



CS1Bh Revision — Part 1

Paul Jackson
 Computer Science
 School of Informatics
 The University of Edinburgh

13th May 2003

Review of CS1Bh

Algorithms	
Overview	1
Divide and Conquer	2.5
Greedy	3.5
Dynamic Programming	2,4,5
Tractability (limits)	21

CS fundamentals	
Induction	6
Invariants	6,19
Compilation	7,8
Logic	18,19
Prog. Langs: SML & C	26,27
Machines	23,24,25

Software Engineering	
Design	10
Development and Testing	11
Debugging	17
Computer Security	28
Client/Server	15,16

Java programming	
Interfaces, Abstract classes	9,22
Java Byte Code	7,8
Graphics, Applets	12,13
Event handling	14
Exceptions	20
Packages, Access control	22

TOPIC 1: Semantic Trees (Note 18)

Algorithm for checking if propositional formula is a tautology

Often more efficient than drawing truth table

To check formula A is a tautology, technique assumes it is false and explores what truth assignments must then be made to parts of A .

The tree checks *all* possibilities, so if no way is found to falsify A , A must be a tautology.

Otherwise, counter-example can be read off tree.

Similar algorithm can check if formula is satisfiable.

Semantic Tree Algorithm

To show A is a tautology:

1. Begin tree with node $f: A$.
2. Repeat until no further progress can be made:
 - (a) Choose undischarged node n , not a variable or a constant.
 - (b) Find rule for n and add the new nodes from rule to leaves of subtree below n .
 - (c) Mark node n as discharged
 - (d) Check every path through n for feasibility. Mark the path as closed (infeasible) if both $t: Q$ and $f: Q$ or both $t: f$ and $f: t$ occurs on it.
3. If all paths are closed, A is a tautology. Any open path gives a counter-example: look at values for variables on path.

Semantic Tree Rules

	\wedge	\vee	\neg	\Rightarrow	\Leftrightarrow
t	$t: A \wedge B$ $t: A$ $t: B$	$t: A \vee B$ $t: A$ $t: B$	$t: \neg A$ $f: A$	$t: A \Rightarrow B$ $f: A$ $t: B$	$t: A \Leftrightarrow B$ $t: A$ $f: A$ $t: B$ $f: B$
f	$f: A \wedge B$ $f: A$ $f: B$	$f: A \vee B$ $f: A$ $f: B$	$f: \neg A$ $t: A$	$f: A \Rightarrow B$ $t: A$ $f: B$	$f: A \Leftrightarrow B$ $t: A$ $f: A$ $f: B$ $t: B$

Semantic Tree Example 1

Is $(\neg P \vee Q) \Rightarrow Q$ a tautology?

Start a semantic tree with:

$$f: (\neg P \vee Q) \Rightarrow Q$$

Semantic Tree Example 2

Applying the rule for \Rightarrow being false, we get:

$$\begin{array}{c}
 * f: (\neg P \vee Q) \Rightarrow Q \\
 | \\
 t: \neg P \vee Q \\
 | \\
 f: Q
 \end{array}$$

Semantic Tree Example 3

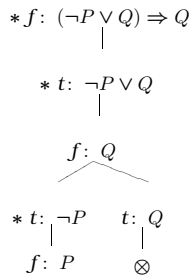
Now applying the rule for \vee true:

$$\begin{array}{c}
 * f: (\neg P \vee Q) \Rightarrow Q \\
 | \\
 * t: \neg P \vee Q \\
 | \\
 f: Q \\
 | \\
 t: \neg P \quad t: Q
 \end{array}$$

Semantic Tree Example 4

8

Finally, we apply the rule for \neg true and note that the right-hand path is contradictory and should be closed.



The left path is still open and the tree is finished, so a counter-example is $\{P \mapsto f, Q \mapsto f\}$

TOPIC 2: Predicate Logic (Note 19)

9

Syntax:

Terms built from

- *individual variables* (e.g. i, j)
- *constants* (e.g. $1, 2, 3$)
- *functions* (e.g. $- + -, - \times -$)

Formulas built from

- *Relations or predicates* on terms (e.g. $- = -, - < -, \text{prime}(-)$)
Can be *atomic* or *defined*.
- *Propositional logic connectives* (e.g. $- \wedge -, - \vee -, - \Rightarrow -, \neg$)
- *Quantifiers*
 - Universal: $\forall -, -$ (read as *for all* or *for every*),
 - Existential: $\exists -, -$ (read as *there exists*).

For example: $\forall i. \exists j. j \times j \leq i \wedge i < (j+1) \times (j+1)$

Quantifier Ranges 1

10

Set of values a quantifier ranges over is implicit in $\forall x. x^2 \geq 0$.

