Trees, algorithms and logics.
A calculator for Priest's *Introduction to Non-Classical Logics*

Work in progress. v5. Marian Călborean (mc@filos.ro) and Andrei Dobrescu (andrei.dobrescu@neurony.ro).
06.06.2024.

Introduction

Tableau proof system - 1/2

First proposed by Beth. Also inspired by Hintikka.

Reaches the current form in Smmulyan (1968) :

“Our tableaux, unlike those of Beth, use only one tree instead of two. Hintikka's tableau method also uses only one tree, but each point of the tree is a finite set of formulas, whereas in ours, each point consists of a single formula. The resulting combination has many advantages-indeed we venture to say that if this combination had been hit on earlier, the tableau method would by now have achieved the popularity it so richly deserves”

Proof that **A ⊃ B, B ⊃ C ⊢ A ⊃C**.

From Priest (2008)

$$\begin{tikzcd}
 & & A \supset B \arrow[d] & & \\
 & & B \supset C \arrow[d] & & \\
 & & \neg(A \supset C) \arrow[d] & & \\
 & & A \arrow[d] & & \\
 & & \neg C \arrow[dl] \arrow[dr] & & \\
 & \neg A \arrow[dl] \arrow[d] & & B \arrow[d] \arrow[dr] & \\
\neg B & C & & \neg B & C \\
\times & \times & & \times & \times
\end{tikzcd}$$

Tableau proof system - 2/2

Also called “semantic tableaux”, this system is strictly syntactic (see rules) just as axiomatic systems or natural deduction.

It does syntactically a counterexample search. Priest: “The tableau procedure is... a systematic search for an interpretation that makes all the formulas on the initial list true. Given an open branch of a tableau, such an interpretation can …be read off from the branch”

Priest on the advantages: “… constructing tableau proofs, and so ‘getting a feel’ for what is, and what is not, valid in a logic, is very easy (indeed, it is **algorithmic**). Another is that the soundness and, particularly, **completeness proofs for logics are very simple** using tableaux. Since these areas are both ones where inexperienced students experience difficulty, tableaux have great pedagogical attractions.”(emphasis added)

1) $\dfrac{\sim\sim X}{X}$

2) $\dfrac{X \wedge Y}{\begin{array}{c} X \\ Y \end{array}}$ $\quad$ $\dfrac{\sim(X \wedge Y)}{\sim X \mid \sim Y}$

3) $\dfrac{X \vee Y}{X \mid Y}$ $\quad$ $\dfrac{\sim(X \vee Y)}{\begin{array}{c} \sim X \\ \sim Y \end{array}}$

4) $\dfrac{X \supset Y}{\sim X \mid Y}$ $\quad$ $\dfrac{\sim(X \supset Y)}{\begin{array}{c} X \\ \sim Y \end{array}}$

Above, the rules of Smullyan 1968.
Below, a proof from Priest (2008)

$$\begin{tikzcd}
 & & \neg(((A \supset B) \wedge (A \supset C)) \supset (A \supset (B \wedge C))) & & & \\
 & & (A \supset B) \wedge (A \supset C) & & & \\
 & & \neg(A \supset (B \wedge C)) & & & \\
 & & (A \supset B) & & & \\
 & & (A \supset C) & & & \\
 & & A & & & \\
 & & \neg(B \wedge C) \arrow[dl] \arrow[dr] & & & \\
 & \neg B \arrow[dl] \arrow[d] & & \neg C \arrow[d] \arrow[dr] & & \\
\neg A & B & & \neg A & B \arrow[d] \arrow[dr] & \\
\times & \times & & \times & \neg A & C \\
 & & & & \times & \times
\end{tikzcd}$$

Tableaux in INCL - Priest (2008)

Priest's *Introduction to Non-Classical Logics: From If to Is* (2008, Cambridge University Press)

- With few exceptions (e.g. Lukasiewicz' continuum-valued logic $Ł_{\aleph}$), logics in the book have **tableau proof systems** with corresponding exercises.
- For this, Priest introduces **original variations of the tableau method**, which were then discussed in the literature (e.g. Johnson 2015)

Other relevant info:

- Most logics in the book have possible-world semantics
- The **exact counting of logics covered is impossible**, since the book indicates how rules can be combined so as to result in new logics (e.g. accessibility relations in modal logic)
- Roy 2006, 2017 give natural deduction systems for the logics in Priest (2008)

Propositional Logics in INCL - Priest (2008)

#	Chapter	Variation of method	References
1.	*Classical Logic*	1.4. Basic tableau method	Beth, Hintikka, Smullyan
2.	*Basic Modal Logic*	2.4: Worlds, accessibility, □-rule reapplication	Fitting & Mendelsohn
3.	*Normal Modal Logics*	3.3: Graphs with ρστη 3.4 Infinity at η/τ and trial/error alternative, 3.6a. Multimodal <P>/<F> for tense logic K^t 3.6b Extending K^t: η'δφβ, 3.10 other (Euclidian etc)	Girle
4.	*Non-Normal Modal Logics*	4.3 □-inhabited at N + ρστη etc 4.4a S0.5 Lemmon logics	Girle
5.	*Conditional Logics*	TBD	
6.	*Intuitionistic Logic*	TBD	
7.	*Many-valued Logics*	*No tableau method*	
8.	*First Degree Entailment*	TBD	
9.	*Logics with Gaps, Gluts and Worlds*	TBD	
10.	*Relevant Logics*	TBD	
11.	*Fuzzy Logics*	*No tableau method*	Pace Olivetti
11a.	*Ap. Many-valued Modal Logics*	TBD	

Quantified Logics in INCL - Priest (2008)

#	Chapter	Variation of method
12	Classical First-order Logic	Partial (without parser)
13	Free Logics	TBD
14	Constant Domain Modal Logics	TBD
15	Variable Domain Modal Logics	TBD
16	Necessary Identity in Modal Logic	TBD
17	Contingent Identity in Modal Logic	TBD
18	Non-normal Modal Logics	TBD
19	Conditional Logics	TBD
20	Intuitionistic Logic	TBD
21	Many-valued Logics	*No tableau method*
22	First Degree Entailment	TBD
23	Logics with Gaps, Gluts and Worlds	TBD
24	Relevant Logics	TBD
25	Fuzzy Logics	*No tableau method*

