Australian National University

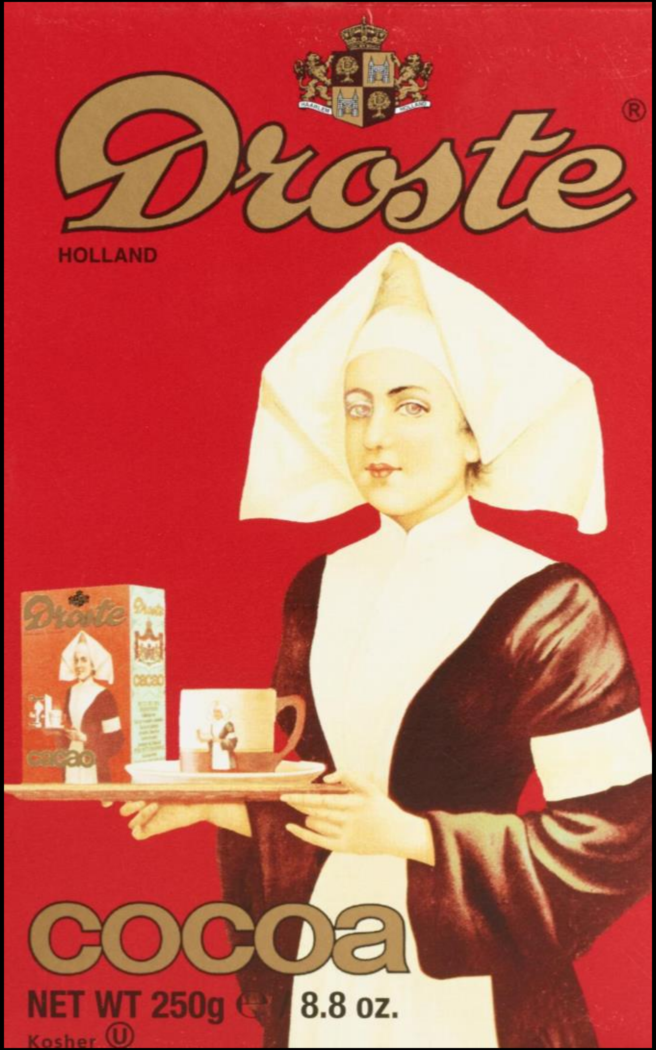
Recursion

C1

Recursive Algorithms

Australian National University

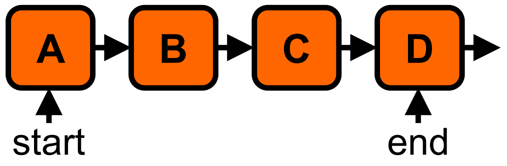
Australian
National
University



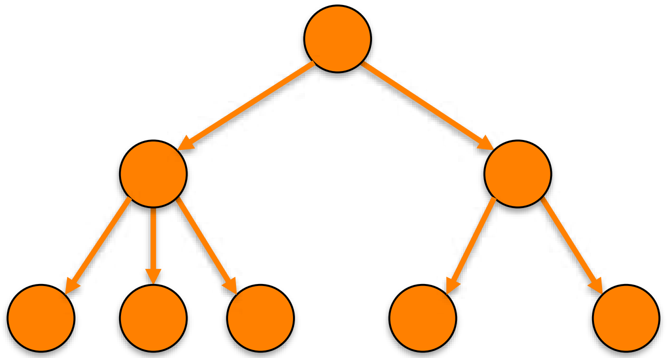
Austr
Natio
Unive

Recursive Data Structure

A recursive data structure is comprised of components that reference other components of the same type.



linked list



tree

Recursive Algorithms

A recursive algorithm references itself.

A recursive algorithm is comprised of:

