Computational Thinking

Topic 2

What is computer software?

2.2 Introduction

There are many different programming languages in existence. A *Wikipedia* page (see [9]) lists 640 (what it calls) 'notable' programming languages, and for every letter of the alphabet there are at least 3 programming languages whose names start with that letter (the letter 'C' begins the names of 68 different programming languages!). Some programming languages are what might be termed general-purpose languages, designed for writing software for use in a wide variety of domains (Python falls into this category), whereas some have been designed with a specific purpose in mind, such as the hardware description language Verilog or the database query language SQL.

The popularity of a programming language can vary dramatically across different groups of users; for example, functional programming languages, like Haskell, are extremely popular amongst academic researchers whereas they have never really been taken up in industry. This might be considered quite surprising as it is (almost always) the case that if we pick two programming languages at random then we can do in one what we can do in the other (there are, though, other reasons for a predilection towards some programming language).

We're now going to look at some different types of programming languages, and how we say what a program is and what it does (that is, provide its syntax and its semantics). We'll see that getting a program into a form understandable to the computer can be a complicated business! We'll begin by looking at the difference between an algorithm and a program (we'll focus on programs in this topic with algorithms coming later) before looking at a variety of different programming paradigms and the ethos of each paradigm. We'll see that there are lots of paradigms and lots of programming languages and we'll look at some of the drivers as regards programming language evolution. We'll take a closer look at one programming paradigm, namely logic programming. Fundamental to any programming language is its syntax and its semantics and we'll see how the syntax and semantics of a programming

language are used to compile or translate a program into something that can be executed by the CPU. We'll examine the compilation process and see how algebraic formalisms such as regular expressions, finite state machines, regular grammars and context-free grammars are all fundamental to this process.

2.3 Layers of abstraction in modern computer systems

Recall our abstraction of a modern computer system, as visualized in Fig. 1. Our focus in what follows is on the level labelled 'programming languages'. We can see from the diagram that there will still be work to do in order to bridge the gap between the program, expressed using one of our 'high-level' programming languages, and the computer hardware (and, in particular, the CPU which does all of the hard work). The words 'high-level' in this context are usually taken to mean that the programming language has been designed with ease of programming in mind rather than ease by which the programs can be made machine-executable.

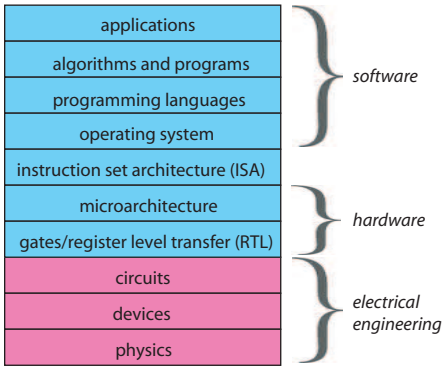

Fig. 1: layers of abstraction in a modern computer system.

> ***'using abstraction and decomposition in tackling a large complex task'***

2.4 Algorithms and programming languages

A **programming language** essentially provides a means for transforming an **algorithm** into a form suitable for execution by a computer; or, more

precisely, into a **program** that can *then* be transformed (trivially) into a form suitable for execution by a computer. We'll look at some algorithms in more detail later but in general they are rather elusive and informal objects; we can think of them (rather vaguely) as just sequences of precise instructions that can be applied to specific data items. Typical concepts associated with algorithms are that:

- an algorithm works on some **input**;
- an algorithm produces some **output**;
- the steps of an algorithm are **precisely stated**; and
- an algorithm can be applied **generally** to a variety of inputs.

An algorithm is *not* a computer program. A computer program is something written in a (concrete and explicitly defined) programming language, and an algorithm can have many different **implementations** as a program (even within the same programming language!). Giving a formal definition of what an algorithm is leads to a 'Catch-22' situation: if we can formally define an algorithm then what we have is a programming language; yet, an algorithm is supposed to be different from a program (something more ethereal and 'syntax-independent').

Don Knuth, a Turing Award winner (1974) and probably the father of the **analysis of algorithms** (that is, the study of algorithm performance), said (see [4]): '*An algorithm must be seen to be believed.*' (**Turing Awards** are Computer Science's Nobel Prizes, by the way). What Knuth mean't was that it's all very well studying an algorithm (theoretically) but to fully understand it we have to implement it (practically) as a computer program (or, more precisely, as computer programs as often there are lots of ways to implement an algorithm). Anyway, in what follows we're going to be dealing with programs and we'll leave the algorithms until later.

2.5 Programming languages

So, let's forget about algorithms for the moment and just work with programming languages and their programs. However, before doing so let us remark that there are two phases to converting an algorithm into something that the computer can execute: the first phase is describing (that is, implementing) the ('ghostly') algorithm using a programming language; and the second is

converting the resulting program into something that is machine-executable. The whole point of having all these different programming languages is so as to make the first phase as satisfactory as possible for different programmers working in different environments (we'll be more specific as to what we mean by 'satisfactory' in a moment). Nevertheless, as well as devising a programming language that is beneficial to programmers, the designers also deal with the second phase but so as to hide this phase from the programmer. As far as programmers are concerned, their programs magically execute on their computers; they even execute on *different* computers made by *different* manufacturers and running *different* **operating systems** (we'll look at operating systems later). As ever in Computer Science, there are lots of hard work, deep insights and some very clever people behind this 'magic'!

Programming languages can be of different **programming paradigms**; that is, of different styles, where the style is dictated by such things as the 'concepts' and 'abstractions' used to represent the elements of a program. Programming paradigms include the following.

- **Imperative** (also known as **procedural**), e.g., Python, Ruby, C, Fortran, C++, C♯, Java, Ada, Javascript, PHP, ...
  Imperative programs are characterized by their use of programming instructions to explicitly change a program's **state**; that is, the current status of entities within the program, such as variables (more precisely, the state is given by the different values held in the different memory locations). The imperative style is closest to the abstraction of computer hardware as a bunch of memory locations the values of which are explicitly manipulated by the CPU (this is the abstraction that we adopt, rather than as a tangle of gates and wires). In theory, it should be easier to transform an imperative program so that a CPU can execute it.

- **Declarative** programs say '*what*' rather than '*how*'.