Proof $\vdash a = b \supset (\exists xSxa \supset (a = a \wedge \exists xSxb))$. *From Priest (2008)*

$$\begin{tikzcd}
 & \neg(a=b\supset(\exists xSxa\supset(a=a\wedge\exists xSxb))) & \\
 & a=b & \\
 & \neg(\exists xSxa\supset(a=a\wedge\exists xSxb)) & \\
 & \exists xSxa & \\
 & \neg(a=a\wedge\exists xSxb) & \\
 & Sca \arrow[dl] \arrow[dr] & \\
\neg a=a & & \neg\exists xSxb \\
\times & & \forall x\neg Sxb \\
 & & \neg Scb \\
 & & Scb \\
 & & \times
\end{tikzcd}$$

Towards a calculator

Non-logical notions

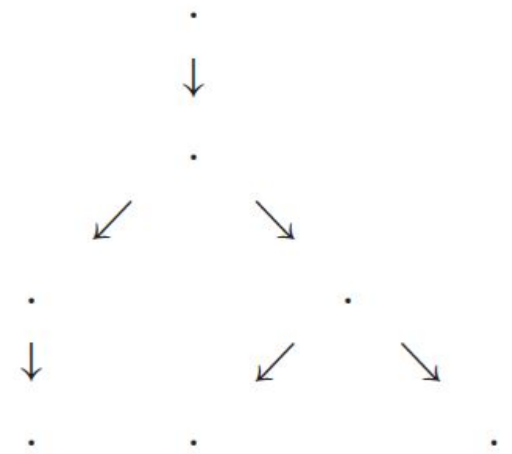
Tree
"… a partial order with a unique maximum element, x_0, such that for any element, x_n, there is a unique finite chain of elements $x_n \leq x_{n-1} \leq \cdot \cdot \cdot \leq x_1 \leq x_0$. " (Priest)

Tree structures:

- **Root node.** In effect there can be more than one: premises and negated conclusion
- **Branch** (finite chain from tip to root). A node can split in two or three etc
- **Tips**. Nodes with no child nodes

***Types of nodes** by information* (our classification):

- **Logical formulas**: this is the standard node: "[P]<F>A"
- **Labels with semantic value**. Examples**:**
  - In modal logics node "irj" hints that world *i accesses world j.*
  - In relevant logic B node "$i" hints that world i is normal
- **Combination of formula and some metadata** of semantic value:
  - In modal logics "A V B, i" hints that A V B holds at world i
  - In intuitionistic logic "A V B, -i" hints that A V B is false at world i



Above, a tree. Below, a proof in tense logic K^t that A ⊢ [P]<F>A. Both from Priest (2008)

$A, 0$

$\neg[P]\langle F\rangle A, 0$

$\langle P\rangle \neg \langle F\rangle A, 0$

$1r0$

$\neg \langle F\rangle A, 1$

$[F]\neg A, 1$

$\neg A, 0$

$\times$

Logical notions

Formulas of the language (**operators** and **predicates** vary with the logic)

- Modal logics may add dyadic ⥽ and monadic □, ◇, [P], <P>, [F], <F>.
- Free logic adds special predicate 𝔈 etc.

Derivation rules *(our classification)*

a) **Formula-based**

- E.g. From A V B, we split the tree for A and for B
- They *consume* nodes (once applied, nodes can be ignored)

b) **Label-based**

- E.g. if the logic is transitive, for any nodes of the form irj and jrk, add irk
- They *consume* nodes but need care when applied (e.g. rule η after others)

c) **Complex: both formula- and label- based**

- E.g. □ is reapplied for any newly accessible world label
- In non-normal *N*, ◇ is applied only at world 0 or if there's a □-formula on the branch
- These *do not consume nodes (in general)* and need care.

The rules for □, ◇, then for extendability, reflexivity, symmetry, transitivity. Finally, an infinite tableau in K_τ proving that ⊬ ¬(◇p ∧ □◇p). All from Priest (2008)

$$\begin{array}{c} \Box A, i \\ irj \\ \downarrow \\ A, j \end{array} \qquad \begin{array}{c} \Diamond A, i \\ \downarrow \\ irj \\ A, j \end{array}$$

$$\begin{array}{c} \eta \\ \\ . \\ \downarrow \\ irj \\ \\ \end{array} \qquad \begin{array}{c} \rho \\ \\ . \\ \downarrow \\ iri \\ \\ \end{array} \qquad \begin{array}{c} \sigma \\ \\ irj \\ \downarrow \\ jri \\ \\ \end{array} \qquad \begin{array}{c} \tau \\ \\ irj \\ jrk \\ \downarrow \\ irk \end{array}$$

$$\begin{array}{c} \neg\neg(\Diamond p \wedge \Box\Diamond p), 0 \\ \Diamond p \wedge \Box\Diamond p, 0 \\ \Diamond p, 0 \\ \Box\Diamond p, 0 \\ 0r1 \\ p, 1 \\ \Diamond p, 1 \\ 1r2 \\ p, 2 \\ 0r2 \\ \Diamond p, 2 \\ 2r3 \\ p, 3 \\ \vdots \end{array}$$

Metalogical Notions – 1/2

Closing and counterexamples

- A branch is (**atomically) closed** iff it contains an (atomic) formula and its negation. Mark it with X.
- A tree is **(atomically) closed** iff all branches are (atomically) closed. If closed, it is atomically closed.
- If a tree does not close, one gets a **counterexample** by assigning values (1 / 0) on an open branch

Note that the tableau method is **sound** and **complete,** but the rules as given have **exponential complexity**.

For our calculator (1):
a) We stop working on a branch at X (any two contradictory atomic or □ / ◇-formulas)

b) We time out after a certain amount of time, to prevent the program hanging.

c) We do not order the rules - in the future we will apply the non-branching rules first.