- one or more base cases
- a remainder that reduces to the base case/s

Example: Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377…

fib(0) = 1 *(base case)*

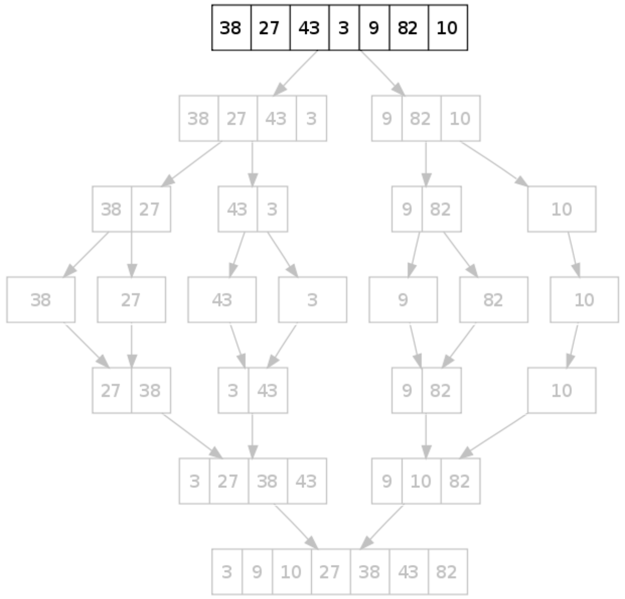
fib(1) = 1 *(base case)*

fib(n) = fib(n-1) + fib(n-2) *(for n ≥ 2)*

Example: Mergesort (von Neumann, 1945)

Sort a list

- List of size 1 *(base case)*
  - Already sorted
- List of size > 1
  - Split into two sub lists
  - Sort each sub list *(recursion)*
  - Merge the two sorted sub lists into one sorted list (by iteratively picking the lower of the two least elements)



Animation: Visualizing Algorithms, Mike Bostock, bost.ocks.org/mike/algorithms

Australian National University

Hash Functions

C2

Hash functions

Choosing a good hash function

Hash Functions

A hash function is a function $f(k)$ that maps a key, k, to a value, $f(k)$, within a prescribed range.

A hash is deterministic. (For a given key, k, $f(k)$ will always be the same).

Choosing a Good Hash Function

A good hash for a given population, P, of keys, $k \in P$, will distribute $f(k)$ evenly within the prescribed range for the hash.

A *perfect hash* will give a unique $f(k)$ for each $k \in P$

Australian National University

C3

Hashing Applications

Java hashCode()

Uses of Hashing

Java `hashCode()`

Java provides a hash code for *every* object

- 32-bit signed integer
- Inherited from `Object`, but may be overridden
- Objects for which `equals()` is **`true`** must also have the same `hashCode()`.
- The hash need not be perfect (i.e. two different objects may share the same hash).

Uses of Hashing

- Hash table (a map from key to value)
- Pruning a search
  - Looking for duplicates
  - Looking for similar values
- Compression
  - A hash is typically much more compact that the key
- Correctness
  - Checksums can confirm inequality

Practical Examples…

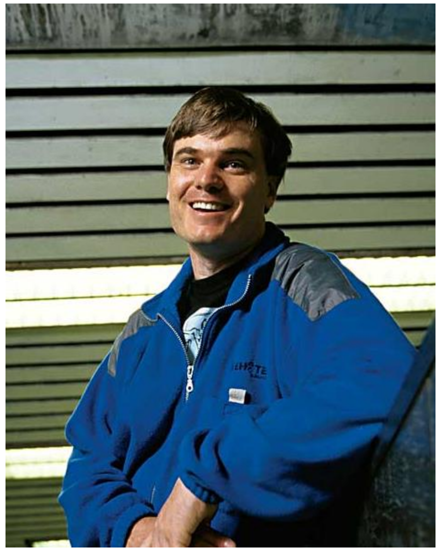
Luhn Algorithm

Used to check for transcription errors in credit cards (last digit checksum).

Hamming Codes

Error correcting codes (as used in EEC memory)