Can make range explicit: $\forall x: \mathbb{Z}. x^2 \geq 0$.
(for all *integers* x , we have $x^2 \geq 0$)

Subsets can be used for ranges.

If $\{i \dots j\}$ is an abbreviation for $\{k \in \mathbb{Z} \mid i \leq k \wedge k \leq j\}$, we can write:

$$\forall x: \{1 \dots 100\}. x^2 \geq 0,$$

An alternate way of writing this restricted range is:

$$\forall x: \mathbb{Z}. 1 \leq x \wedge x \leq 100 \Rightarrow x^2 \geq 0$$

Quantifier Ranges 2

11

With existentials, we use \wedge instead of \Rightarrow when eliminating subset notation.

$$\exists x: \{1 \dots 100\}. x^2 \geq 0,$$

is equivalent to

$$\exists x: \mathbb{Z}. (1 \leq x \wedge x \leq 100) \wedge x^2 \geq 0$$

TOPIC 3: Assertions

12

(Notes 6,19, Q Sheets 1,3)

Logical formulas can be used to assert facts about a program's state. They can document methods and clarify their operation.

```

public static int isum(int n) {
    int s = 0;
    // A: s == 0
    for(int i = 1; i <= n; i++) {
        // B: s == ((i-1)*i)/2
        s = s + i;
        // C: s == (i*(i+1))/2
    }
    // D: s == (n*(n+1))/2
    return s;
}

```

Assertion Kinds 2

14

```

/** requires n >= 0 ;
/** ensures \result == (\sum int k; 0 <= k & k <= n ; k);

public static int isum(int n) {
    int s = 0;

    /** loop_invariant s == (\sum int k; 0 <= k & k < i ; k);
    for(int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}

```

Here, using ESC/Java notation, **requires** introduces the pre-condition, **ensures** introduces the post-condition, and the notation $(\sum \text{int } k; 0 \leq k \wedge k < i ; k)$ is an ASCII version of

$$\sum_{k=0}^{i-1} k$$

Assertion Kinds

13

Special kinds include:

Pre-condition: what must be true when a method begins.

Post-condition: what should be true when a method finishes.

Loop Invariant: States what should be true just before every execution of a loop's test.

Checking Assertions

15

Options include:

Convert assertions into *run-time* checks using `assert` statements.

Use tool for checking assertions *statically* at compile time. Unfortunately existing tools (e.g. ESC/Java) are very limited, but are getting better.

Trace program executions between assertions. For any execution from some assertion 1 to some following assertion 2, assume assertion 1 is true and check that assertion 2 is true.

```
class A {
    private int[] x;

    /** requires x != null && x.length != 0;
     *  ensures
     *  (\exists int n; 0 <= n & n < x.length
     *  & \result == x[n])
     *  & (\forall int n; 0 <= n & n < x.length
     *  ==> \result <= x[n]);
     */

    public int findMin() {
        int min = x[0];
        for (int i = 1; i != x.length; i++) {
            if (x[i] < min) min = x[i];
        }
        return min;
    }
}
```

Induction Example

18

Consider proving that the static method

```
public static int rsum(int n) {
    if (n == 0)
        return 0;
    else
        return rsum(n - 1) + n;
}
```

satisfies the specification formula:

$$\forall n:\mathbb{N}. \text{rsum}(n) = n(n+1)/2$$

The basic rule of mathematical induction over the natural numbers $0, 1, 2, \dots$ says the following:

To show that any property $\phi(n)$ holds for all natural numbers n , it is sufficient to

1. prove a **base case** $\phi(0)$,
2. prove a **step case** that for all k , if $\phi(k)$ is assumed to hold, then $\phi(k+1)$ also holds.

When $\phi(k)$ is assumed in the step case proof, it is often called the *induction hypothesis*.

Induction Example cont.

19

Using induction on the natural number n we have

The base case:

We must prove that $\text{rsum}(0) = 0(0+1)/2$

Observing that $\text{rsum}(0)$ evaluates to 0, we see this is trivially true.

The step case:

we get to *assume* the induction hypothesis

$$\text{rsum}(k) = k(k+1)/2 \quad (\text{I.H.})$$

and we must *prove*

$$\text{rsum}(k+1) = (k+1)(k+2)/2. \quad (1)$$

We observe that $k+1 \neq 0$, so that $\text{rsum}(k+1)$ evaluates to $\text{rsum}(k) + k + 1$.

Using this fact and the inductive hypothesis (I.H.), (1) reduces to:

$$k(k+1)/2 + k + 1 = (k+1)(k+2)/2$$

which is simple arithmetic.