Metalogical Notions – 2/2

Infinite tree in Kη for ⊬ □p. Corresponding countermodel. Simpler countermodel findable by trial and error. All from Priest (2008)

$$\begin{array}{c} \neg\Box p, 0 \\ \Diamond\neg p, 0 \\ 0r1 \\ \neg p, 1 \\ 1r2 \\ 2r3 \\ \vdots \end{array}$$

Algorithms and decidability

- An algorithm is **fair** if each rule application that could be made eventually is.
- In classical logic and modal logic K, a fair algorithm always terminates (proved).

However, in other modal logics and quantified logics:

- For a valid inference, the tree will close, hence the algo will stop.
- Else it may not stop, running into **an infinite counterexample** (see at right)

$$\begin{tikzcd} & \neg p & & \\ w_0 \arrow[r] & w_1 \arrow[r] & w_2 \arrow[r] & \cdots \end{tikzcd}$$

For our calculator (2):

a) The algorithm is fair (it iterates ordered rules and has only finite premises now).

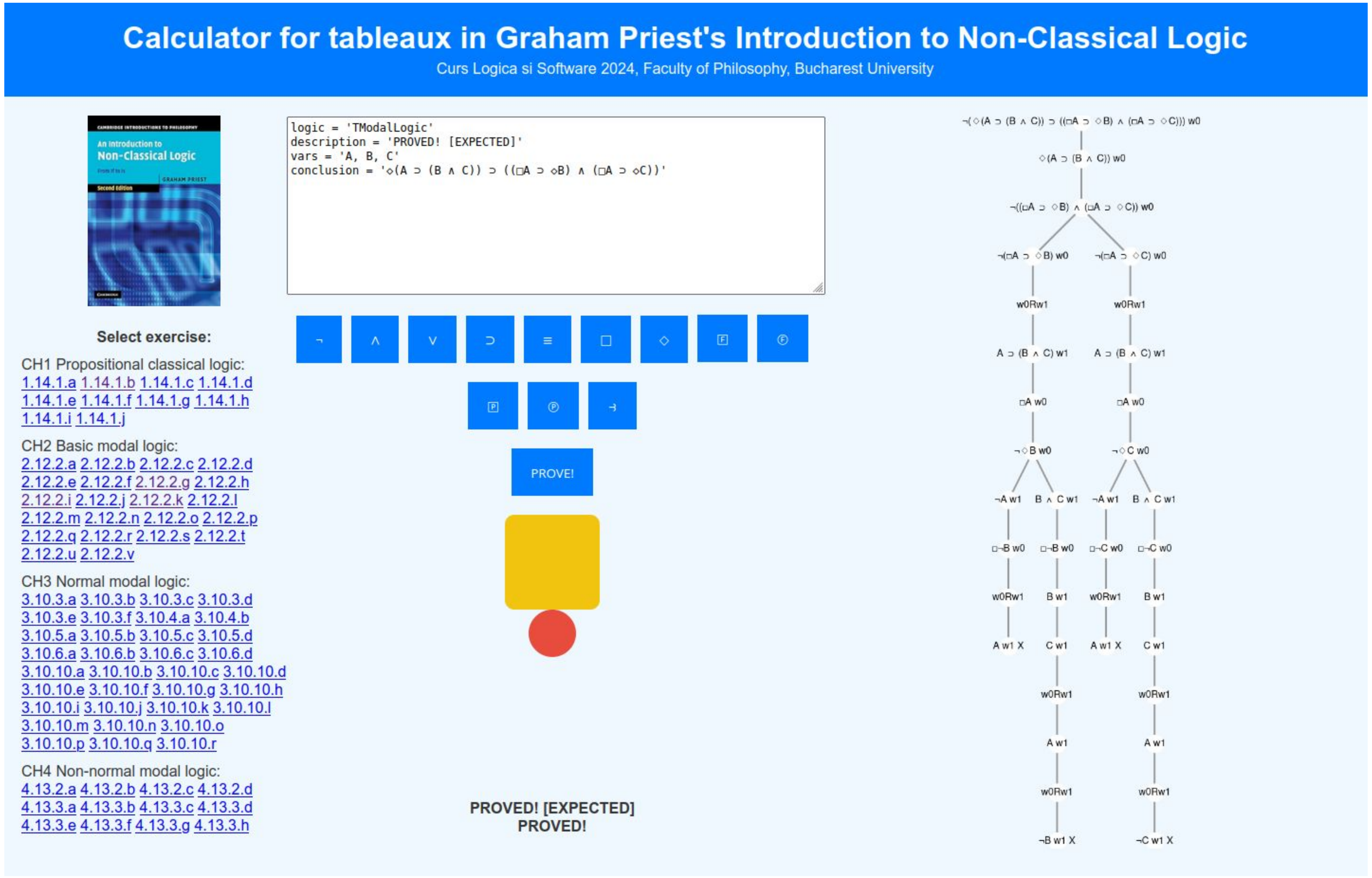
b) We **time out** in both situations: too expensive compute and infinite models.

$$\begin{tikzcd} w_0 \arrow[loop above] \\ \neg p \end{tikzcd}$$

c) In the future we plan:

- Detect infinite tableaux early by pattern (e.g. how many worlds are spawn)
- After time out, have a *trial-and-error* procedure to look for finite countermodels

The calculator



Calculator for tableaux in Graham Priest's Introduction to Non-Classical Logic

Curs Logica si Software 2024, Faculty of Philosophy, Bucharest University

```
logic = 'TModalLogic'
description = 'PROVED! [EXPECTED]'
vars = 'A, B, C'
conclusion = '◇(A ⊃ (B ∧ C)) ⊃ ((□A ⊃ ◇B) ∧ (□A ⊃ ◇C))'
```

Select exercise:

CH1 Propositional classical logic:
1.14.1.a 1.14.1.b 1.14.1.c 1.14.1.d
1.14.1.e 1.14.1.f 1.14.1.g 1.14.1.h
1.14.1.i 1.14.1.j