Practical Examples…

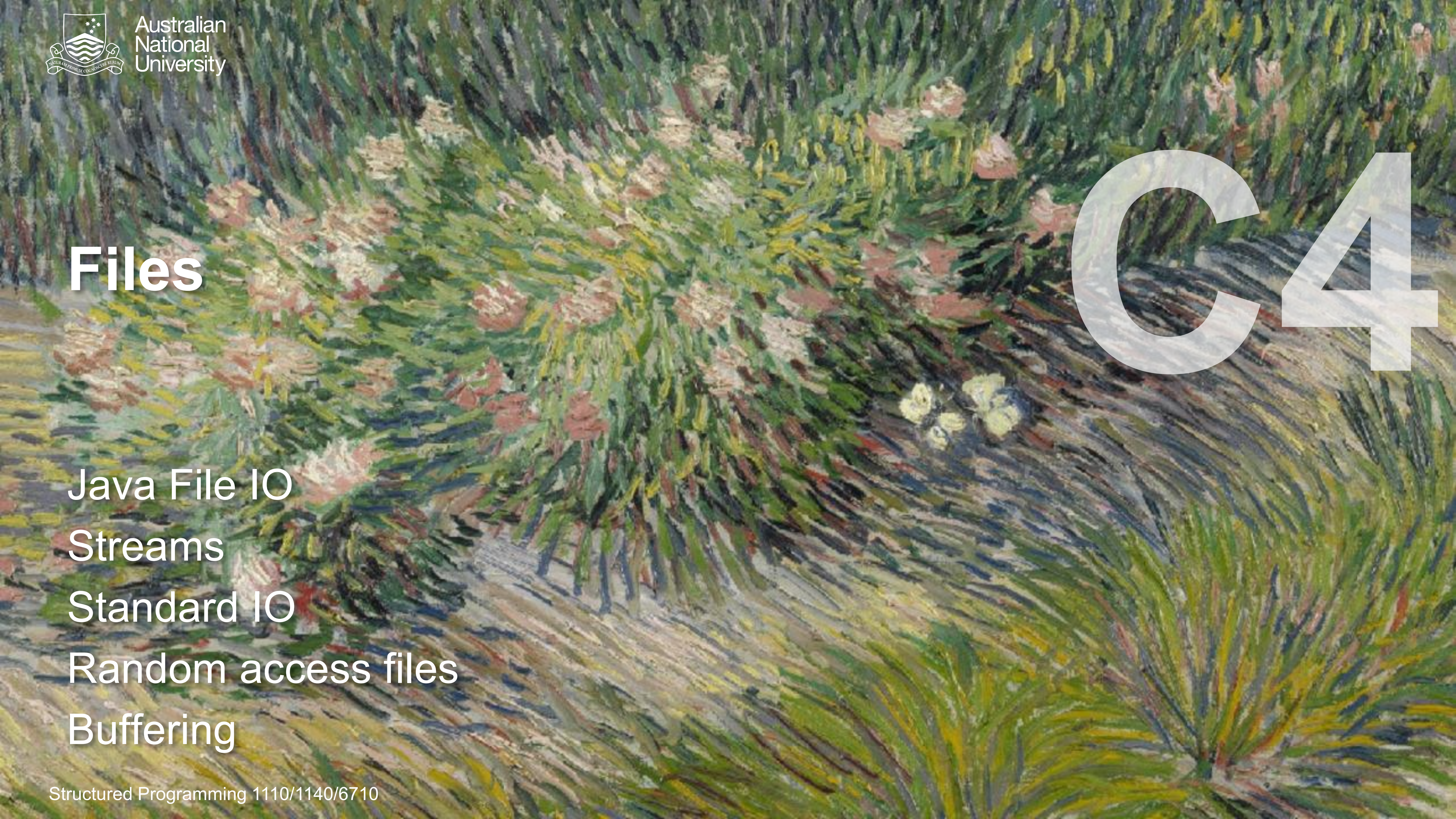


rsync (Tridgell)

Synchronize files by (almost) only moving the parts that are different.

MD5 (Rivest)

Previously used to encode passwords (but no longer).



C4

Files

Java File IO

Streams

Standard IO

Random access files

Buffering

File IO as Streams

A **stream** is a standard abstraction used for files:

A sequence of values are read.

A sequence of values are written.

The stream reflects the sequential nature of file IO and the physical characteristics of the media on which files traditionally reside (e.g. tape or a spinning disk).

A
maxell
UR
90
maxell
POSITION NORMAL

TDK
DATA CARTRIDGE
DC-6250
Cartouche Numérique
FUJIFILM
DG4-150
DataLife

Files C4

Java I/O: Byte Streams

The classes `InputStream` and `OutputStream` allow you to read and write streams of bytes to and from streams including files (subclasses: `FileInputStream` and `FileOutputStream`).

- Open the stream
- Read or write from the stream (in **`byte`**s)
- Wrap operations in a **`try`** clause
- Use **`finally`** to close the streams

`int`s are used, even though **`byte`**s are transferred(!)

Java I/O: Character Streams

When reading and writing characters, you should use the classes `Reader` and `Writer`, which allow you to read and write streams of characters to and from streams including files (subclasses: `FileReader` and `FileWriter`).

`int`s are used, even though `char`s are transferred.