  - **Functional**, e.g., Python, Ruby, C♯, Erlang, OCamL, Haskell, Lisp, Javascript, ...
    Functional programs are defined as collections of 'mathematical' functions and computation is undertaken by composing these functions together. Unlike imperative programming languages, there is no notion of 'state' in a functional program, with the **flow of**

control (which in an imperative program comes about by using, for example, if-statements and while-loops) being determined by function composition (especially by applying functions to themselves; but more of this concept later!). A typical functional program to compute $n!$ might be:

```
fact 0 = 1
fact n = n * fact(n - 1)
```

'interpreting code as data and data as code'

– **Logic**, e.g., Prolog, Oz, Gödel, Datalog, ...
Logic programs specify a 'logical solution' (using the reasoning incumbent within some **logic**); that is, a logic program generally consists of a set of **axioms** and a **goal statement**, along with some **rules of inference**. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms using the rules of inference. A typical logic program might have as rules of inference:

```
grandmother(X,Y) :- mother(X,Z), parent (Z,Y).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

and as axioms:

```
mother(mary,stan).
mother(gwen,alice).
mother(valery,gwen).
father(stan,alice).
```

with a goal statement of:

```
grandmother(stan,gwen).
```

In this case, simple reasoning by hand verifies that the goal statement can not be inferred (note that ':-' is to be read as 'implies from right to left'; commas are to be interpreted as 'and'; uppercase letters denote variables; and lower-case letters denote atoms).

'choosing an appropriate representation or modelling the relevant aspects of a problem to make it tractable'

- **Data-oriented**, e.g., SQL, dBase, Visual FoxPro, ...
  Data-oriented programs work with data through manipulating and searching **relations** (also known as **tables**). Such languages are prevalent in applications involving **databases**, which are, essentially, collections of tables of data. There is a whole (mathematical) theory of relations, and operations and concepts from this theory find their way into data-oriented programming languages. A typical operation in some data-oriented program might be to take a table consisting of students and their module marks together with a table consisting of students and their lecture attendance records and produce a table consisting only of the students who had failed the module and hadn't attended lectures (by the way, there is a strong correlation between the two!).

- **Scripting**, e.g., Python, Ruby, PHP, Javascript, Perl, ...
  Scripting programs are designed to automate frequently used tasks that involve calling or passing commands to external programs. They are designed for 'gluing' existing programs together: they assume the existence of a set of component programs and are intended primarily for computing using these component programs as individual entities. The origin of the term 'scripting' comes from the analogy where a movie script, i.e., the scripting program, tells the actors, i.e., the component programs, what to say, i.e., how to interact. Scripting programming languages originated with the operating system Unix where 'shell scripts' were sequences of instructions for the operating system to execute.

2.6 Programming languages

But there are even more programming paradigms!

- **Assembly**, e.g., MACRO-11, MIPS, C--, ...
  Assembly languages are regarded as low-level, rather than high-level, programming languages and they provide the interface between **machine code** (which consists of sequences of 0s and 1s that are directly

understandable by the CPU) and a high-level programming language (such as C or Fortran) in that a high-level program would first be **compiled** into an assembly language program and then into machine code (it is relatively easy to transform assembly language programs into machine code). Assembly languages provide instructions for explicitly manipulating data within the abstraction of computer hardware as a bunch of memory locations containing values: there are instructions for accessing the individual locations of main memory as well as the registers and the accumulator. However, every computer architecture has its own assembly language (dependent upon the processor's design) and consequently **portability** is a problem (that is, getting a program to run on different platforms). Programming in an assembly language by hand gives a high degree of control over the use of memory and the CPU but is painstakingly tedious; that's one reason why we express our algorithms in more user-friendly, high-level programming languages.

- **Concurrent**, e.g., Python, Ada, Cilk, Erlang, Java, Occam, ...
  Concurrent languages provide facilities for **concurrency** within programs. For example, a concurrent program might support a number of different **threads**, each of which undertakes its own computation and so that there are interactions between these threads, or different **processes** (that is, executing computations). Concurrent programs can still be executed on a single processor as the processor divides its time between the different threads or processes (that is, it **interleaves** executions and **time-shares**). There are two basic forms of interactions: via some **shared-memory**, where threads, for example, read and write to (the same) shared-memory locations in order to communicate (e.g., Erlang); or via **message passing** where processes, for example, communicate by sending messages to one another (e.g., Ada). The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different threads or processes, and coordinating access to resources that are shared amongst threads or processes.

- **Dataflow**, e.g., Vizsim, Oz, Openwire, ...
  Dataflow programs are specified by 'flows of data' (usually visual). So, a dataflow program might be represented as 'nodes' of data with 'arrows' from one node to another to represent the effect of some operation

on the data (within some node). Tracing the arrows from node to node represents an execution of the program. Dataflow programming languages were originally developed in order to make parallel programming easier (which they haven't!).

- **Fourth-generation**, e.g., SAS, Visual Dataflex, ...
  Fourth-generation languages originated between 1970 and 1990 and were built around databases and usually used in a commercial setting. They followed on from (guess what!) third-generation programming languages and were superseded by (yes, you've guessed it!) fifth-generation programming languages. Key to fourth-generation programming languages was the intention to reduce the effort, time and cost of developing software. They met with varying degrees of success!

- **List-based**, e.g., Lisp, Tcl, Scheme, Logo, ...
  List-based programming languages are built around the data-structure known as the **list**. Typical operations in a list-based programming language might be to strip the head element from a list or to concatenate two lists into one list. The archetypal list-based programming language is Lisp, which is the second oldest programming language still in use today (after Fortran), dating back to 1958. List-based programming languages are popular within **artificial intelligence** (**AI**) (which is the study and design of artificial agents).

