing, selection, iteration and call/return
6. Statements
7. Data abstraction
8. Functional and procedural abstraction
9. Packaging
10. Concurrency and non-determinism
11. Exceptions
12. Meta-programming and reflection

# Selected ideas in programming languages

1. Throw away the syntax – Scheme, Forth
2. Orthogonality vs one-way-to-say – Algol-68, Lua, Pascal
3. When to bind – Python (dynamic) vs Occam (very static)
4. Organising arguments – keyword arguments vs signature ordering
5. Algebraic datatypes and pattern matching
6. Collection comprehensions – set builder notation
7. Making functions first order – lambdas
8. Making types first order – generic vs type variables
9. Pragmatic types – lengths do not add to areas; ranging
10. Type inference
11. Throw away the control flow - continuations
12. Concurrent friendly notions – map/reduce, array operations

# A Zoo of specification and implementation technologies

1. Deterministic lexing parsing and parsing
2. General lexing and parsing
3. Intermediate form languages
4. Structural Operational Semantics
5. Other approaches to formal semantics
6. Control flow analysis
7. Dataflow analysis
8. Traditional optimisations
9. Static scheduling
10. Dynamic scheduling