File I/O: Buffering

Reading data one byte at a time is costly. Buffering is used to absorb some of that overhead.

Disk: ~10ms   SSD: ~100µs RAM: ~10ns   Register: ~1ns

In Java the `BufferedReader` and `BufferedWriter` classes can be used to buffer data read or written with `FileReader` and `FileWriter`.

To be sure that a buffer is flushed, call `flush()`, or close the file.

Java Command Line IO

Three standard IO streams (globally-defined objects):

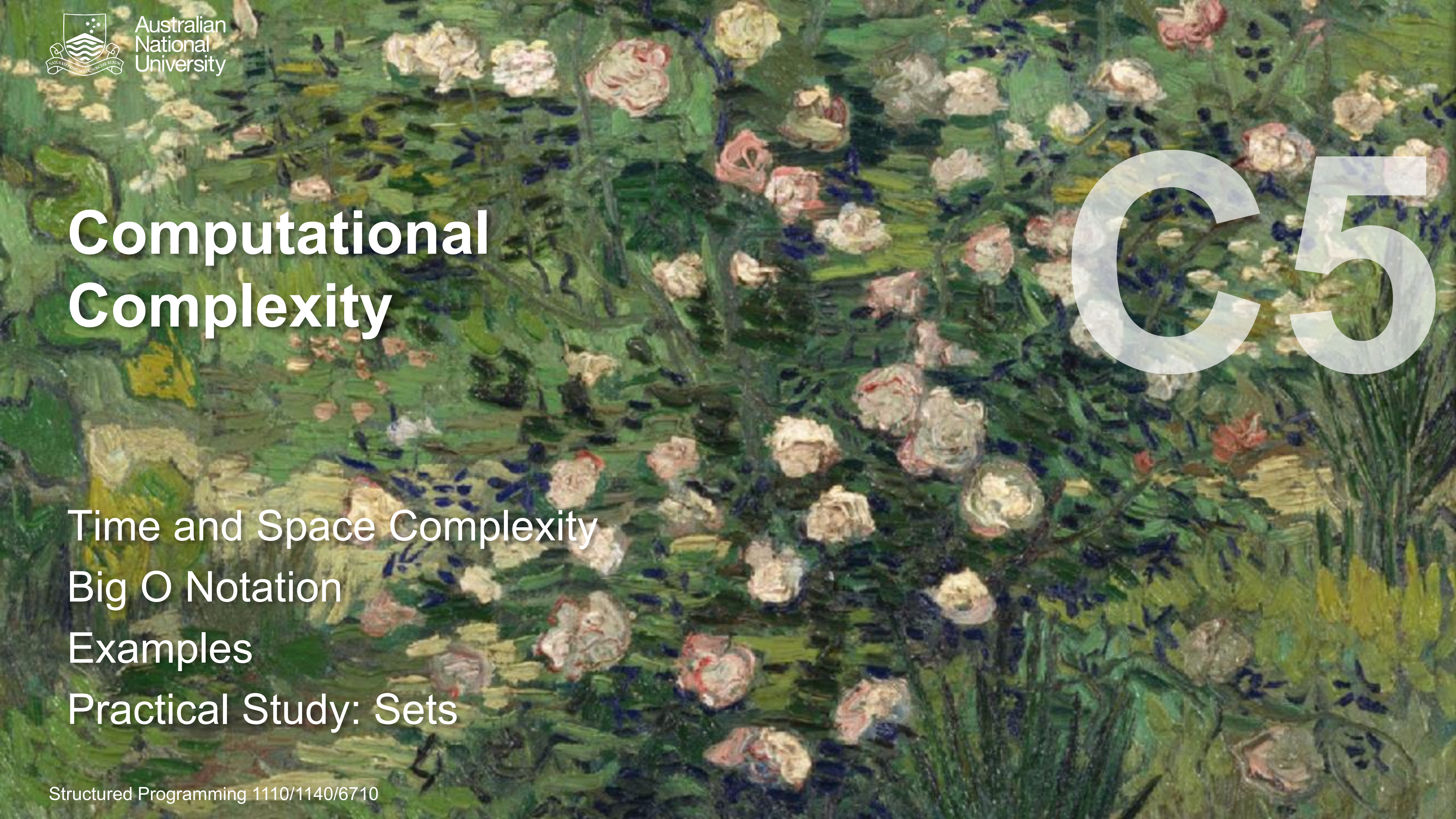
- Standard input `System.in`
- Standard output `System.out`
- Standard error `System.err`

```
byte b = (byte) System.in.read();
System.out.write(b);
System.out.flush();
System.err.write(b);
```

