Propositional formulas
Visitor Pattern

Tutorial 10 (20th April 2021)

Ike Mulder

$((A \wedge B) \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$

Radboud University

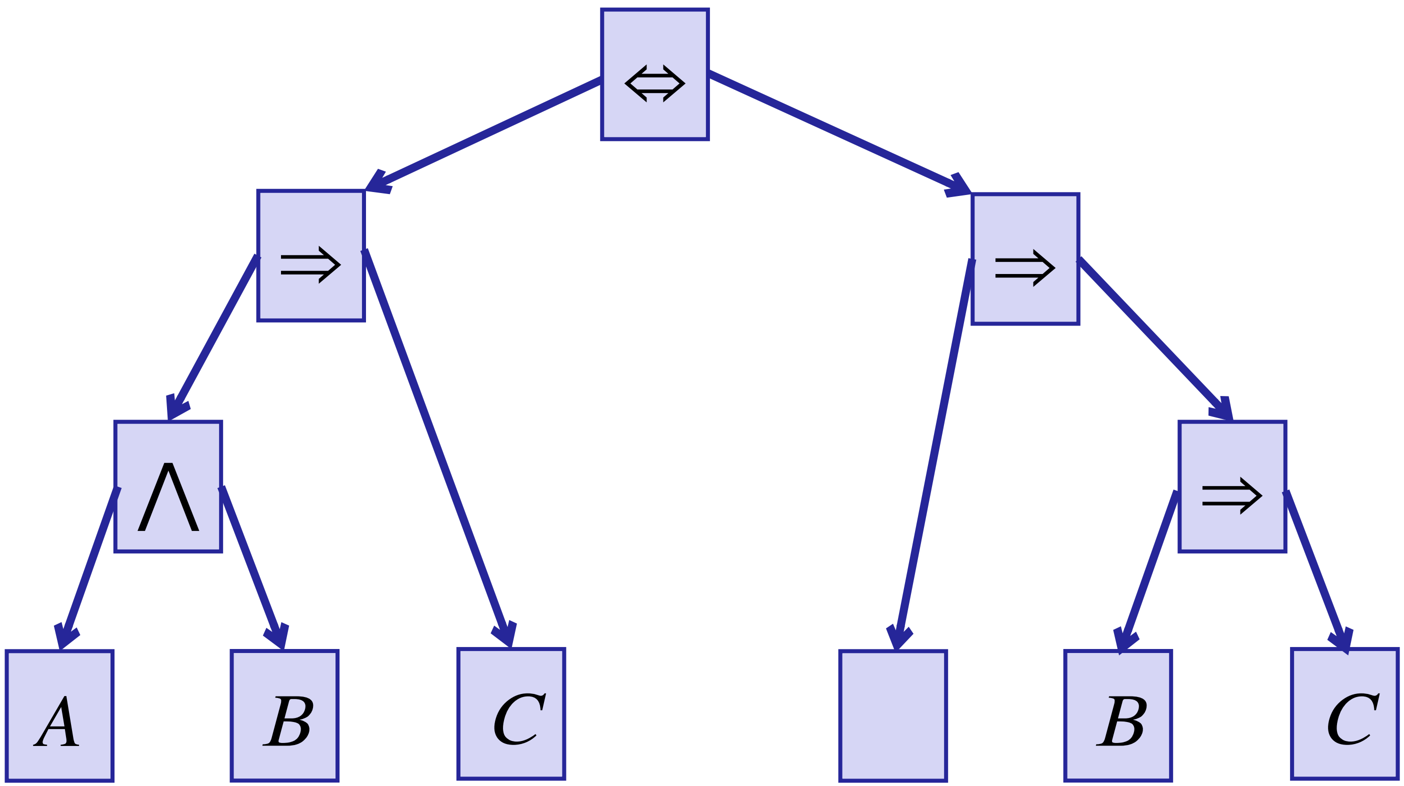
Propositional Logic

$$((A \wedge B) \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$$

$$F ::= \textbf{true} \mid \textbf{false} \mid \text{Atomic} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

Formulas as trees

$((A \wedge B) \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$

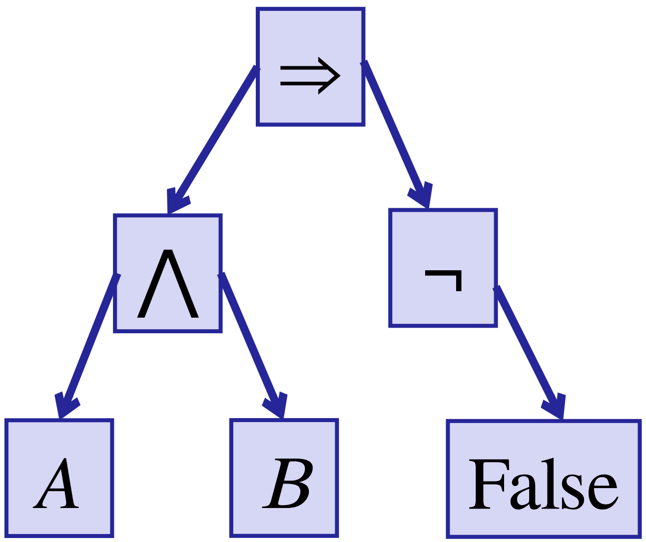


Formulas as trees

$$F ::= \textbf{true} \mid \textbf{false} \mid \text{Atomic} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

What kind of nodes?