- **Visual**, e.g., Lava, Prograph, Piet, ...
  Visual languages allow the user to specify programs using visual representations rather than by providing textual instructions; as such, dataflow programming languages are special cases. Of course, using visual programming languages requires totally different user interfaces so that the programmer can specify the programs. There are some weird and wacky visual programming languages, one of which is Piet whose programs are bitmaps that look like abstract art! Piet is an **esoteric** programming language; that is, a programming language that has been designed so as to test the boundaries of program design, as a proof of concept or as a joke!

There are many more programming paradigms (we haven't even mentioned **object-oriented**!). As can be inferred from above, some program languages

(one of which is Python) can operate according to a variety of different paradigms.

> ***'judging a program not just for correctness and efficiency but for aesthetics'***

2.7 Why more and more languages?

All of this brings us to a fundamental question: '*Why on earth do people keep on designing more and more programming languages, especially when anything that can be done in one programming language can be done in any other*?'

There are a number of 'drivers' that have forced the evolution of programming language design; moreover, these drivers, allied with advances in computing technology and new applications of computing, will continue to inspire programming language designers to come up with new programming languages in future (trust me!).

- **Productivity**
  Programming languages such as C, C++ and Java, are very detailed and lead to lots of lines of code, and so lots of expense in development and maintenance. Also, the rise of the Internet in the 1990's meant a brand new application area for programming. What was required were programming languages that sped up the software development process, reduced response times and costs, and supported fast user-interface development. So emerged **rapid applications development** (**RAD**) languages, like Python and Ruby, and subsequently the rise of scripting languages.

- **Reliability**
  It is incredibly difficult to come up with computer programs that are correct (we'll look at **program correctness** in more detail later). Some programming languages have features so that programs written in that language will be more reliable. Some of these features might involve: **type checking**, where we try to ensure that, for example, comparisons of two different **types** of values in a program execution, say a number and a letter of the alphabet, do not arise; and **exception**

handling, which is the ability to deal with errors that arise during a program's execution. Some programming languages allow for **aliasing** which is the ability to use different names to reference the same memory location. Aliasing can be a very dangerous feature as programmers often forget that the value of a variable has changed even though the variable itself has not been touched. To see this, think of a program in which there are two 'pointers', x and y, to the same memory location and where the name x does not appear (apart from when it is initially set up). If a program instruction changes the value of the memory location pointed to by y then the value pointed to by x has changed too, without x featuring!

'type checking: recognizing the virtues and dangers of aliasing'

- **Security**
  The Internet has raised the question of whether we can trust our programming languages. Consider the following scenario. You visit a web-page and in doing so you execute some Javascript, embedded in the web-page, but you execute this Javascript on your *own* machine. This might enable a malicious programmer to breach your security. In practice, Javascript programs are only allowed to undertake 'web-related' activity and not general purpose programming tasks, and their executions are closely tied to the web-page within which they are embedded (so, they don't have access to data resulting from visits to other web-pages).

- **Execution speed**
  As we have said, imperative programming languages are closely tied to our abstraction of computer hardware as a bunch of memory locations the values of which are explicitly manipulated by the CPU. Such imperative programming languages will not really be usable on a parallel computer (such as a multi-core or GPGPU machine) and consequently we will not achieve the increases in execution speed promised by parallel computing.

- **Curiosity**
  Computer Scientists will never stop having ideas about how things

might be done differently. As soon as someone describes some algorithm in a different way, out come the programming language designers to build a new programming language!

- **Style**
  As they say: '*Beauty is in the eye of the beholder*', and this applies equally to programming languages!

So you can see that there are many different meanings we could apply to the word 'satisfactory' that we used earlier when discussing the ability of a programming language to describe an algorithm.

There are some general principles that any half-decent programming language should adhere to. Any programming language should:

- be easy to use, with its programs easy to read, write and understand;
- support 'abstraction' so that adding new features and concepts should be possible;
- support testing, debugging and program verification (that is, proving that programs are correct; more of this later); and
- be inexpensive to use, in terms of execution time, memory usage and maintenance costs.

Some of these qualities are matters of opinion; that's what makes the design and appreciation of programming languages so contentious!

2.8 Logic programming: a glimpse

Let's build on our earlier very quick look at **logic programming**. Logic programming dates back to the late fifties and was developed by, amongst others, John McCarthy (McCarthy, one of the founding fathers of **artificial intelligence**, was presented with a Turing Award in 1971).

Logic programming essentially uses mathematical logic in order to 'compute'. Logic can be used:

- **procedurally**, e.g., '*if someone supports Newcastle United then they must be mad*'; or
- **declaratively**, e.g., '*Iain supports Newcastle United*'.

Incorporating logic procedurally and declaratively allows us to 'compute', e.g., from above we have 'computed' that '*Iain is mad*'.

The archetypal logic programming language is **Prolog**, with Prolog particularly prevalent in artificial intelligence environments (such as robotics, knowledge management, intelligent agents, natural language processing and e-learning). Prolog programs come in two parts:

- the **program** (or **database**; essentially, the facts that you know and the rules that you can apply to deduce new facts); and
- the **query** (the question you want answering, given what you know and what you can deduce).

2.9 A Prolog program

Here is a simple Prolog program. You'll quickly be able to grasp both the syntax and the semantics.

Some facts about Greek gods:

```
female(artemis).              female(persephone).
male(apollo).                 mother(gaia,uranus).
parents(uranus,gaia,rhea).    parents(cronus,rhea,zeus).
parents(cronus,rhea,hera).    parents(cronus,rhea,demeter).
parents(zeus,leto,artemis).   parents(zeus,leto,apollo).
parents(zeus,hera,hebe).      parents(demeter,zeus,persephone).
```

Notes: a full-stop terminates a fact; 'atoms' (such as $artemis$ and $rhea$) start with a lower-case letter; and the name of a fact, such as $female$, is the initial prefix of the fact. Examples of how we might interpret the facts above are as follows:

$male(appollo)$.
'*Apollo is male.*'
$mother(gaia, uranus)$.
'*Gaia is the mother of Uranus.*'
$parents(zeus, leto, artemis)$.
'*Zeus and Leto are the parents of Artemis.*'

Some rules about Greek Gods:

```
mother(X,Y) :- parents(_,X,Y).
female(X) :- mother(X,_).
father(X,Y) :- parents(X,_,Y).
sibling(X,Y) :- parents(A,B,X), parents(A,B,Y), X \= Y.
grandma(X,Y) :- mother(X,Z), parents(_,Z,Y).
grandma(X,Y) :- mother(X,Z), father(Z,Y).
descend(X,Y) :- mother(X,Y).
descend(X,Y) :- father(X,Y).
descend(X,Y) :- descend(X,Z), descend(Z,Y).
```

Notes: ':-' is to be read as 'implies from right to left'; '_' denotes a 'wild card'; variables (such as X and Y) start with a capital letter; commas are to be read as 'and'; and \= is to be read as 'not equals'. Examples of how we might interpret the rules above are as follows:

$female(X)$:- $mother(X, \_)$.
'*If X is the mother of someone then X is female.*' or '*Any mother is female.*'

$father(X, Y)$:- $parents(X, \_, Y)$.
'*If X is the first parent of Y then X is the father of Y.*'

$sibling(X, Y)$:- $parents(A, B, X), parents(A, B, Y), X \backslash= Y$.
'*If X and Y are different persons and have the same parents then they are siblings.*'

$grandma(X, Y)$:- $mother(X, Z), father(Z, Y)$.
'*If X is the mother of Z and Z is the father of Y then X is the grandma of Y.*'

Note the 'recursive' rule $descend(X, Y)$:- $descend(X, Z), descend(Z, Y)$. We might interpret this rule as: '*If X is a descendant of Z and Z is a descendant of Y then X is a descendant of Y.*'

'thinking recursively'

Queries to our program and the responses might be as follows:

`?- mother(gaius,zeus).`	Q: Is Gaius the mother of Zeus?
`no`	A: no
`?- descend(X,hebe).`	Q: Give me the descendants of Hebe

`zeus`	A: Zeus
`;`	Q: Give me another
`hera`	A: Hera
...	

If you fancy a play online then you can 'logic program' at:

`http://kti.mff.cuni.cz/~bartak/prolog`

2.10 Research glimpse: Programming in Parallel

Why should we program in parallel? According to Simon Peyton-Jones and Satnam Singh (of Microsoft Research):

- **Performance**. We need to write parallel programs to achieve improving performance from each new generation of multi-core processors.
- **Hiding latency**. Even on single-core processors we can exploit concurrent programs to hide the latency of slow I/O operations to disks and network devices.
- **Software structuring**. Certain kinds of problems can be conveniently represented as multiple communicating threads which help to structure code in a more modular manner, e.g., by modelling user interface components as separate threads.
- **Real world concurrency**. In distributed and real-time systems we have to model and react to events in the real world, e.g., handling multiple server requests in parallel.

What are the obstacles? Programming in parallel is hard and going to get harder! As Anwar Ghuloum (Principal Engineer, Intel) says: '*Ultimately, developers should start thinking about tens, hundreds and thousands of cores now in their algorithmic development and deployment pipeline.*' Programmers will need to 'think parallel'.

The **Stanford Pervasive Parallelism Lab** (**PPL**) has the following goal: '*The goal of the PPL is to make heterogeneous parallelism accessible to average software developers through domain-specific languages* (*DSLs*)*, allowing the domain expert to develop parallel software without becoming an expert in parallel programming. Our approach is to use ... an underlying*

architecture that provides efficient mechanisms for communication, synchronization and performance monitoring.'

'processing in parallel'

2.11 Research glimpse: Ubiquitous/Pervasive Computing

Both hardware and software are increasingly being embedded in a variety of environments. Currently very active research areas of Computer Science are **ubiquitous computing** and **pervasive computing**, also known as **ubicomp** and the **Internet of Things**. Ubiquitous computing involves the integration of computers and software into everyday objects and activities so that we can control (or have controlled for us) remote aspects of our lives. For example, a sensor in our fridge might detect that we were low on milk and automatically order more.

The enabling (engineering) technology is **RFID** (**radio frequency identification**) which uses radio waves to transfer data from an electronic tag to a reader. Bulk reading can be undertaken (over a distance of metres), and tags can be active (where they have their own energy source) or passive (where they operate using energy transmitted by the reader).

Green computing is a currently vogue area of Computer Science involving anything to do with energy conservation within the world of information technology (and is clearly very relevant to ubiquitous computing). This can range from designing chips which use less power to writing programs where the main unit of resource is energy expended (rather than the execution time as is usually the case). There are numerous areas of research relating to programming within an energy-conscious environment, one of which is the design of programming languages that will both make programmers aware of their programs' energy usage and also give them some control over this usage. In designing programming languages for the Internet of Things, the focus tends to be on energy awareness, security, resilience and communications.

2.12 Alan Perlis quotations

Alan Perlis was a pioneer of programming languages. He was the first person to receive a Turing Award (in 1966). The citation read: '*for his influence*

in the area of advanced programming techniques and compiler construction' (we'll take a look at compilers in a moment). He is responsible for a great many quotations as regards programming and programming languages (see [5]). Here are one or two!

- '*It is easier to write an incorrect program than understand a correct one.*'
- '*A language that doesn't affect the way you think about programming, is not worth knowing.*'
- '*To understand a program you must become both the machine and the program.*'
- '*There will always be things we wish to say in our programs that in all known languages can only be said poorly.*'
- '*Think of all the psychic energy expended in seeking a fundamental distinction between "algorithm" and "program".*'
- '*Most people find the concept of programming obvious, but the doing impossible.*'