“New” I/O (`java.nio.file`)

Java NIO offers simpler, event-driven interface

- `Path` — replaces `java.io.File`
- `FileSystem` — factory class for objects in the filesystem
- `WatchService` — utility class to detect file system changes through event notification
- `Files` —create, rename, copy, modify attributes and delete files



Australian National University

Computational Complexity

C5

Time and Space Complexity

Big O Notation

Examples

Practical Study: Sets

Context

Key computational resources:

- Time
- Space
- Energy

Computational complexity is the study of how problem size affects resource consumption for a given implementation.

- Worst case
  - the complexity of solving the problem for the worst input of size *n*
- Average case
  - is the complexity of solving the problem on an average.

(Computational) Scaling

1. Identify *n*, the number that characterizes the problem size.
   – Number of pixels on screen
   – Number of elements to be sorted
   – etc.
2. Study the algorithm to determine how resource consumption changes as a function of *n.*

Big O Notation

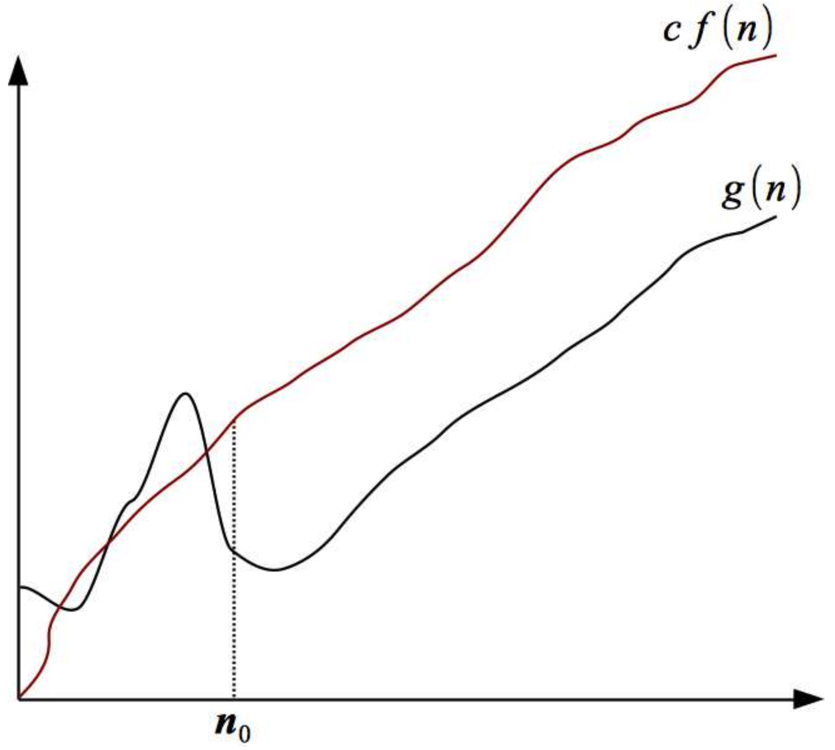
Suppose we have a problem of size *n* that takes *g(n)* time to execute in the average case.

We say:

$g(\mathrm{n}) \in O(f(n))$

if and only if there exists a constant $c > 0$ and a constant $n_0 > 0$ such that for all $n > n_0$:

$g(n) \leq \mathrm{c} \times f(n)$



Simple Examples

- Constant $O(1)$
  - Time to perform an addition
- Logarithmic $O(\log(n))$
  - Time to find an element in a (balanced) BST
- Linear $O(n)$
  - Time to find an element within a list
- $O(n \log(n))$
  - Average time to sort using mergesort
- Quadratic $O(n^2)$
  - Time to compare n elements with each other

Time Complexity: Counting Statements

Time complexity can estimated by simply counting the number of statements to be executed.