TOPIC 5: Interfaces and Abstract Classes (Note 9)

20

Interfaces and Implementations

An interface defines a *behaviour* of, or a *protocol* for interaction with, an object.

Interfaces are *implemented* by classes.

A particular behaviour or protocol can be common to many (otherwise unrelated) classes.

Example: the `Comparable` interface which declares the `compareTo` method. Many classes implement this interface. The meaning of the `compareTo` is (informally) defined in the interface by documentation.

Interfaces and Polymorphism

22

Using interfaces as types allows flexibility for polymorphism.

With class inheritance: when a parameter `AClass` `a` appears in some method header, we can use any argument from *any subclass* of `AClass`.

With interfaces: when an interface `AnInterface` `a` appears in some method header, we can use any argument from *any implementing class* of `AnInterface`.

A Simple Interface

21

```
interface Emailable {
    void sendEmail(String msg); // Send me an email message
}

class Student implements Emailable {
    String forename;
    String surname;
    String matricno;
    // Send an email message to SMS account of student
    public void sendEmail(String msg) {
        System.out.println("To: " + matricno +
            "@sms.ed.ac.uk" + "\ n" + msg);
    }
}
```

Abstract Classes and Interfaces

23

An *abstract class* is something between an interface and an ordinary class. Some of the methods may be declared with the **abstract** modifier, in which case no method body is given. (In an interface, all methods are **abstract**, so the modifier is optional).

A class (abstract or concrete) extends exactly one direct superclass. This is the usual class hierarchy: the top class is `Object`.

Like interfaces, abstract classes cannot be instantiated. But interfaces have their own hierarchy, separate from the class hierarchy. An interface can extend zero or more other interfaces. If an interface `I` extends `J`, `K`, then any class which implements `I` must also implement `J` and `K`.

```

abstract class ConnectedPerson implements Messagable {
    public void sendMessage(int priority, String msg) {
        switch (priority) {
            case URGENT_PRIORITY: sendMessage(msg); break;
            case MEDIUM_PRIORITY: sendEmail(msg); break;
            default: sendFax(msg);
        }
    }
    abstract public void sendFax(String msg);
    abstract public void sendEmail(String msg);
    abstract public void sendTextMessage(String msg);
}

```

Engineering Issues

26

Interfaces can be used when code has not yet been written. They can form part of a contract or requirements specification between a client and supplier. They are also helpful when several workers are building a system.

Interfaces and abstract classes help foster *reuse*. Using standardized interfaces reduces familiarization overhead

A library can offer functionality using several different underlying data structures or algorithms in a clean way, by extending an abstract class. Think of abstract classes as *placeholders* in the inheritance hierarchy, providing templates for implementing concrete classes.

TOPIC 6: Exceptions (Note 20)

28

An exception (informally) is an event that occurs during the execution of a program, which disrupts the normal flow of instructions. Many kinds of error can cause exceptions—problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element.

Java provides language constructs for managing exceptions. These allow us to **declare** exceptions, **throw** exceptions, and **handle** exceptions.

Reasons for a dedicated exception mechanism: separating error code, propagation of errors up the call stack, organising error types.

Checked exceptions and the `throws` clause

30

Exceptions are split into **checked** and **unchecked** exceptions. Runtime exceptions and errors are unchecked, all others are checked.

The checked exceptions are ones which the Java compiler will force you either to handle with **try-catch-finally**, or to “duck” by specifying that your method **throws** that exception.

```

public String getWord() throws FileNotFoundException {
    ...
}

```

Notice that you may need to specify **throws** in your method even if you do not explicitly **throw** the exception yourself. An exception must be specified whenever some method called within the scope of your method may throw that exception.

We use **extends** for inheritance ...

```

class MyClass extends YourClass {
    ...
}

```

and **implements** for interface satisfaction:

```

class MyClass implements YourInterface, HerInterface {
    ...
}

```

Implementation of several interfaces partly substitutes for the possibility of *multiple inheritance* which Java does not allow. Java only has *single inheritance*.

The Java API

27

Interface `Comparable` — specifies the comparison relation `compareTo`. Implemented by many classes.

Interface `Collection` — specifies a notion of a group of objects, with methods like `add`, `contains`, `isEmpty`, `Iterator`. Implemented by `List`, `Set`, `AbstractCollection`, amongst others.

Abstract class `AbstractCollection` — contains an implementation of the `Collection` interface in terms of the abstract methods `iterator` and `size`.