- Nodes with two children:
  - $\Rightarrow$
  - $\wedge$
  - $\vee$
- Nodes with one child:
  - $\neg$
- Leaves:
  - Atomic names: A, B, …
  - Constants: True, False



Operations on formulas

We have a digital representation of formula’s, now what?

Recursive operations on formula’s:

- Human readable representation (exercise)
- Evaluate truth value (exercise)
- Collect atoms
- Count nodes
- As NAND formula
- Satisfiable?
- Tautology?

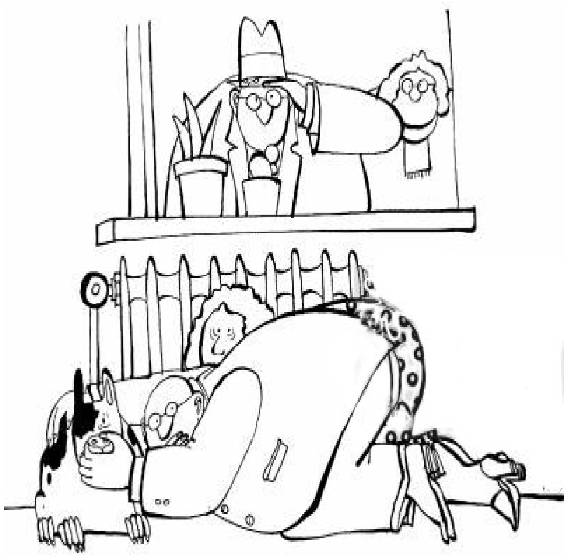
How to create new recursive operations WITHOUT changing our Formula implementation?

Evaluating Formulas

```
public boolean eval( Form f ) {
  if ( f instanceof BinOpForm ) {
    BinOpForm bof = (BinOpForm) f;
    BinOp bo = bof.getOp();
    switch( bo ){
      case AndOp:
          return eval(bof.getLeft()) && eval(bof.getRig
      case OrOp:
          return eval(bof.getLeft()) || eval(bof.getRi
      case ImpliesOp:
          return ! eval(bof.getLeft()) || eval(bof.getRig
      }
  } else if ( f instanceof NotForm ) {
    return ! eval( ((NotForm) f).getOperand() );
  } else if ( f instanceof ConstantForm ) {
    switch ( (ConstantForm) f ){
      case True:  return true;
      case False: return false;
    }
  } else if ( f instanceof AtomForm ) { … }
}
```

Try to avoid `instanceof`

- Use dynamic binding!
  - add eval as abstract method to Form
  - each concrete class should implement this operation
- Disadvantage: each time you add a new operation your complete data structure has to be adjusted
  - what if this cannot be done?
- Solution: use the *visitor pattern*!



Visitor pattern

Visitor pattern can be used when:

- you need operations on a **(recursive) datastructure**;
- the datastructure is **fixed** beforehand;
  - ie.e no new types of nodes are allowed to be created
- you **don’t want to change** the datastructure for new operations.

Visitor pattern – implementation

```
public interface DataStruct extends Visitable { }
```

```
public interface Visitable {
    void accept( Visitor visitor );
}
```

- Suppose DataStruct has two implementing classes:
  - Case1
  - Case2

```
public interface Visitor {
    void visitCase1 ( Case1 s );
    void visitCase2 ( Case2 s );
}
```

Visitor pattern – Formula (1)

```
public interface Form extends Visitable { }
```

```
public interface Visitable {
    void accept( FormVisitor vis
}
```

```
public interface                 isitor {
    void vi           orm    ( NotForm form );
    vo        tAtomForm  ( AtomForm form );
    ..
}
```

in principle ok, but we do it differently

Visitor pattern – Formula (2)

- Alternatively
  - Leave out the visitable interface
  - use *overloaded* visit methods (one name, different signatures)

```
public interface Form {
    void accept( FormVisitor visitor );
}
```

```
public interface FormVisitor {
    void visit( NotForm form );
    void visit( AtomForm form );
    ...
}
```