- Traps
  - Simple statements are constant time
  - Library calls may have arbitrary complexity

Concrete Examples

Consider hashing into a table of *n* elements…

```
public int hash(Integer key, int buckets) {
     return key % buckets;
}
```

Constant time, $O(1)$

Concrete Examples

Consider summing a list of size *n*…

```
public int sum(ArrayList<Integer> list) {
  int rtn = 0;
  for(Integer i: list) {
     rtn += i;
  return rtn;
}
```

Linear time, $O(n)$

Concrete Examples

```
public int minDiff(ArrayList<Integer> values) {
  int min = Integer.MAX_VALUE;                           1
  for (int i = 0; i < values.size(); i++) {             n
    for (int j = i + 1; j < values.size(); j++) {       (n – 1)n/2
      int diff = values.get(i)-values.get(j);           (n – 1)n/2
      if (Math.abs(diff) < min)                         (n – 1)n/2
        min = Math.abs(diff);                           (n – 1)n/2
    }
  }
}
```

$S(N) = 1 + n + 4\,((n - 1)\,n/2) = 1 + n + 2\,n^2 - 2n = 2n^2 - n + 1 \in O(n^2)$

Note: $n - 1 + n - 2 + \ldots 2 + 1 = (n - 1)\,n/2$

Australian National University

C6

Formal Grammars

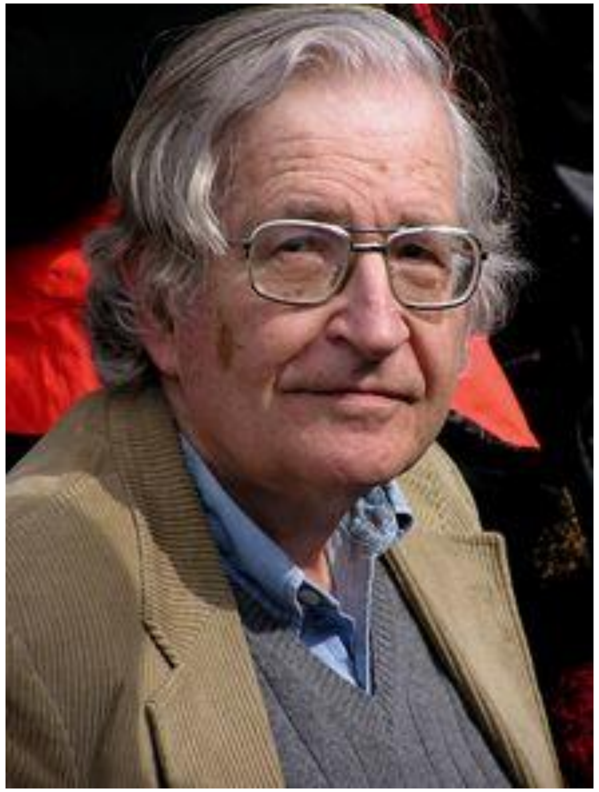
Grammars

EBNF

Formal Grammars

Formal languages are distinguished from natural languages by their artificial construction (rather than natural emergence).

Noam Chomsky is often credited with opening the field of formal grammars while studying natural languages.