2.13 Syntax and semantics

Every programming language has:

- **syntax**: the rules that govern what makes a program 'legitimately written'; and
- **semantics**: the rules that govern what a program 'means'.

The syntax of a programming language should be precisely defined so that as to what constitutes a legal program is unequivocal (after all, it's the machine that tells us when our programs don't make sense and so there must be a program, and hence an algorithm, that decides this). Also, the semantics of a programming language should be precisely defined so that we are left in no doubt as to what a program will do when it is executed. As regards both syntax and semantics, the word 'precise' should be interpreted as 'mathematically formal'. As it happens, whilst every programming language

has a formal syntax, not every programming language has a formal semantics: the semantics of C and Fortran, for example, have only been described using informal English and not precise mathematics, unlike ML where there is a complete formal semantics.

There are a number of problems with only having an informal semantic understanding of a programming language:

- '*How can we be sure that what the programmer thinks the program does is what it actually does*?'
- '*How can we prove* (*or give a rigorous argument*) *that the program does what it's meant to do*?
- '*How can I be sure that my program executes identically when I run it on different machines*?'

In Computer Science, informality leads to ambiguity which leads to divergence which leads to errors!

Computer Science turns to different concepts within **Mathematics**, such as **set theory**, **logic** and **category theory**, to define different types of formal semantics:

- **denotational semantics**: a program's meaning is given mathematically as a suitable mathematical structure (such as a function or a partial function), and these mathematical structures should be composable so that the **denotations** of parts of programs can be assembled to obtain a denotation of the program itself;
- **operational semantics**: a program's meaning is given in terms of the individual steps of a computation the program makes when it executes; for example, each individual step might be described according to how the program's state changes; and
- **axiomatic semantics**: a program's meaning is given indirectly in terms of a collection of logical properties it satisfies, and how these properties are maintained during or after execution.

As computers become ever more pervasive and are used in more and more safety-critical applications (like aircraft, nuclear power stations, cars and so on), more and more demands are being placed upon software. Increasingly

there are demands placed upon the programming languages that are used to write safety-critical programs with these demands relating to their formal semantics (or lack thereof!).

2.14 Compilation vs. interpretation

High-level programming languages, such as Python, Java, C and C++, are designed to be used by humans but ultimately have to be executed by a machine. Let's look at how we move from high-level programs to machine code. There are two essential mechanisms for doing this.

- A **compiler** transforms the high-level program to machine code for execution by the CPU (recall that different types of CPUs have different machine code so a compiler is specific to a particular type of CPU). The programming languages C, C++, Fortran and Haskell, for example, all have their programs compiled directly to machine code (passing through some assembly language on the way).
- An **interpreter translates** one instruction (or possibly a small number of instructions) of the high-level program at a time, as it is needed, with the translations interspersed with the activity of the program itself. The programming languages Python, Ruby and Javascript, for example, all have their programs translated.

There are trade-offs as to whether compilation or interpretation is preferable.

- Compiled programs tend to run faster as the execution of the resulting compiled program (machine code) isn't interspersed with compilation activity.
- Compilation spends time optimising so as to enhance performance. This isn't possible with interpretation where the high-level program is handled piecemeal.
- Compilation spends time analysing so as to search for bugs and eliminate certain run-time errors. Again, this isn't possible with interpretation where the high-level program is handled piecemeal.

- Interpreted programs tend to use memory better. To see why this might be the case, imagine compiling a large program. You would need enough (fast) memory to store the whole compiled program. However, if the instructions of the high-level program are handled one by one then you only need enough memory to store (the translations of) a small number of instructions.

- Interpreted programs facilitate development in that as a program is developed, it is executed and bugs are unearthed with the code then being rewritten (in sometimes what can seem a never-ending cycle!). If a program has to be compiled each time it is to be executed then the compilation time can mount up. Interpretation is (pretty much) free from this compilation time.

Of course, as is normally the case in Computer Science, things are never clear-cut for there is a spectrum of possibilities between 'pure compilation' and 'pure interpretation'.

For example, some programming languages, such as Java, have their programs compiled into what is called **bytecode**, which is intermediate code that is fast to interpret, and then translated by a **Java Virtual Machine** for the target processor. Thus, all a processor needs is a Java Virtual Machine for that specific processor rather than its own special compiler.

Dynamic (just-in-time) compilation is when some of the operations within the compilation of the source program (like optimisation) are postponed until the initial stages of execution have been completed. The reason for doing this is that nowadays programs increasingly use **run-time libraries** where (other people's library) code is slotted in. Without knowing exactly what this code is, the compiler cannot fully optimise. Of course, because the library is run-time, this information doesn't become available until the (semi-) compiled program is executed. A negative aspect of dynamic compilation is that it increases initial execution time.

2.15 Compilation

As we have heard, a **compiler** translates high-level programs into machine code (we'll return to where compilers reside within our computer system abstraction later). There are 4 essential phases to compilation.

- A **lexical analyser** reads a high-level program as a string of symbols and converts it into basic syntactic components, i.e., **tokens** in a **token stream** (it recognises, for example, reserved words from the programming language and other basic symbols). As an illustration, the piece of code:

```
{   let x = 1;
    x := x + y;
}
```

  might be converted into the token stream:

  LBRACE LET ID/x EQ NUM/1 SEMIC
  ID/x ASS ID/x PLUS ID/y SEMIC RBRACE

- A **syntax analyser** (or **parser**) recognises the syntactic structure of the token stream and this results in a **parse tree** (or a **syntax tree**) as shown in Fig. 2. This parse tree often contains redundant information and is simplified; it is also used by the compiler to undertake various analysis operations (like type-checking).

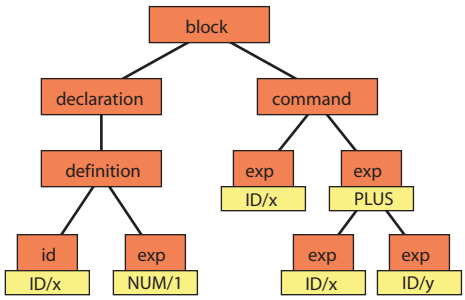

Fig. 2: a parse tree constructed by a syntax analyser.

  You can see how the tree's structure reflects the structure of the original source program.

- A **translation phase** flattens the (modified, simplified and optimised) parse tree into a linear sequence of **intermediate code**. For example, the Java code:

```
if (x >= 3){
    y = -x;
} else {
    y = x;{
```

ultimately translates as intermediate code (actually, Java bytecode):

`iload_4`	load x (4th load variable, say)
`iconst_3`	load 3
`if_icmpgt L36`	if greater (i.e., condition false) then jump to L36
`iload_4`	load x
`ineg`	negate it
`goto L37`	jump to L37
`label L36`	
`iload_4`	load x
`label L37`	
`istore_7`	store y (7th local variable, say)

(Recall that Java is not compiled all the way to machine code but to bytecode.)

- A **code generation** phase converts the intermediate code into **assembly** (and then on to **machine code**). For example, C code corresponding to the Java code above becomes (on an **ARM processor**):

`LDR r0,[fp,#-4-16]`	load x (4th local variable)
`MOV r1,#3`	load 3
`CMP r0,r1`	
`BGT L36`	if greater then jump to L36
`LDR r0,[fp,#-4-16]`	load x
`RSB r0,r0,#0`	negate it
`STMDB sp!,r0`	i.e., PUSH $r0$ (to local stack)
`B L37`	jump to L37
`L36: LDR r0,[fp,#-4-16]`	load x
`STMDB sp!,r0`	i.e. PUSH $r0$
`L37: LDMIA sp!,r0`	i.e., POP $r0$
`STR r0,[fp,#-4-28]`	store y (7th local variable)

Recall that every type of processor has its own assembly language and machine code.

2.16 Lexical analysis

Let's look at two of the phases of compilation in more detail as there are some fundamental notions of Computer Science that underlie their methodology. The lexical analysis phase can account for more than 50% of compile time, for character handling can be expensive and there is a large number of characters in a program in comparison to tokens. Tokens (such as keywords, identifiers, numeric constants and string constants) are almost always defined using regular expressions (this yields an effective way of recognizing tokens).

A **regular expression** over some **alphabet** Σ (that is, some finite set of symbols) is defined as follows:

- any $a \in \Sigma$ is a regular expression;
- $\emptyset$ and ϵ are special regular expressions; and
- if ω and ω' are regular expressions then so are:
  - $(\omega\omega')$
  - $(\omega \mid \omega')$ ($\mid$ is also written $+$ or $\cup$ or $\vee$)
  - (ω^*).

So, we can build up a vast collection of regular expressions by repeatedly applying the above constructions. (The parentheses (and) are used to avoid ambiguity and, in order to make things more readable, we sometimes omit them when our construction is clear.)

Every regular expression denotes a **set of strings** (or **language**) over Σ:

- a, $\emptyset$ and ϵ denote the sets of strings $\{a\}$, $\{\}$ (the empty set of strings) and $\{\epsilon\}$ (the set containing just the empty string ϵ), respectively; and
- if ω and ω' denote the sets of strings R and S then:
  - $(\omega\omega')$ denotes $\{xy : x \in R, y \in S\}$
    (the string xy is the string x **concatenated** with the string y)
  - $(\omega \mid \omega')$ denotes $R \cup S$

– (ω^*) denotes $\{x_1x_2\ldots x_n : n \geq 0, x_1, x_2, \ldots, x_n \in R\}$
($*$ is called **Kleene iteration**; note that the set denoted by (ω^*) always includes the empty string ϵ: put $n = 0$ into the definition).

Our methods of defining regular expressions and defining the sets of strings denoted by regular expressions are both **recursive**; that is, we define something 'in terms of itself'. Recursion is an extremely powerful concept and is ubiquitous in Computer Science. We'll see it in its full glory when we look at specific (recursive) algorithms later on.

'thinking recursively'

2.17 Some regular expressions

Let's look at some sets of strings and see if we can work out a regular expression that represents them (it turns out that not every set of strings can be represented by a regular expression; the ones below can, though).

- S consists of all strings over $\{a, b\}$ beginning with a and ending in b.

We note that any string in S consists of the concatenation of the string a (represented by the regular expression a) with *any* string and then with the string b (represented by the regular expression b). We have at our disposal an operation we can use that allows us to represent concatenation. So, if we can come up with a regular expression to describe the set of all strings then we'll almost have what we want. The set of *all* strings can be represented by the regular expression $(a|b)^*$; that is, 'concatenate the symbols a and b as and when you like (and possibly not at all, so as to obtain the empty string)'. So, our set S is represented by the regular expression $a(a|b)^*b$.

- S consists of all strings over $\{a, b\}$ containing the string $abab$ (as a contiguous sub-string).

We can think of any string in S as 'any old string concatenated with the string $abab$ and then concatenated with any old string'. We have just seen that the set of all strings over $\{a, b\}$ is represented by the regular expression $(a|b)^*$ and so the set S is represented by the regular expression $(a|b)^*abab(a|b)^*$ (as $abab$ is a regular expression denoting the string $abab$).

- S consists of all strings over $\{a, b\}$ in which no two a's appear consecutively.

How might we think of our set S? We can build any string in S by repeatedly concatenating (onto the end of our current string) an a or a b but so that if we have just placed an a onto the end of our previous string then we must next concatenate a b onto our string or stop the process. This amounts to repeated concatenations of the strings ab and b, which is represented by the regular expression $(ab|b)^*$. However, we must allow for the possible final concatenation being an a and so we form the regular expression $(ab|b)^*(a|\epsilon)$ (the ϵ, which translates as 'add on nothing', allows us to finish our string with a b if we wish).

However, there are other regular expressions representing S. We can think of building the strings in S in a different way. We might describe the process of construction as 'start off with a string of b's; then repeatedly concatenate an a followed by at least one b, before possibly closing off the process by concatenating an a'. The regular expression describing this process is $b^*(abb^*)^*(a|\epsilon)$. (By the way, it is an extremely difficult problem to check whether two regular expressions describe the same set of strings; we'll find out much later on precisely what we mean by 'difficult'!)

- S consists of all strings over $\{a, b, c\}$ where there is an even number of c's.

In building strings from the set S, we are simply not concerned with how we use the a's and b's, so long as whenever we concatenate our current string with an 'odd-numbered' c, at some point in the future we concatenate our string with another 'even-numbered' c. A regular expression describing this process is $(a|b)^*(c(a|b)^*c(a|b)^*)^*(a|b)^*$.