Formulas as Trees - Constants

```
public class TrueForm implements Formula {
    @Override
    public void accept( FormVisitor v ) { ...}
}
```

```
public class FalseForm implements Formula {
    @Override
    public void accept( FormVisitor v ) { ...}
}
```

```
public enum Constant implements Formula {
   ...
   private final boolean value;
   ...
}
```

‘Enum pattern’

- You need objects **without mutable attributes**;
- There is a **finite** number of different **cases**;
- These cases are known at compile time

Formulas as Trees – Binary operations I

```
public class AndForm implements Formula  {
    private Formula leftOperand;
    private Formula rightOperand;

    public AndForm(Formula left, Formula right) {
        this.leftOperand  = left;
        this.rightOperand = right;
    }
    ...
}
```

Code will be nearly identical for

- OrForm
- ImpliesForm

Formulas as Trees – Binary operations II

```
public class BinaryOperator implements Formula {
    private BinOp binOp;
    private Formula leftOperand;
    private Formula rightOperand;

    public BinaryOperator(
        BinOp op, Formula left, Formula right
    ) {
        this.binOp = op;
        this.leftOperand  = left;
        this.rightOperand = right;
    }
    ...
}
```

Formulas as Trees – Binary operations III

How to implement BinOp?

- Attributes:
  - Precedence (immutable)
  - String representation (immutable)
- Cases:
  - $\Rightarrow$
  - $\wedge$
  - $\vee$

Use the enum pattern!

Formulas as Trees – Binary operations IV

```
public enum BinOp {
    AndOp      ( "/\\" ),
    OrOp       ( "\\/" ),
    ImpliesOp  ( "=>" ) ;

    private String string;

    private BinOp ( String string ) {
        this.string  = string;
    }
}
```

Formulas – visitor pattern

Any concrete `Form` class should implement `accept`

```
public interface FormVisitor {
    void visit( BinaryOperator form );
    void visit( Not form );
    void visit( Atom form );
    void visit( Constant form );
}
```

```
public class BinaryOperator imple
    private BinOp binOp;
    private Formula leftOperand;
    private Formula rightOperand;

    public BinaryOperator(
         BinOp op, Formula left, Formula right
     ) { … }

     public void accept( FormVisitor v ) {
        v.visit( this );
     }
    ...
}
```

Formulas – visitor pattern

- That was it! Now you can introduce new operations (without changing the Formula class and its implementations)
- How does this work?
- Example: Collecting all atomic propositions

Ex: Collecting atomic propositions

```
public class CollectVarsVisitor implements FormVisitor {

    private final Set<String> collectedVariables = new HashSet<> ();

    public void visit( Constant form ) { }

    public void visit( Atom form ) {
        collectedVariables.add( form.getId() );
    }

    public void visit( Not form ){
        form.getOperand().accept( this );
    }

    public void visit( BinaryOperator form ) {
        form.getLeft().accept( this );
        form.getRight().accept( this );
    }

    public Set<String> getCollectedVariables() {
        return collectedVariables;
    }
}
```

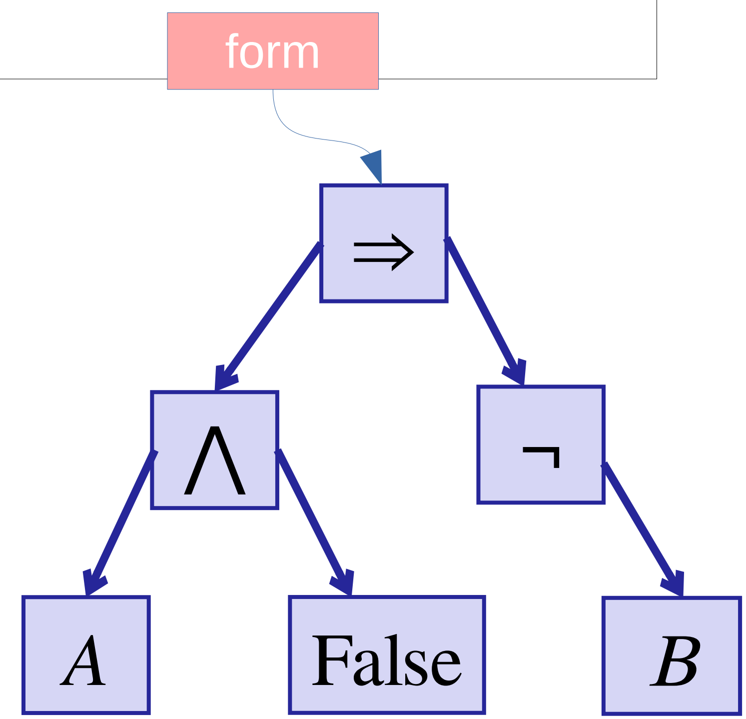
dynamic binding is used to invoke the right method