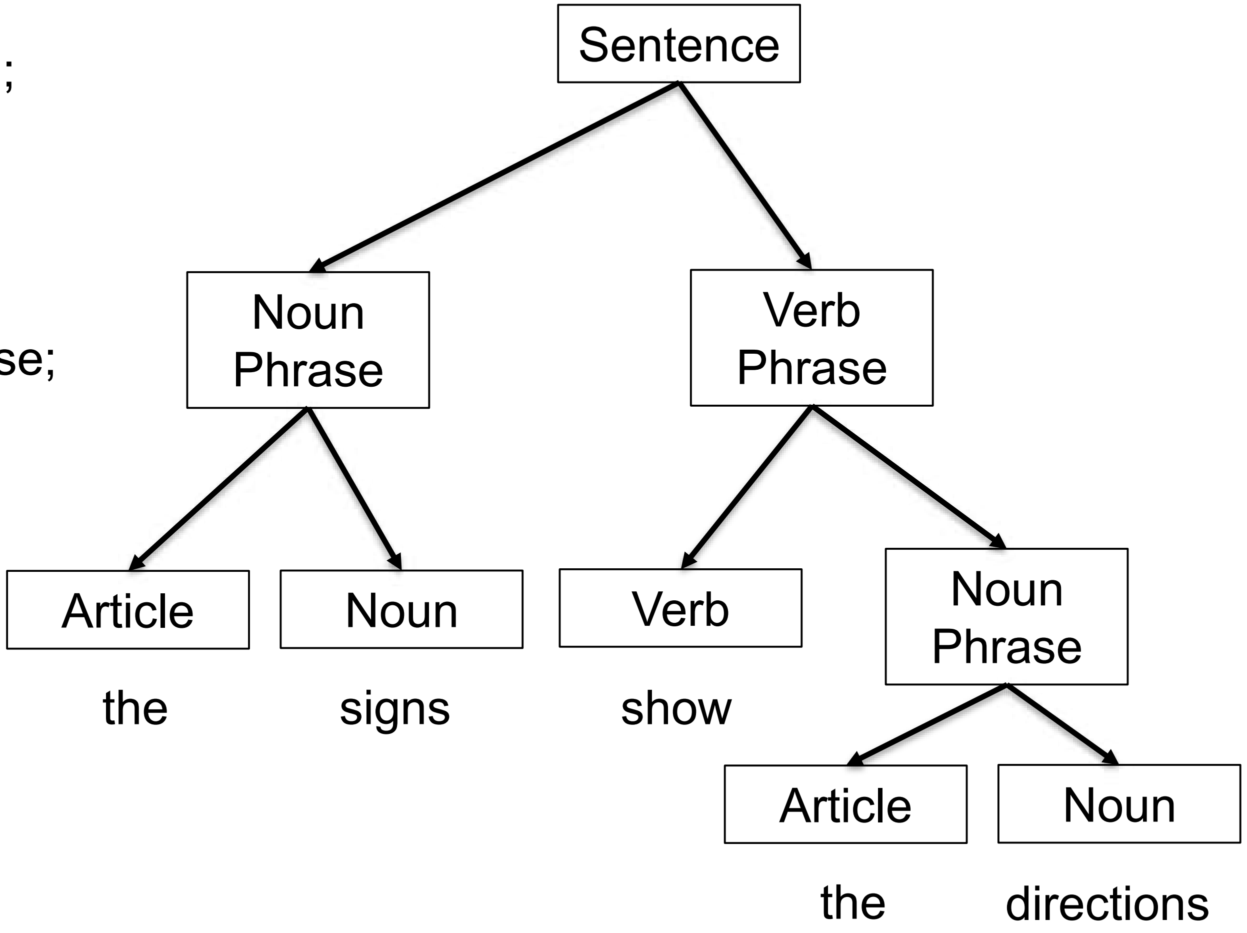

Duncan Rawlinson (Creative Commons)

Noam Chomsky

Generative Grammars

Sentence = Noun Phrase, Verb Phrase, [Noun Phrase];
Noun = signs, directions, lives
Article = the
Verb = show, matter, look
Adjective = big, small, white, black
Noun Phrase = [Article], [Adjective], Noun | Noun Phrase;
Verb Phrase = Verb, [Noun Phrase];

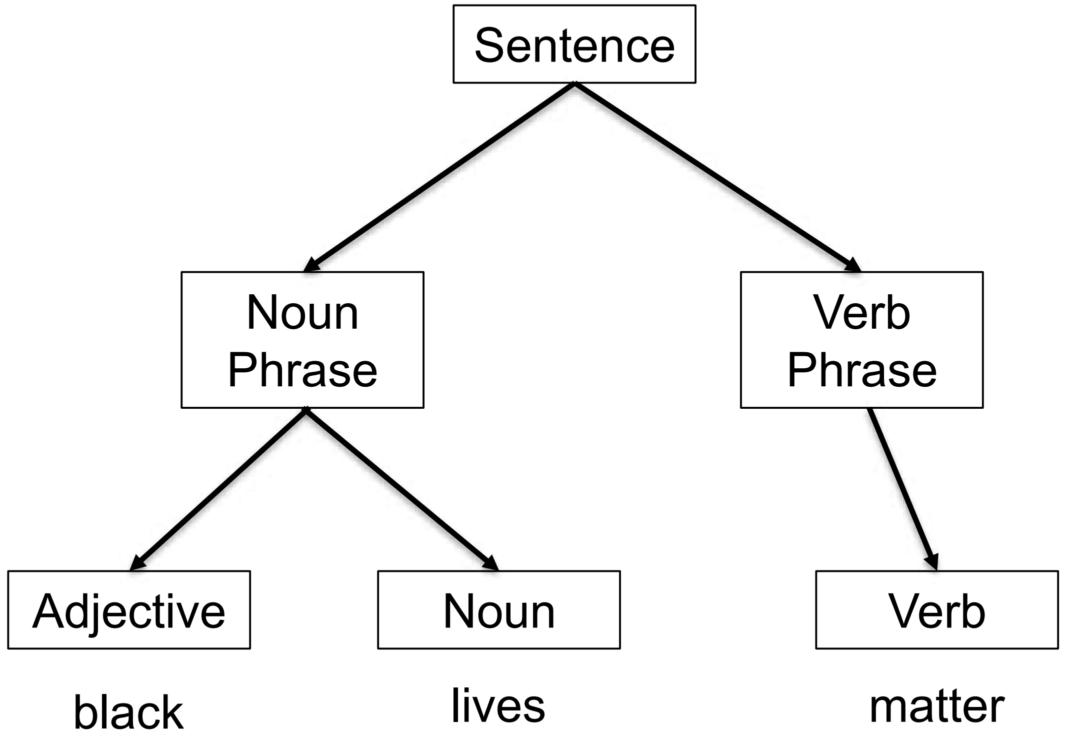
The signs show the directions.
Small big directions matter the black white signs.