CH2 Basic modal logic:
2.12.2.a 2.12.2.b 2.12.2.c 2.12.2.d
2.12.2.e 2.12.2.f 2.12.2.g 2.12.2.h
2.12.2.i 2.12.2.j 2.12.2.k 2.12.2.l
2.12.2.m 2.12.2.n 2.12.2.o 2.12.2.p
2.12.2.q 2.12.2.r 2.12.2.s 2.12.2.t
2.12.2.u 2.12.2.v

CH3 Normal modal logic:
3.10.3.a 3.10.3.b 3.10.3.c 3.10.3.d
3.10.3.e 3.10.3.f 3.10.4.a 3.10.4.b
3.10.5.a 3.10.5.b 3.10.5.c 3.10.5.d
3.10.6.a 3.10.6.b 3.10.6.c 3.10.6.d
3.10.10.a 3.10.10.b 3.10.10.c 3.10.10.d
3.10.10.e 3.10.10.f 3.10.10.g 3.10.10.h
3.10.10.i 3.10.10.j 3.10.10.k 3.10.10.l
3.10.10.m 3.10.10.n 3.10.10.o
3.10.10.p 3.10.10.q 3.10.10.r

CH4 Non-normal modal logic:
4.13.2.a 4.13.2.b 4.13.2.c 4.13.2.d
4.13.3.a 4.13.3.b 4.13.3.c 4.13.3.d
4.13.3.e 4.13.3.f 4.13.3.g 4.13.3.h

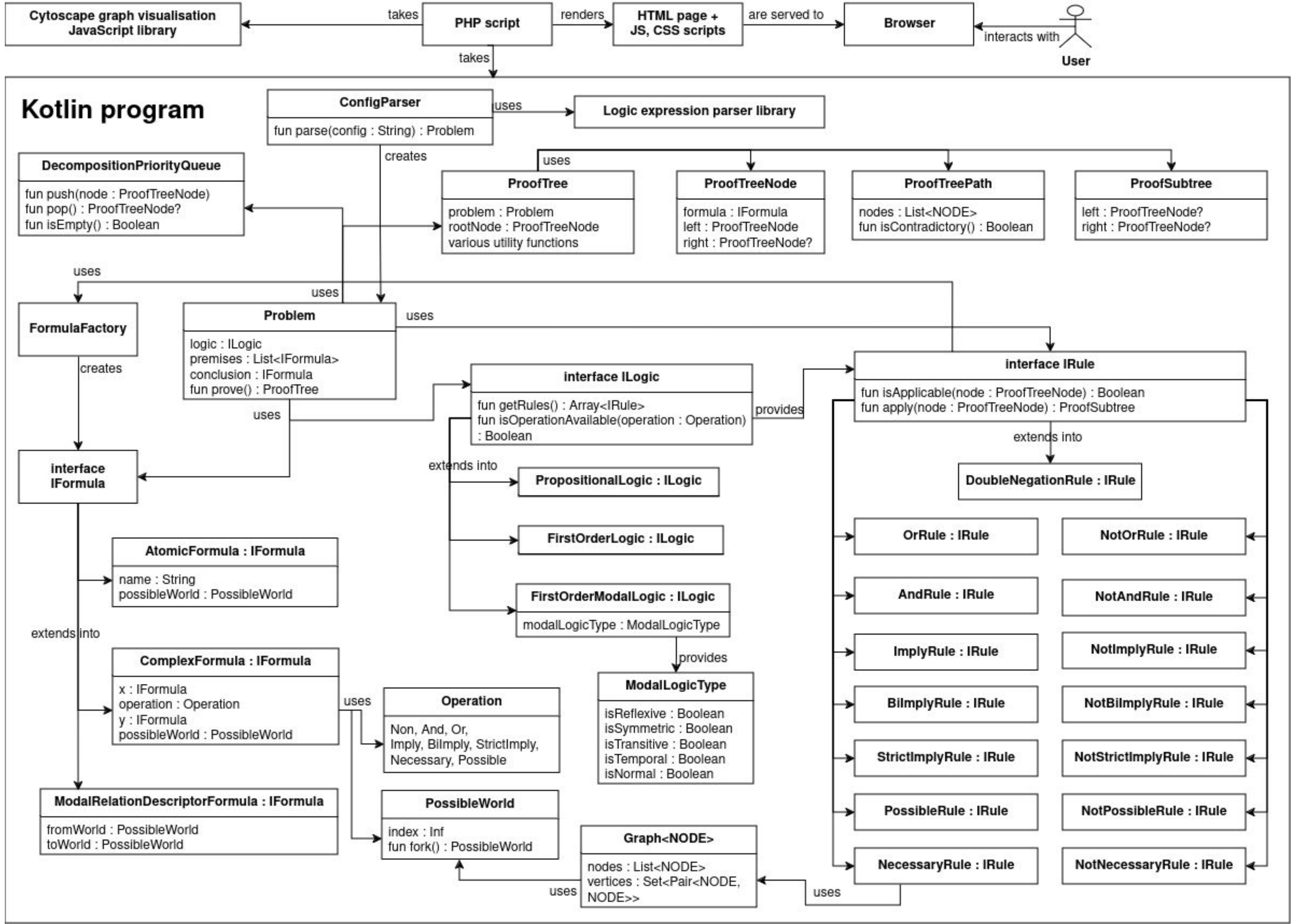
PROVED! [EXPECTED]
PROVED!