Ex: Collecting atomic propositions

```
public class CollectVarsVisitor implements FormVisitor { ... }

public Set<String> collectVars(Formula form) {
   CollectVarsVisitor visitor = new CollectVarsVisitor();
   form.accept(visitor);
   return visitor.getCollectedVariables();
}
```

```
form.accept(visitor)
visitor.visit(BinaryOperator form)
 1) form.left.accept(visitor)
   visitor.visit(BinaryOperator form.left)
    1) form.left.left.accept(visitor)
      visitor.visit(Atom form.left.left)
        1) visitor.vars.add(“A”)
    2) form.left.right.accept(visitor)
      visitor.visit(Constant form.left.right)
 2) form.right.accept(visitor)
   visitor.visit(Not form.right)
    1) form.right.operand.accept(visitor)
      visitor.visit(Atom form.right.operand)
        1) visitor.vars.add(“B”)
```



Evaluating Formulas

- Truth assignment: function that maps propositional variables to true or false

```
private Map<String,Boolean> environ;
```

- Evaluation yields a boolean: convenient to have visitors that return booleans: Both interfaces change

```
public interface Form {
    boolean accept( FormVisitor visitor );
}
```

```
public interface FormVisitor {
    boolean visit( BinaryOperator form );
    boolean visit( Not form );
    boolean visit( Atom form );
    boolean visit( Constant form );
}
```

Evaluating Formulas

- Evaluation yields a boolean: convenient to have visitors that return booleans: Both interfaces change

```
public interface Form {
    boolean accept( FormVisitor visitor );
}
```

```
public interface FormVisitor {
    boolean visit( BinaryOperator form );
    boolean visit( Not form );
    boolean visit( Atom form );
    boolean visit( Constant form );
}
```

- **Exercise**: use generics to unify the two FormVisitor interfaces

Evaluating Formulas II

```
public class EvalFormVisitor implements FormVisitor {
    private Map<String,Boolean> environ;

    public EvalFormVisitor ( Map<String,Boolean> environ ) {
        this.environ = environ;
    }
```

- Evaluation of binary operators

```
public boolean visit( BinaryOperator form ) {
    boolean lres = form.getLeft().accept( this );
    boolean rres = form.getRight().accept( this );
    switch( form.getOp() ){
        case AndOp:     return  lres && rres;
        case OrOp:      return  lres || rres;
        case ImpliesOp: return !lres || rres;
    }
}
```

Strategy pattern

Required functionality:

- differs only in the choice of algorithm

Solution:

- abstract the algorithm away, replace it by an attribute

Operators

constant-specific method implementations

```
public enum BinOp implements BinaryOperator<Boolean> {
    AndOp     ( "/\\" , (b1, b2) → b1 && b2),
    OrOp      ( "\\/" , (b1, b2) → b1 || b2),
    ImpliesOp ( "=>"  , (b1, b2) → !b1 || b2);

    public final String string;
    public final BinaryOperator<Boolean> op;

    private BinOp ( String string, BinaryOperator<Boolean> op ) {
        this.string = string;
        this.op = op;
    }

    public Boolean apply(Boolean l, Boolean r) {
        return op.apply(l, r);
    }
}
```

predefined generic interface BinaryOperator<T>

T apply(T y,T y)

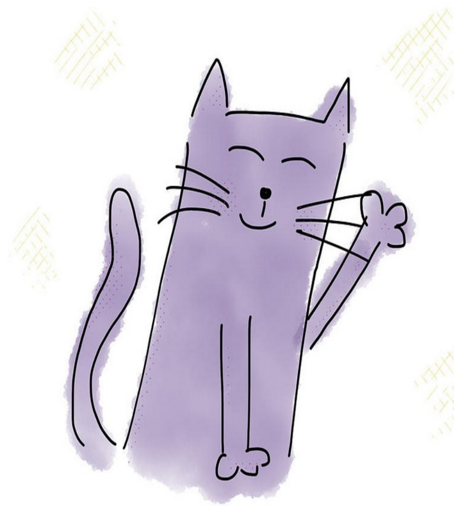
Evaluating Formulas II

```
public class EvalFormVisitor implements FormVisitor {
    private Map<String,Boolean> environ;

    public EvalFormVisitor ( Map<String,Boolean> environ ) {
        this.environ = environ;
    }
```

- Evaluation of binary operators

```
public boolean visit( BinOpForm form ) {
    return form.getOp().apply(
        form.getLeft().accept( this ),
        form.getRight().accept( this )
    );
}
```



Unfortunate naming

- BinaryOperator<T>
  - predefined in java.util.function
  - class of functions sending two T’s to one T
- BinOp
  - enum defined by us
  - defines string representation and precedence of operations
- BinaryOperator
  - class defined by us, implements Formula
  - represents a node in the tree representation of a formula

Finally

- Discuss use of generics?
- Code some Visitors?