Generative Grammars

Sentence = Noun Phrase, Verb Phrase, [Noun Phrase];
Noun = signs, directions, lives
Article = the
Verb = show, matter, look
Adjective = big, small, white, black
Noun Phrase = [Article], [Adjective], Noun | Noun Phrase;
Verb Phrase = Verb, [Noun Phrase];

The signs show the directions.
Small big directions matter the black white signs.



Syntactically correct productions (sentences) don't always convey meaning!
E.g. "I tested positively towards negative."

Extended Backus-Naur Form

EBNF is a standard way of representing the syntax of a formal language (but *not* the semantics!)

- Terminal symbols
  - e.g. characters or strings
- Production rules
  - combinations of terminal symbols

Robert McClure

Niklaus Wirth

Extended Backus-Naur Form

Very basic syntax of EBNF production rules:

- ‘=’ defines a production rule
- ‘|’ identifies alternates (e.g. ‘1’ | ‘2’ | ‘3’)
- ‘{’, ‘}’ identify expressions that may occur zero or more times (e.g. ‘1’, { ‘0’ })
- ‘[’, ‘]’ identify expressions that may occur zero or one time (e.g. ‘1’, [‘0’])
- ‘,’ identifies concatenation
- ‘-’ identifies exceptions
- ‘(’, ‘)’ identify groups
- ‘;’ terminates a production rule

Example EBNF grammar

```
PROGRAM DEMO1
BEGIN
  A0:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:="Hello world!";
END.
```

(* a simple program syntax in EBNF − Wikipedia *)

```
program = 'PROGRAM', white space, identifier, white space,
           'BEGIN', white space,
           { assignment, ";", white space },
           'END.' ;
identifier = alphabetic character, { alphabetic character | digit } ;
number = [ "-" ], digit, { digit } ;
string = '"' , { all characters - '"' }, '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

Simple EBNF Grammars

Grammar for arrangement of characters that are:

- Natural numbers?

```
natural = '0' | (nzdigit, { digit }) ;
nzdigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
digit = '0' | nzdigit ;
```

- Integers?

```
integer = '0' | (['-'], nzdigit, { digit }) ;
```

- Decimal numbers?

```
real = (['-'], natural, [('.' { digit }, nzdigit)]) – '-0' ;
```

- 24hr time, digital clock?

```
time = hour, ':', min ;
hour = ( ( '0' | '1' ) , digit ) | ( '2' , ( '0' | '1' | '2' | '3')) ;
min = ( '0' | '1' | '2' | '3' | '4' | '5' ), digit ;
```

Australian
National
University

C7

Threads

Concurrency

Concurrency, Processes and Threads

- Concurrency
  - Multiple activities *(appear to)* occur simultaneously, (e.g. recording this lecture and displaying this slide).
  - ‘Time slicing’ allows a single execution unit to give the appearance of concurrent execution
- Process
  - Distinct execution context that (by default) shares nothing (e.g. IntelliJ, PowerPoint, Quicktime recorder)
- Thread
  - Intra-process execution context (e.g. IntelliJ’s compiler)

Why Threads?

- ‘Concurrency’
  - Separate concerns (e.g. rendering v logic)
  - Good for: distinct tasks that naturally occur concurrently
- ‘Parallelism’ (a special case of concurrency)
  - Break task into pieces, exploit parallel hardware
  - Good for: computationally intensive problems that can be readily partitioned