What is Computer Software?

Algorithm - Sequences of precise instructions that can be applied to data items
Programming Language - A means for translating an algorithm into a form suitable for execution by a computer. Into a program that can then be translated into a form suitable for execution by a computer.

An algorithm is not a computer program

- Program is written in a programming language
- There are an infinite number of **implementations** of algorithms as programs

1 Programming languages

Programming languages can be of different **programming paradigms**:

- Imperative - Statements change a programs **state** (closest to "memory abstraction" of CPU) (Python, C++, Java etc) - Take data, do operation on it, then write that data back to memory
- Declarative - Programs say **what** rather than **how** - Lack flow control: loops, if statements etc
  - Functional - Programs are defined as "mathematical" functions (don't have standard set of syntax: if etc). To get an answer build up a large functional expression that gives the correct output given an input
  - Logic - Programs specify a "logical solution" - input rules and known information, then ask a question
- Data-oriented - Programs work with data through manipulating and searching relations (tables). Tables have things in common that can be linked together to get more information.
- Scripting - Designed to automate frequently used tasks that involve calling or passing commands to external programs. These languages have lots of libraries to make things easier to do
- Object Oriented - Small piece of code relating to a particular object (person etc) create instances of object with different information, link these together

More rare types:

- Assembly - Low level languages providing the interface between machine code and a high-level programming language. Faster as closer to what the machine processes so less overhead
- Concurrent - Provides facilities for concurrency (threads, processes); may be **message-passing** or **shared-memory**.
- Dataflow - Programs specified by flows of data (usually visual, like a flowchart)
- Fourth-generation - High level languages built around databases
- List based - Built around list data structure
- Visual - Allows user to specify programs using visual representations

1.1 Why more and more languages

Programming language design has been driven by

- Productivity
  - C++, Java etc
  - Expensive to develop/maintain
  - Rapid application development (RAD) languages - quicker to make, quicker to sell, quicker to make money
- Reliability
  - Features to enhance error detection (type checking etc)

  - Aliasing - 2 variables referring to the same piece of data, at some time change the value of one of the variables. This changes the other value also

- Security
  - Do we trust programming languages? - Using these languages to send bank detauls etc
- Execution speed
  - Imperative languages are not tied to sequential computers
  - Not to parallel/distributed computers
  - Need speed for medical, scientific etc
- Curiosity and style

1.2 Programming languages should

- Be easy to use,read and write (collaborate)
- Support abstraction (adding new features)
- Support testing, debugging and verification
- Be inexpensive to use and maintain

1.3 Logic Programming

- Dates back to the late fifties
- Uses mathematical logic to ”compute”
- Learn more and more until you know the answer to a question

Logic is used

- Procedurally - Rule
- Declaratively - Input

Incorporating logic procedurally and declaratively allows us to ”compute”

Programs come in 2 parts

- The program - Things you know and a list of rules
- The query - What do you want to learn?

1.3.1 Example prolog program

- Start with a list of facts (full stop terminates a fact)
- Add some rules (Underscore is a wildcard) (use commas as and to allow for more complicated rules)

```
female(artemis).
female(persephone).
male(apollo).
mother(gia,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).
%rules
mother(X,Y):-parents(_,X,Y).
female(X):-mother(X,_). %If a person is the mother of anybody, they are female
father(X,Y):-parents(X,_,Y)
%These rules can be continued to make other rules about families
%Query
?- mother(gia,zeus).
no
```

2 Research Glimpse - Parallel Computing

Programming in parallel
Why should we program in parallel

- Performance - speedup when using all cores
- Hiding latency (e.g. slow IO) (keep doing instructions until information received)
- Real world concurrency (lots of people able to use a service)

What are the obstacles

- Programming in parallel is very hard, dealing with lots of threads and things going on at the same time. Multiple threads accessing the same piece of data at the same time could cause errors

3 Research Glimpse - Ubiquitous/Pervasive computing

Currently very active research areas are **ubiquitous** and **pervasive** computing (a.k.a ubicomp and the Internet of Things):

- The integration of computers and software into everyday objects and activities

RFID(radio frequency identification) uses radio waves to transfer data from an electronic tag to a reader:

- bulk reading can be undertaken (over a distance of metres)
- tags can be active or passive

Programming languages for the Internet of things

- focus on energy awareness, security, resilience and communications

4 Syntax and Semantics

Every programming language has

- Syntax - How a program is written
- Semantics - what a program means. What does it try to achieve

Problems with an informal semantic understanding

- How can we be sure that what the programmer thinks the program does it what it actually does
- How can we prove that the program does what it is meant to do
- How can I be sure that my program executes identically when I run it on different machines - sometimes have to target OS or hardware

In computer science informality → ambiguity → divergence → errors
Computer Science turns to Mathematics to define different types of formal semantics