From our descriptions above, it would appear that there might be a link between the sets of strings represented by regular expressions and the sets of strings constructed according to some process (after all, when we thought above about how to build our regular expressions, we moved into 'construction mode' to describe the sets). It turns out that there is! This link helps us to automate the processing of regular expressions, and so of tokens within compilation.

2.18 Finite state machines

We'll now describe the computational processes that allow us to alternatively define the sets of strings that are representable by regular expressions.

A **finite state machine** M (**FSM**; also known as a **finite state automaton**, **FSA**) is defined as:

$$M = (\Sigma, Q, \delta : Q \times \Sigma \to Q, q_0 \in Q, F)$$

where:

- Σ is some finite alphabet;
- Q is some finite set of **states** with **initial state** q_0 and set of **final states** $F \subseteq Q$; and
- $\delta : Q \times \Sigma \to Q$ is the **transition function**.

On **input** any string $a_1 a_2 \dots a_n$ over Σ (where $n \geq 0$, and so we may input the empty string if we wish), M yields a sequence of states $q_0, q_1, q_2, \dots, q_n$ via:

$$q_1 = \delta(q_0, a_1), q_2 = \delta(q_1, a_2), q_3 = \delta(q_2, a_3), \dots, q_n = \delta(q_{n-1}, a_n)$$

(obviously, if our input string is empty then our sequence is simply q_0). Our input string is **accepted** by our FSM M if the resulting sequence of states $q_0, q_1, q_2, \dots, q_n$ ends in a final state; that is, is such that $q_n \in F$. Thus, M **accepts** a set of strings over Σ; that is, a **language** (just as a regular expression represents such a set of strings).

FSM's are most easily represented pictorially where: each state is named in a circle; circles with a double border denote final states; the initial state is pointed at by an arrow; and the transition $\delta(q, a) = q'$ is depicted by drawing an arrow from the circle representing state q to the circle representing state q' and labelling this arrow with a. For example, an FSM M that accepts the set of strings not containing 2 consecutive a's is depicted in Fig. 3 (final states are shaded for clarity).

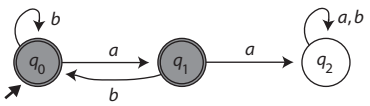

Fig. 3: a representation of an FSM.

Here is the important theorem!

Theorem 1 *A set of strings is represented by a regular expression if, and only if, it is accepted by a finite state machine* (*such sets of strings are called the* **regular languages**).

2.19 Some equivalences

So, let's return to the regular expressions we derived previously. By Theorem 1, we should be able to devise FSMs accepting the same sets of strings. The FSMs depicted in Fig. 4 do exactly that!

As to whether we use a regular expression to represent a set of strings or an FSM to accept the set of strings depends upon the actual set of strings and the context within which we are working (and is also a matter of taste). As can be seen from the examples above, sometimes it is easier to describe a set of strings by using a regular expression and sometimes by using an FSM. Regular expressions and finite state machines crop up *all over* Computer Science. In fact, they have so many different applications and are used in so many different ways that it is difficult to know where to start; so much so that I'm not going to say anything more about this! You'll meet them often enough in your studies.

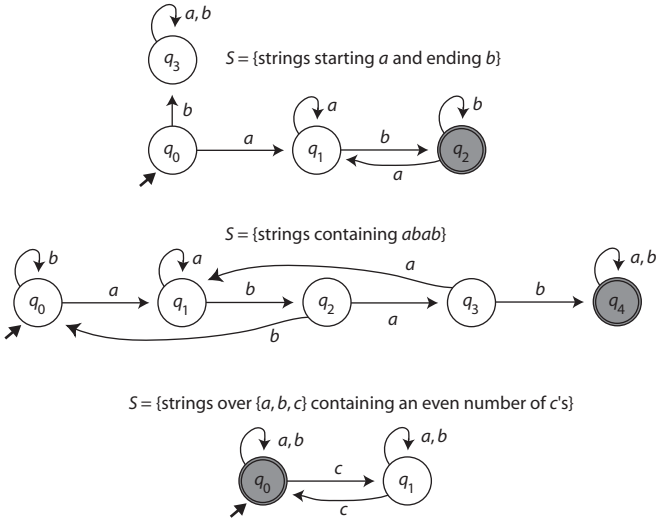

Fig. 4: some more FSMs.

2.20 Syntax analysis

Now let's turn to the syntax analysis phase of compilation and look at some of the underlying fundamental concepts. Just as there are some ingenious mathematical notions and methods hidden away in the lexical analysis, so it is for syntax analysis. In looking more closely at syntax analysis, we'll introduce *another* method by which we can describe sets of strings. Just to remind you: in syntax analysis, our sets of strings consist of the token streams derived from the programs of some high-level programming language. Our primary goal is to take such a token stream and decide whether it corresponds to a legitimate program or not, so that if it does then we'll transform it into machine code (possibly with some useful optimisations).

The syntax of a programming language is usually described by a context-free grammar. A (**phrase-structure**) **grammar** is a tuple (N, T, s, R), where:

- T and N are disjoint finite sets of **terminal** and **non-terminal** symbols, respectively;
- $s \in N$ is the **start symbol**; and
- R is a finite set of **productions** or **rules** so that

$$R \subseteq (N \cup T)^+ \times (N \cup T)^*$$

  and where if $(\alpha, \beta) \in R$ then α contains at least one symbol from N.

(Note that $(N \cup T)^+$ denotes all strings over $N \cup T$ except the empty string.)

In a **context-free grammar**, productions are of the form:

- $b \to a_1 a_2 \ldots a_k$, with $k \geq 0$, $b \in N$, and $a_1, a_2, \ldots, a_k \in N \cup T$

(note that as $k \geq 0$, a production might be of the form $b \to \epsilon$, where ϵ is the empty string).

A production $b \to a_1 a_2 \ldots a_k$ can be **applied** to a string ω containing b via:

$$\omega = \ldots b' \underline{b} b' \ldots \Rightarrow \omega' = \ldots b' \underline{a_1 a_2 \ldots a_k} b' \ldots$$

so as to transform the string ω into the string ω': the left-hand side of some production appearing in ω (that is, just a solitary symbol of N, such as b

above) is replaced in ω by the right-hand side of the production (such as $a_1 a_2 \ldots a_k$, above) so as to get the string ω' (note that the left-hand side of every production is always a symbol in N and that we use the symbol $\Rightarrow$ to denote the application of some production to some string).

A grammar **generates** the set of strings over T that can be obtained by repeatedly applying productions starting from the string s. A sequence of applications of productions to derive some string is called a **derivation**.

Note that we may have a choice of production to apply; that is, the process of string derivation is **non-deterministic**. So, there may be more than one string generated starting from s and also a number of derivations witnessing that some string is generated by a context-free grammar. Again, just like recursion, non-determinism is a fundamental concept within Computer Science and we'll meet it again later on.

'using and understanding the power of non-determinism'

In the context of compilation, if the legitimate programs of our high-level programming language can be defined using a context-free grammar (that is, we can define the programming language's syntax using a context-free grammar) then we have a method for deciding whether some program is indeed well-written: roughly speaking, we simply check to see whether it can be generated by the context-free grammar.

So, now we have three different methods of describing sets of strings: using regular *expressions*; using finite state *machines*; and using context-free *grammars*. Whereas there is a direct relationship between regular expressions and FSMs (see Theorem 1), certain grammars are partners in this relationship too but not context-free ones (we'll return to this point soon).

2.21 A context-free grammar

Let's look at a real context-free grammar and see if we can work out the set of strings it generates. Consider the context-free grammar $(N = \{s, t, u\}, T = \{a, b\}, s, R)$, with R consisting of the productions:

$$s \to \epsilon \qquad s \to bs \qquad s \to at \qquad t \to bs \qquad t \to \epsilon \qquad t \to au.$$

Starting with (the start symbol) s, there are 3 choices as to which production to apply:

- $s \to \epsilon$ and we are finished having generated ϵ
- $s \to bs$
- $s \to at$