Source code: https://github.com/mcfilos/ls2024

Stable version: www.filos.ro/ls/inclcalculator

Development version (Andrei): https://andob.info/logica_si_software_2024/

General schema



Basics

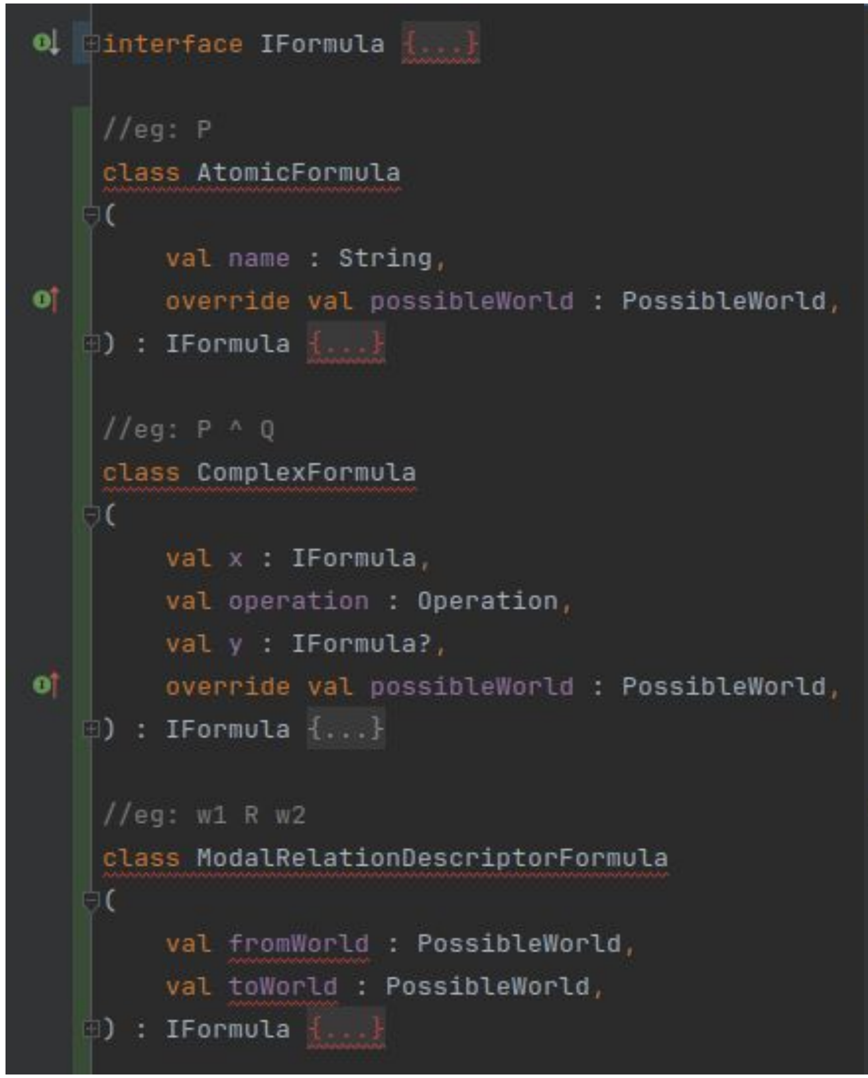
Core written in Kotlin programming language

- Implemented from scratch, using basic collection types (Array, List, Map).
- Hence, easy to translate to other programming languages. It could compile into a binary for JVM execution.
- For public display, it compiles to JavaScript source code (quite readable!)
- It has solved all exercises put to it from Priest (2008) for covered logics.

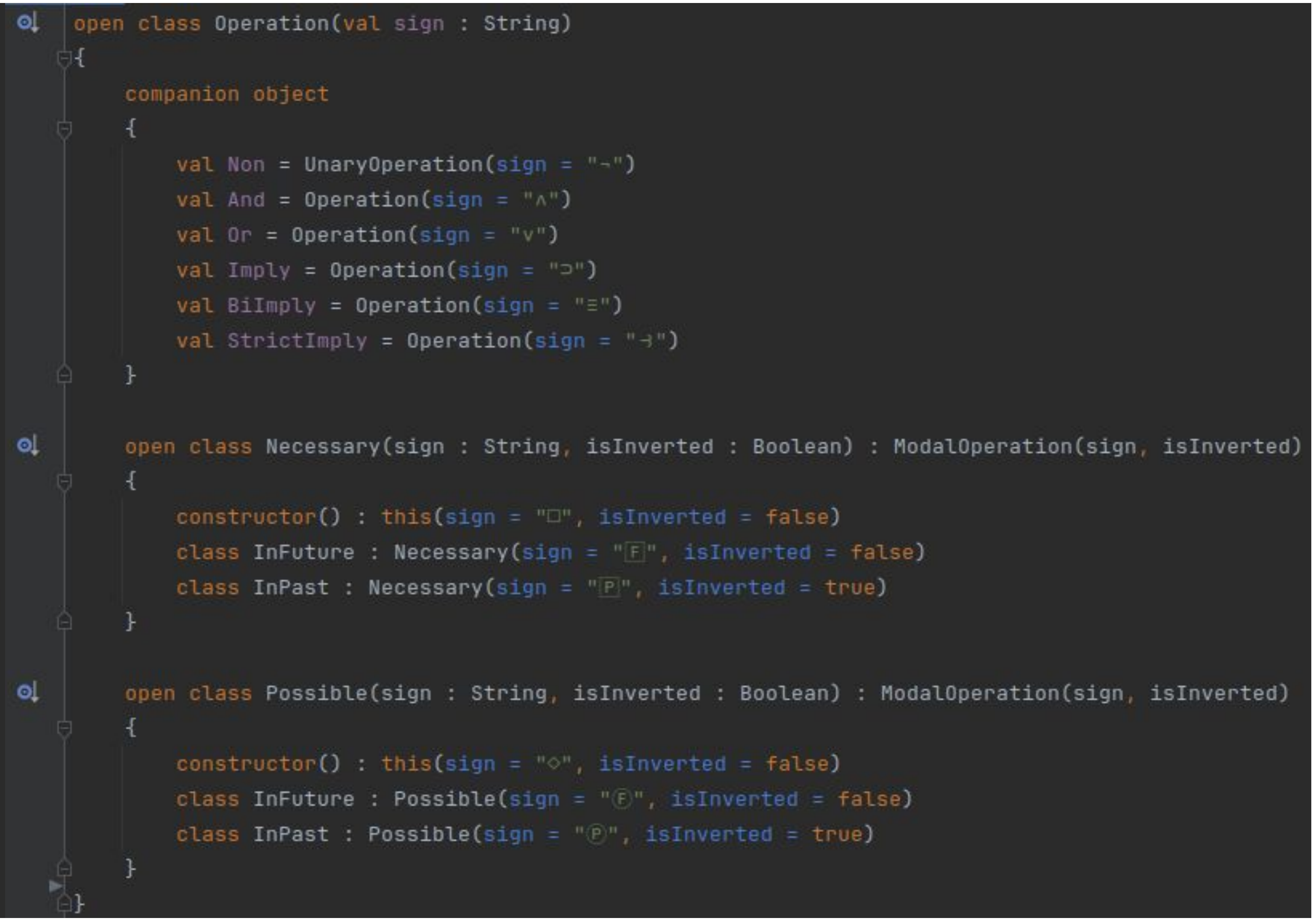
Other dependencies

- *Logik* Kotlin logical expression parser: https://github.com/zimri-leisher/logik
- *Cytoscape* JS library for visual tree rendering: https://js.cytoscape.org/
- A bit of PHP on the server-side, to softcode the demo exercises from the book.