Organization of Exceptions in Java

29

Java organizes the builtin exceptions in `java.lang`:

- All exceptions are subclasses of `Throwable`
- Exceptions are split into **errors**, subclasses of `Error`, and **exceptions**, subclasses of `Exception`.
- Exception has a special subclass `RuntimeException`

Errors are usually fatal problems which cannot be rectified, (e.g. `LinkageError`, `VirtualMachineError`).

Exceptions are problems which can potentially be caught. But *runtime exceptions* are problems which can be raised practically anywhere (e.g. `ArithmeticException`, `IndexOutOfBoundsException`, `NullPointerException`).

Creating new exceptions

31

We can declare our own new exceptions simply by defining a class which **extends** one of the supplied base exception classes, usually the `Exception` class.

Here's an example which includes a constructor which is passed a string argument to create a customized error message:

```

public class WrongNumberException extends Exception {
    WrongNumberException(String message) {
        // Construct an exception message for
        // the toString method, using the given
        // message and the super-class constructor.
        super("Wrong number: " + message);
    }
}

```

To throw an exception, we use Java's `throw` statement:

```
throw someThrowableObject;
```

The throwable object is an instance of any subclass of the `Throwable` class. The constructor for the subclass may well include arguments. A typical example is to include an error message describing the exceptional circumstance.

```
if (args.length != 3)
    throw
    new WrongNumberException("3 command line args needed.");
```

Exceptions are handled with `try`, `catch`, `finally`. Notice that the `finally` block is always executed, however `try` exits: normally; with a caught/uncaught exception; or with `break`, `continue`, `return`.

Successive `catch` clauses are tested against an exception using the exception hierarchy.

```
try {
    // Some code that may raise exceptions
}
catch (SomeException e1) {
    // Process e1
}
catch (SomeOtherException e2) {
    // Process e2
}
finally {
    // Code that is always executed
}
```

TOPIC 7: Software Engineering in Java (Note 22)

Issues discussed in this note included:

- Packages
- Inheritance
- Polymorphism
- Abstract classes and interfaces
- Access control
- Encapsulation

Overriding and `super`

If a method header is repeated in a subclass, the new method *overrides* the method from the superclass. We can still access the superclass method using the keyword `super`.

```
class YourClass {
    int i = 1;
    int f() { return 2*i; }
}

class MyClass extends YourClass {
    int f() { return -(super.f()); }
}
```

Overriding fields is possible but not recommended. It's called *shadowing*.

Polymorphism and `instanceof`

A common use of inheritance is to allow *polymorphism*. A *polymorphic data structure* is one which can store objects of any class. The collections classes are typical examples.

When we extract an object `o` from a collection class, to begin with we only know that it belongs to the `Object` class. We can use `instanceof` to find out if it belongs to a particular class:

```
if (o instanceof Double) {
    Double d = (Double) o;
    ...
}
```

Access control for class members

Hiding helps ensure *encapsulation* — insulation of an object from its surroundings. Hiding for class members is achieved using Java's *visibility modifiers*: `public`, `private`, and `protected`.

There are four different visibility levels, in order of increasing restriction: `public`, `protected`, `package`, and `private`.

	defining class	class in same package	subclass in other package	non-subclass in other package
public	✓	✓	✓	✓
protected	✓	✓	✓	×
package	✓	✓	×	×
private	✓	×	×	×

Encapsulation in Java

We can use access controls to achieve encapsulation by:

- hiding fields (making them `private` or `protected`)
- providing public, validating *constructors* and *accessors*

Constructors typically set fields of the object; they should validate to be sure that the settings are in the correct ranges.

An *accessor* method is one which either *gets* or *sets* some fields which are hidden. By convention, the name often starts with `get` or `set`.

Access controls for packages

A package is accessible to any classes within it, and to those outside it whenever it is accessible on the host file system on a path known to the compiler — package-level access is not controlled by the Java language itself.

A dotted name such as `java.util.Date` or an `import` directive is used to refer to something from another package:

```
import java.util.Date;
```

However, there are some engineering disadvantages with using `import...`

To specify that our class is part of a package, we use the `package` directive, which must appear as the first statement in the Java file:

```
package uk.ac.ed.dcs.cs1.studentdata;
```

The convention for naming packages is to use a reversed domain name, with the name of the package as the last component.

Most implementations of the Java platform map package names to a hierarchical directory structure, meaning that the `studentdata` package must be stored in a directory ending in `uk/ac/ed/dcs/cs1/studentdata`.

A top-level class is always accessible within its package. It will be accessible outside its package as well only if it is declared with the `public` modifier:

```
public class Student { ... }
```

Of course, the files comprising the package must be made available to others for them to use our classes.
