Implementation of tree traversals

Some considerations

- tree traversals allow computations on trees by (recursively) visiting their nodes
- in other words, a tree traversal implements some kind of operation on trees
- the visit of different types of nodes require different code

A couple of examples

- operations on a file system
  - computes the size of files/folders
  - find files in (sub)folders
- operations on Abstract Syntax Trees
  - returns the string corresponding to an expression in prefix notation
  - computes and returns the value of an expression

Code structured by operations versus by data

- functional programming favors code structuring by operations
- object-oriented programming favors code structuring by data

	data →	**Square**	**Circle**	**Rectangle**
operations ↓		*class implementing Shape* ↓	*class implementing Shape* ↓	*class implementing Shape* ↓
area	*function defined by pattern matching* →	Square area	Circle area	Rectangle area
perimeter	*function defined by pattern matching* →	Square perimeter	Circle perimeter	Rectangle perimeter

Example of tree traversals in F# (by operations)

```
type FileSysTree =
    | File of name: string * size: int
    | Folder of children: FileSysTree list

let rec size = (* the size of a file/folder *)
    function
    | File(_, size) -> size
    | Folder children -> List.fold (fun acc node -> acc + size node) 0 children

let rec find name = (* true if there is a file named 'name' *)
    function
    | File(fname, _) -> fname = name
    | Folder children -> List.exists (find name) children
```

Example of tree traversals in Java (by data)

```
public interface FileSysTree {
    int size();
    boolean find(String name);
}
public class File implements FileSysTree { // nodes of type File
    private String name; private int size;

    public File(String name, int size){...}
    public int size(){return size;}
    public boolean find(String name){return this.name==name;}
}
public class Folder implements FileSysTree { // nodes of type Folder
    private final List<FileSysTree> children = new LinkedList<>();

    public Folder(FileSysTree... children) {...}
    public int size(){
        var res = 0;
        for (var node : children) res += node.size();
        return res;
    }
    public boolean find(String name){
        for (var node : children) if(node.find(name)) return true;
        return false;
    }
}
```

Tree traversals by data versus by operations in OOP

Two approaches to implement tree traversals in OOP

- traversal by data: object methods for visiting nodes are grouped by classes (columns in the table below). This is the standard OOP approach
- traversal by operation: object methods for visiting nodes are grouped by operations (rows in the table below)
  This is a more complex approach based on the visitor pattern

A pictorial view

	node type 1	**node type 2**	**...**
operation type 1	*method to visit nodes of type 1 for op. of type 1*	*method to visit nodes of type 2 for op. of type 1*	...
operation type 2	*method to visit nodes of type 1 for op. of type 2*	*method to visit nodes of type 2 for op. of type 2*	...
...	...	...	...

Traversal structured by data

Pros

- new types of tree nodes can be added without modifying existing code
- simpler and slightly more efficient solution

Cons

- the object methods for visiting nodes are scattered all over the classes implementing the different types of node
- defining new types of traversal requires modification of all the classes implementing the different types of node
- less general solution

Traversal structured by operations

Pros

- new types of traversal can be implemented without modifying existing code
- the object methods for visiting nodes are contained in the single class that implements the specific traversal
- more general solution

Cons

- adding new types of nodes requires modification of all the classes implementing the different types of traversals
- more complex and slightly less efficient solution

Visitor pattern

The visitor pattern: OO implementation of traversal structured by operations

Main ingredients

Based on two mutually recursive types (defined with interfaces in Java)

- the type of (nodes of the) trees (`FileSysTree` in the example)

```java
public interface FileSysTree {
    // unique generic object method
    <T> T accept(Visitor<T> v);
}
```

- the generic type `Visitor<T>` of traversals (= operations on trees)

```java
public interface Visitor<T> {
    // an object method for each type of node
    T visitFile(String name, int size);

    T visitFolder(List<FileSysTree> children);
}
```

Visitor pattern

Remarks

- `T` in `Visitor<T>` is the type of the results returned by the traversals

  Examples:
  - the traversal to compute `size` implements `Visitor<Integer>`
  - the traversal to compute `find` implements `Visitor<Boolean>`
- for the visitor pattern the type of (nodes of the) trees has to declare only one method:

```
<T> T accept(Visitor<T> v);
```

- `Visitor<T>` declares one method for each type of node:

```
T visitFile(String name, int size);
T visitFolder(List<FileSysTree> children);
```

- the parameters of the methods in `Visitor<T>` correspond to the object fields of the visited node

Visitor pattern

Implementation of method `accept`

```
public class File implements FileSysTree { // nodes of type File
    private private String name; int size;

    public File(String name, int size){...}

    public <T> T accept(Visitor<T> v) { return v.visitFile(name, size); }
}

public class Folder implements FileSysTree { // nodes of type Folder
    private final List<FileSysTree> children = new LinkedList<>();

    public Folder(FileSysTree... children) {...}

    public <T> T accept(Visitor<T> v) { return v.visitFolder(children); }
}
```

Visitor pattern

Remarks on method `accept`

- `accept()` delegates the corresponding method of the visitor
- a mechanism of double dynamic dispatch is implemented:
  - first, method `accept()` is called, and the implementation in the class of the node is executed
  - then, the corresponding visit method is called on the visitor, and the implementation in the class of the visitor is executed
- no change to `accept()` is needed when a new traversal is implemented
- the content of the private fields of nodes is directly passed to the methods of the visitor: no getters are needed by visitors to retrieve the data of the visited nodes