Model class structure: Formula + Operation

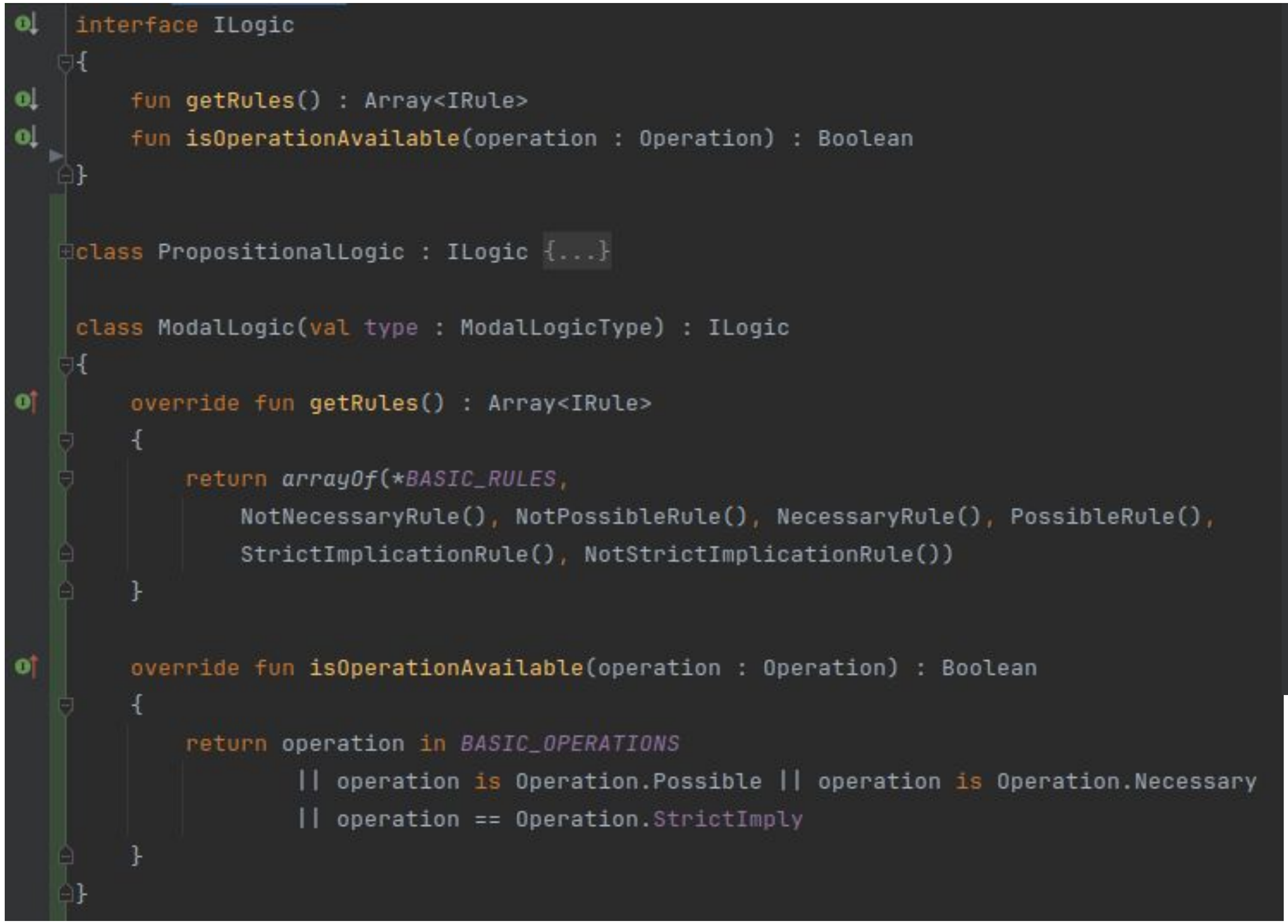


```kotlin
interface IFormula {...}

//eg: P
class AtomicFormula
(
    val name : String,
    override val possibleWorld : PossibleWorld,
) : IFormula {...}

//eg: P ^ Q
class ComplexFormula
(
    val x : IFormula,
    val operation : Operation,
    val y : IFormula?,
    override val possibleWorld : PossibleWorld,
) : IFormula {...}

//eg: w1 R w2
class ModalRelationDescriptorFormula
(
    val fromWorld : PossibleWorld,
    val toWorld : PossibleWorld,
) : IFormula {...}
```