- Denotational Semantics - Meaning is given mathematically as a suitable mathematical structure, all about functions. Follow functions through to show the program is correct
- Operational Semantics - A program's meaning is given in terms of the steps of a computation the program makes when it runs. Looks at all the state changes
- Axiomatic semantics - A program's meaning is given indirectly in terms of a collection of logical properties it satisfies

5 Compilation vs interpretation

High level programming languages are designed to be used by humans ... but ultimately have to be executed by a computer

Complied languages

- All of the program is complied into **machine code** to be run on a **specific machine**. Converted to different machine code for intel vs amd etc

Interpreted Languages

- Only one instruction at a time is **translated**, as it is needed. With the translations interspersed with the activity of the program itself. Generally these are scripting languages

Comparison:

- Compiled programs are faster, don't have to keep going back to convert to MC
- Compilers spend time analysing/optimising
- Compilation avoids some run time errors better
- Interpreted programs use less memory in compilation, don't have memory overhead of compilation in terms of optimisation etc. This is only true during compilation. Compiled languages will probably use less memory when running
- Interpreted programs facilitate development. Faster to develop on, large compiled programs will take ages for compiling

Spectrum of possibilities:

- **Bytecode**: Java compiled into bytecode and then translated by a java virtual machine, half way between two

6 Compilation

6.1 character stream

The character stream is the input code that the programmer gives the machine

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

6.2 lex

The lexical analyser reads strings of symbols and converts them into basic syntactic components i.e. tokens in a token stream

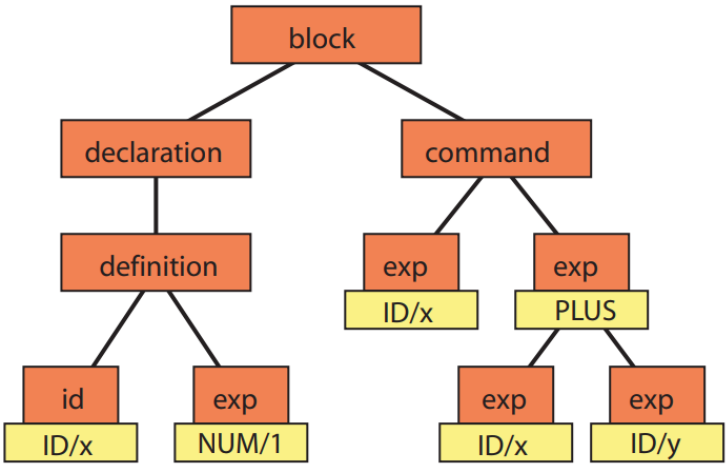
6.3 Token Stream

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

6.4 syn

The syntax analyser recognises the syntactic structure of the token stream and often results in a **parse tree**. This is where the optimisation starts to happen

6.5 parse tree



6.6 trans

Translation phase flattens the (modified) tree into a linear sequence of **intermediate code**

6.7 intermediate code

This is something like bytecode

6.8 cg

Code Generation phase converts the intermediate code into **assembly** and then into **machine code**. This optimises for the specific processor architecture

6.9 machine code

7 Lexical analysis

A **regular expression** over some **alphabet** Σ (finite set of symbols)

- Any a $\in \Sigma$ is a regular expression. A regular expression is any letter from the alphabet
- $\varnothing$ (empty set) and ϵ (empty string) are regular expressions
- If ω and ω' are regular expressions then so are:
  - $(\omega\omega')$ - x concatenated with y
  - $(\omega|\omega')$ - Logical or for example $gr|a|e|y$ would fit for both spellings of the colour
  - $(\omega*)$ - 0 or more copies of ω in a row

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

8 Some regular expressions

- Consider defining all strings over $\{a, b\}$ beginning with a and ending in b
  - $a(a|b)^*b$ - a at start and b at end with as many as and bs in the middle as needed
- Consider defining all strings over $\Sigma = \{a, b\}$ containing the string abab
  - $(a|b)^*abab(a|b)^*$
- Consider defining all strings over $\Sigma = \{a, b\}$ where no two a's appear consecutively
  - $(ab|b)^*(a|\epsilon)$
  - $b^*(abb^*)^*(a|\epsilon)$
  - There are different regular expressions that do the same thing, it is a difficult problem to decide whether two regular expressions describe the same set of strings
- Consider defining all strings over $\Sigma = \{a, b, c\}$ where there is an even number of c's
  - $(a|b)^*(c(a|b)^*c(a|b)^*)^*(a|b)^*$
- Regular expressions
  - Found in search engines, word processors, text editors, etc
  - Python provides regular expression support via the module re

9 Finite state machines

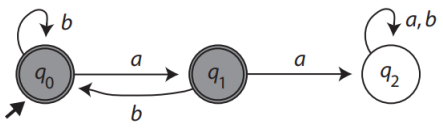
A **finite state machine (FSM)** $M = (\Sigma, Q, \delta : Q \times \Sigma \to Q, q_0 \in Q, F$ where:

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

On input a string $a_1, a_2...a_n$ over Σ, M yields a state sequence $q_0, q_1, ..., q_n$ via:

- $q_1 = \delta(q_0, a_1), q_2 = \delta(q_1, a_2), ...q_n = \delta(q_{n-1}, a_n)$
- $a_1, a_2, ...a_n$ is **accepted** if $q_n \in F$

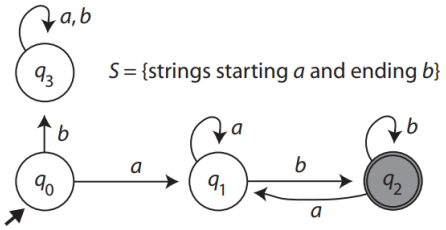
Pictorial representation: The FSM below accepts the set of strings not containing 2 consecutive a's



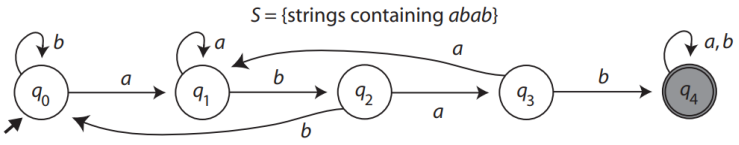
Theorem: A set of strings is denoted by a regular expression if, and only if, it is accepted by a finite state machine (such sets of strings are called the **regular languages**

9.1 Examples

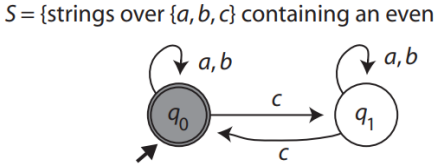
9.1.1 Example 1



9.1.2 Example 2



9.1.3 Example 3



10 Syntax analysis

- While tokens are defined using regular languages, programming language syntax is almost always describable by a **context-free grammar**
- Need to check that what has been written follows the rules of the language
- Some things not picked up by lexical analysis will be picked up by syntax analysis
- **A phase structure grammar** is a four tuple (N,T,s,R) where
  - T and N are disjoint finite sets of **terminal** and **non terminal** symbols
  - $s \in N$ is the start symbol
  - R is a finite set of productions or rules
- In a context free grammar, productions are of the form:
  - $b \to a_1 a_2 ... a_k$ with $k \geqslant 0, b \in N$ and $a_1, a_2, ..., a_k \in N \cup T$

Example: A production $b \to a_1 a_2 ... a_k$ can be **applied** to a string ϵ containing b via:

$$\omega = ...b'bb''... \Rightarrow = ...b'a_1 a_2 ... a_k b''$$

- A grammar **generates** the set of strings over T that can be obtained by repeatedly applying productions starting from the string s
- Note that we may have a choice of production to apply; that is, the process is **non deterministic**

11 A context free grammar

- Consider the context-free grammar (N={s,t,u}, T={a,b},s,R) with R consisting of the productions:

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

- We start with s; there are 3 choices in which production to apply
  - $s \to \epsilon$ and we are finished having generated ϵ
  - $s \to bs$
  - $s \to at$
- Perhaps we repeatedly apply $s \to bs$:
  - $s \to bs \to bbs \to bbbs \to ... \to b^n s$
  - And then $s \to at$ to get $b^n at$, or $s \to \epsilon$ to finish and generate b^n (where $n \geqslant 1$)
- So, if we haven't finished then we have derived $b^n at$ where $n \geqslant 0$
- Perhaps we apply $t \to bs$ to derive $b^n abs$ ... or maybe we apply $t \to \epsilon$ to generate $b^n a$ ... or maybe $t \to au$ to derive $b^n aau$ from whence we are stuck
- So if we haven't finished or aren't stuck then we've got $b^n abs$: go again
- Arguing in this way we generate the knowledge that you cannot have 2 consecutive as

12 Regular Grammars

- Regular grammars are 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 $a \in T, c \in N$
  - $b \to \epsilon$

12.1 Theorem

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

- So, we can build regular grammars generating the earlier sets of string
- There exist context-free languages that are not regular e.g., $\{a^n b^n : n \geqslant 1\}$
- There are special machines called **pushdown automata** that characterize the languages generated by context free grammars
  - essentially, finite state machines equipped with extra memory called a **stack**
- If the syntax of a programming languages can be described using a context free grammar then it turns out that we can quickly build parse trees and so efficiently spot programs that are incorrectly written

13 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 |
  - non-terminals appear inside <>
  - ::= is used instead of →

Uses ::= instead of →
Non terminals appear inside <>

For example $s \to at$ would become $< s >::= a < t >$