Visitor pattern

Implementation of traversal for operation `size`

```
public class Size implements Visitor<Integer> {

    public Integer visitFile(String name, int size) {return size;}

    public Integer visitFolder(List<FileSysTree> children) {
        var res = 0;
        for (var node : children)
            res += node.accept(this);
        return res;
    }

    public static void main(String[] args) { // just for testing
        var folder = new Folder(new File("a", 10), new Folder(new File("b",2),
             new File("c",21)));
        assert folder.accept(new Size()) == 33; // 10+2+21
    }
}
```

Visitor pattern

Remarks on the visitor methods

- the whole implementation of the traversal is confined in a single class
- in `visitFolder` the traversal is propagated with `node.accept(this)`:
  - `node` may contain a `File` or a `Folder` object
  - it is not possible to statically decide whether `visitFile` or `visitFolder` is called
- with the double dynamic dispatch mechanism the right visit method is called:
  - `node.accept(this)` passes the current visitor **`this`** $= v$ which implements the traversal
  - the right `accept` method is called thanks to dynamic dispatch
  - `accept` calls the right visit method of v thanks to dynamic dispatch

Visitor pattern

Implementation of traversal for operation `find`

```
public class Find implements Visitor<Boolean> {
    private final String name; // name to find

    public Find(String name){this.name = requireNonNull(name);}

    public Boolean visitFile(String name, int size) {return this.name==name;}

    public Boolean visitFolder(List<FileSysTree> children) {
        for (var node : children) if(node.accept(this)) return true;
        return false;
    }

    public static void main(String[] args) { // just for testing
        var folder = new Folder(new File("a", 10), new Folder(new File("b",2),
             new File("c",21)));
        assert folder.accept(new Find("c"));
        assert ! folder.accept(new Find("d"));
    }
}
```

Remark: parameters of the tree operations are handled with visitor fields

Object-oriented implementation of an interpreter

Two traversals of the Abstract Syntax Tree have to be implemented:

- one for the static semantics: operation typecheck
- one for the dynamic semantics: operation execute/eval

First solution: traversals structured by data (outlined)

The F# declarations

```
typecheckStmt: staticEnv -> stmt -> staticEnv
typecheckExp: staticEnv -> exp -> staticType
executeStmt: dynamicEnv -> stmt -> dynamicEnv
evalExp : dynamicEnv -> exp -> value
```

in Java becomes

```
public interface Stmt extends AST{
   Type typecheck(StaticEnv env);
   void execute(DynamicEnv env);
}
public interface Exp extends AST{
   void typecheck(StaticEnv env);
   Value eval(DynamicEnv env);
}
```

Object-oriented implementation of an interpreter

Remarks

- `StaticEnv` and `DynamicEnv` are abbreviations for `GenEnvironment<Type>` and `GenEnvironment<Value>`
- the methods do not need to return environments as results:

  differently to F#, in Java environments are mutable objects

Object-oriented implementation of an interpreter

First solution: traversals structured by data (outlined)

Implementation of the methods scattered all over the classes of the nodes

```
public class PrintStmt implements Stmt {
   private final Exp exp;
   ...
   @Override public void typecheck(StaticEnv env){...}
   @Override public void execute(DynamicEnv env){...}
}
public abstract class BinaryOp implements Exp {
   protected final Exp left;
   protected final Exp right;
   ...
}
public class Mul extends BinaryOp {
   ...
   @Override public AtomicType typecheck(StaticEnv env){...}
   @Override public IntValue eval(DynamicEnv env){...}
}
```

Object-oriented implementation of an interpreter

Solution with the visitor pattern: traversals structured by operations (outlined)

```
public interface AST {
   <T> T accept(Visitor<T> visitor);
}
public interface Visitor<T> {
   T visitPrintStmt(Exp exp);
   T visitAdd(Exp left, Exp right);
   ...
}
public class PrintStmt implements Stmt {
   private final Exp exp;
   ...
   @Override public <T> T accept(Visitor<T> visitor) {
     return visitor.visitPrintStmt(exp);
   }
}
public abstract class BinaryOp implements Exp {
   protected final Exp left;
   protected final Exp right;
   ...
}
public class Mul extends BinaryOp {
   ...
   @Override public <T> T accept(Visitor<T> visitor) {
    return visitor.visitMulStmt(left,right);
   }
}
```

Object-oriented implementation of an interpreter

Solution with the visitor pattern: traversals structured by operations (outlined)

Implementation of the typechecker:

```
public class Typecheck implements Visitor<Type> {
   private final StaticEnv env = new StaticEnv(); // initially empty

   @Override
   public Type visitPrintStmt(Exp exp) {
      exp.accept(this);
      return null;
   }
   @Override
   public AtomicType visitMul(Exp left, Exp right) {
      checkBinOp(left, right, INT);
      return INT;
   }
   ...
}
```

Object-oriented implementation of an interpreter

Solution with the visitor pattern: traversals structured by operations (outlined)

Implementation of the interpreter:

```
public class Execute implements Visitor<Value> {
   private final DynamicEnv env = new DynamicEnv(); // initially empty
   private final PrintWriter printWriter; // output stream used to print values

   @Override
   public Value visitPrintStmt(Exp exp) {
      printWriter.println(exp.accept(this));
      return null;
   }
   @Override
   public IntValue visitMul(Exp left, Exp right) {
      return new IntValue(left.accept(this).toInt()*right.accept(this).toInt());
   }
   ...
}
```