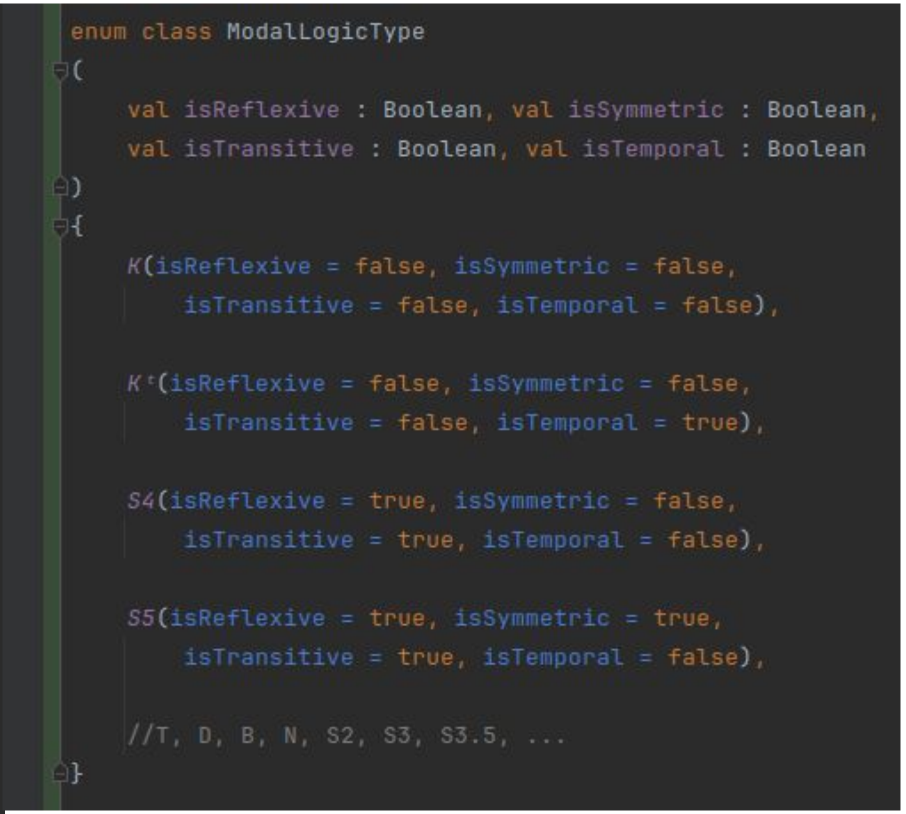


```kotlin
open class Operation(val sign : String)
{
    companion object
    {
        val Non = UnaryOperation(sign = "¬")
        val And = Operation(sign = "∧")
        val Or = Operation(sign = "v")
        val Imply = Operation(sign = "⊃")
        val BiImply = Operation(sign = "≡")
        val StrictImply = Operation(sign = "⥽")
    }

    open class Necessary(sign : String, isInverted : Boolean) : ModalOperation(sign, isInverted)
    {
        constructor() : this(sign = "□", isInverted = false)
        class InFuture : Necessary(sign = "🄵", isInverted = false)
        class InPast : Necessary(sign = "🄿", isInverted = true)
    }

    open class Possible(sign : String, isInverted : Boolean) : ModalOperation(sign, isInverted)
    {
        constructor() : this(sign = "◇", isInverted = false)
        class InFuture : Possible(sign = "Ⓕ", isInverted = false)
        class InPast : Possible(sign = "Ⓟ", isInverted = true)
    }
}
```

Model class structure: Logic



```kotlin
interface ILogic
{
    fun getRules() : Array<IRule>
    fun isOperationAvailable(operation : Operation) : Boolean
}

class PropositionalLogic : ILogic {...}

class ModalLogic(val type : ModalLogicType) : ILogic
{
    override fun getRules() : Array<IRule>
    {
        return arrayOf(*BASIC_RULES,
            NotNecessaryRule(), NotPossibleRule(), NecessaryRule(), PossibleRule(),
            StrictImplicationRule(), NotStrictImplicationRule())
    }

    override fun isOperationAvailable(operation : Operation) : Boolean
    {
        return operation in BASIC_OPERATIONS
                || operation is Operation.Possible || operation is Operation.Necessary
                || operation == Operation.StrictImply
    }
}
```



```kotlin
enum class ModalLogicType
(
    val isReflexive : Boolean, val isSymmetric : Boolean,
    val isTransitive : Boolean, val isTemporal : Boolean
)
{
    K(isReflexive = false, isSymmetric = false,
        isTransitive = false, isTemporal = false),

    Kᵗ(isReflexive = false, isSymmetric = false,
        isTransitive = false, isTemporal = true),

    S4(isReflexive = true, isSymmetric = false,
        isTransitive = true, isTemporal = false),

    S5(isReflexive = true, isSymmetric = true,
        isTransitive = true, isTemporal = false),

    //T, D, B, N, S2, S3, S3.5, ...
}
```