Perhaps we repeatedly apply $s \to bs$ to get the derivation:

$$s \Rightarrow bs \Rightarrow b^2s \Rightarrow b^3s \Rightarrow \ldots \Rightarrow b^ns$$

and then $s \to at$ to get b^nat, or $s \to \epsilon$ to finish and generate b^n. So: either we've finished and have generated b^n, for some $n \geq 1$; or we haven't finished and we've derived b^nat, for some $n \geq 1$. Alternatively, if our first application is not the rule $s \to bs$ but the rule $s \to at$ then we have derived the string at. Consequently, no matter how we begin, if we haven't finished then we must have derived a string of the form b^nat, for some $n \geq 0$.

Perhaps we then apply $t \to bs$ to get $b^nabs \ldots$ or maybe we apply $t \to \epsilon$ to generate b^na (and finish) $\ldots$ or maybe we apply $t \to au$ to derive b^naau from whence we are stuck! So, if we haven't finished and aren't stuck then we've derived b^nabs, for some $n \geq 0$. Now go again!

Repeatedly arguing in this way yields that we generate the set of strings

$$\{\omega : \omega \text{ does not contain 2 consecutive } a\text{'s}\}.$$

We saw earlier that this set of strings can be defined using regular expressions or FSMs. Whereas working out the set of strings represented by some regular expression or accepted by some an FSM can be difficult, from above it would appear to be the case that working out the set of strings generated by some context-free grammar is just as difficult if not more so! In fact, it turns out that *any* set of strings definable using regular expressions or FSMs can also be generated by some context-free grammar but that there are sets of strings generated by context-free grammars that cannot be defined using *any* regular expression or FSM; that is, context-free grammars are 'more powerful' than regular expressions and FSMs. This could well account for the apparent increase in difficulty in working out the set of strings generated by such a grammar.

2.22 Regular grammars

Whilst context-free grammars are more powerful than regular expressions and FSMs, it turns out that there is a type of grammar that has exactly

the same power as regular expressions and FSMs, namely regular grammars, which are very special types of context-free grammars.

A **regular grammar** (N, T, s, R) is a context-free grammar where all productions are of one of the following forms:

- $b \to a$ where $a \in T$
- $b \to ac$ where $a \in T$ and $c \in N$
- $b \to \epsilon$

In fact, the context-free grammar we have just looked at is actually a regular grammar too.

Theorem 2 *A set of strings is represented by a regular expression if, and only if, it is accepted by a finite state machine if, and only if, it is generated by a regular grammar.*

So, for each of the sets of strings we looked at earlier (the ones that we showed are represented by some regular expression and accepted by some FSM), you should be able to build a regular grammar that generates exactly this set of strings! Here are some grammars that do just this.

- $S = \{$strings starting a and ending $b\}$
  grammar $(N = \{s, t\}, T = \{a, b\}, s, R)$, where R is:

  $s \to at \quad t \to at \quad t \to bt \quad t \to b.$

- $S = \{$strings containing $abab\}$
  grammar $(N = \{s, t_1, t_2, t_3, u\}, T = \{a, b\}, s, R)$, where R is:

  $s \to as \quad s \to bs \quad s \to at_1 \quad t_1 \to bt_2 \quad t_2 \to at_3 \quad t_3 \to bu$
  $u \to au \quad u \to bu \quad u \to \epsilon.$

- $S = \{$strings where no two a's appear consecutively$\}$
  grammar $(N = \{s, t\}, T = \{a, b\}, s, R)$, where R is:

  $s \to \epsilon \quad s \to bs \quad s \to at \quad t \to \epsilon \quad t \to bs.$

- $S = \{$strings over $\{a, b, c\}$ containing an even number of c's$\}$
  grammar $(N = \{s, t\}, T = \{a, b, c\}, s, R)$, where R is:

$s \to as \quad s \to bs \quad s \to ct \quad t \to at \quad t \to bt \quad t \to cs$
$s \to \epsilon.$

There are special 'machines' called **pushdown automata**, which are essentially finite state machines equipped with extra specialized memory called a **stack**, that characterize the languages generated by context-free grammars. These pushdown automata help us to automate the processing of context-free grammars; in particular, if the syntax of a programming language can be described using a context-free grammar then it turns out that we can very quickly build parse-trees and so efficiently spot programs that have been written incorrectly.

2.23 Backus Naur Form

In the world of programming languages, context-free grammars are usually expressed in **Backus Naur Form** (**BNF**):

- productions with the same left-hand sides can be separated with | (so as to denote choice)
- non-terminals appear inside $\langle\ \rangle$
- ::= is used instead of $\to$ (typewriters don't have a $\to$ key!)

Let's look at a context-free grammar presented using BNF. Consider defining a floating point number, e.g., 3.45×10^{-4}. Here is a grammar that allows us to do this.

start symbol $\langle F \rangle$
$\langle s \rangle$ ::= + | −
$\langle d \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\langle J \rangle ::= \langle d \rangle \langle J \rangle \mid \langle d \rangle$	integer
$\langle I \rangle ::= \langle s \rangle \langle J \rangle \mid \langle J \rangle$	signed integer
$\langle H \rangle ::= \langle J \rangle \mid p\langle J \rangle \mid \langle J \rangle p \langle J \rangle$	digits possibly involving a decimal
$\langle G \rangle ::= \langle H \rangle \mid e\langle I \rangle \mid \langle H \rangle e \langle I \rangle$	decimal digits maybe with an integral exponent
$\langle F \rangle ::= \langle G \rangle \mid \langle s \rangle \langle G \rangle$	signed decimal digits maybe with an integral exponent

So, a derivation to generate 3.45×10^{-4} is:

$$\langle F \rangle \Rightarrow \langle G \rangle \Rightarrow \langle H \rangle e \langle I \rangle \Rightarrow \langle J \rangle p \langle J \rangle e \langle I \rangle \Rightarrow \langle d \rangle p \langle J \rangle e \langle I \rangle$$
$$\Rightarrow 3p \langle J \rangle e \langle I \rangle \Rightarrow 3p \langle d \rangle \langle J \rangle e \langle I \rangle \Rightarrow 3p \langle d \rangle \langle d \rangle e \langle I \rangle \Rightarrow 3p4 \langle d \rangle e \langle I \rangle$$
$$\Rightarrow 3p45e \langle I \rangle \Rightarrow 3p45e \langle s \rangle \langle J \rangle \Rightarrow 3p45e - \langle J \rangle \Rightarrow 3p45e - \langle d \rangle$$
$$\Rightarrow 3p45e - 4$$

References

The books below all include introductory material as regards basic aspects of programming languages and compilers.

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers*: *Principles, Techniques and Tools* (2nd edition), Pearson (2007) [005.453 AHO].

[2] N. Dale and J. Lewis, *Computer Science Illuminated* (4th edition), Jones and Bartlett (2009).

[3] D. Grune, H. Bal, C. Jacobs and K. Langendoen, *Modern Compiler Design* (2nd edition), Wiley (2009) [on order].

[4] D.E. Knuth, *The Art of Computer Programming Volume* 1: *Fundamental Algorithms*, Addison-Wesley (1968).

[5] A.J. Perlis, Epigrams on Programming, *ACM SIGPLAN Notices*, vol. 17, no. 9 (1982) 7–13 [e-copy available from library].

[6] M.L. Scott, *Programming Language Pragmatics* (2nd edition), Morgan Kaufmann (2006) [005.13 SCO].

[7] R.W. Sebesta, *Concepts of Programming Languages* (9th edition), Addison Wesley (2009).

[8] A.B. Webber, *Modern Programming Languages*: *A Practical Introduction* (2nd edition), Franklin, Beedle and Associates (2002).

[9] *List of programming languages*, Wikipedia: the free encyclopedia (2012), URL: `http://en.wikipedia.org/wiki/List_of_programming_languages` [page last accessed: 10th September 2017].