Data structures

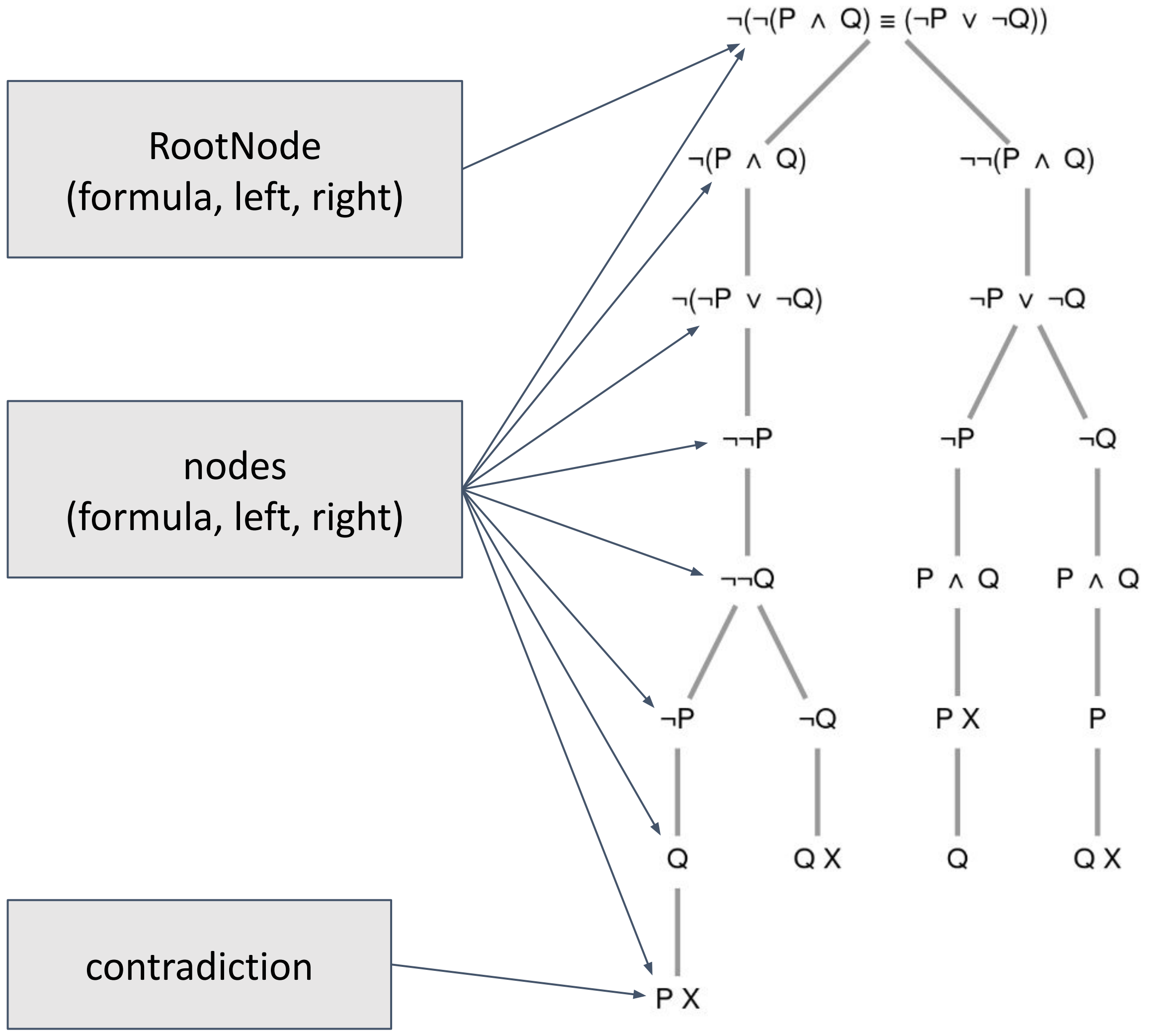
Data structures

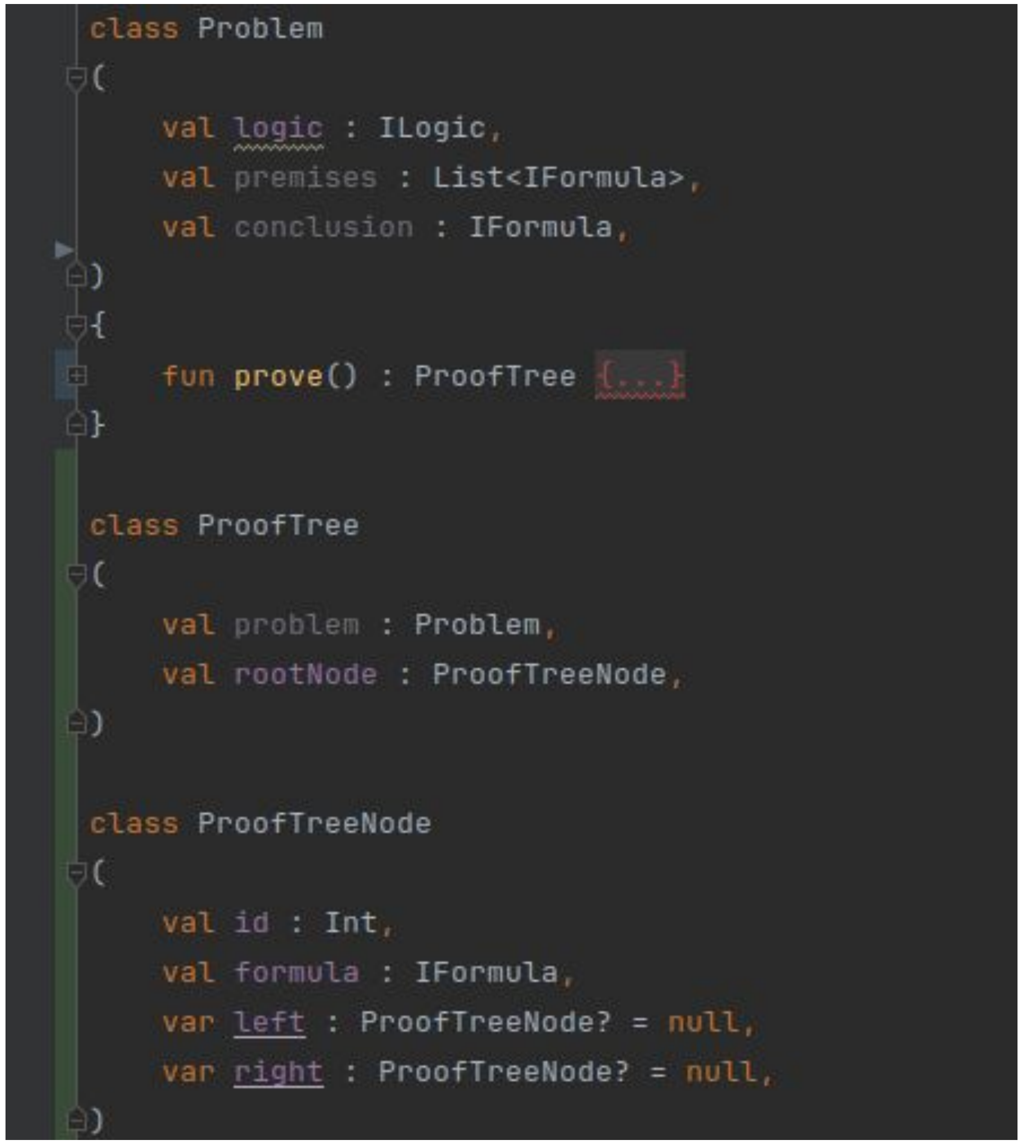
A **binary tree**: *ProofTree*, *ProofTreeNode*, *ProofTreePath* classes
A **priority queue**: the *DecompositionPriorityQueue* class
A **directed graph**: the *Graph<T>* class

The algorithm takes a *Problem* as input and outputs a *ProofTree.*

- The Problem consists of a logic, 0..n premises and a conclusion.
- The program can either prove the problem or disprove the problem.
- **Problem is proved** iff the ProofTree has contradiction on all branches.
- If the problem is disproved, the program provides a **counterexample**.
- The program will **timeout** (neither prove nor disprove) above a limit.

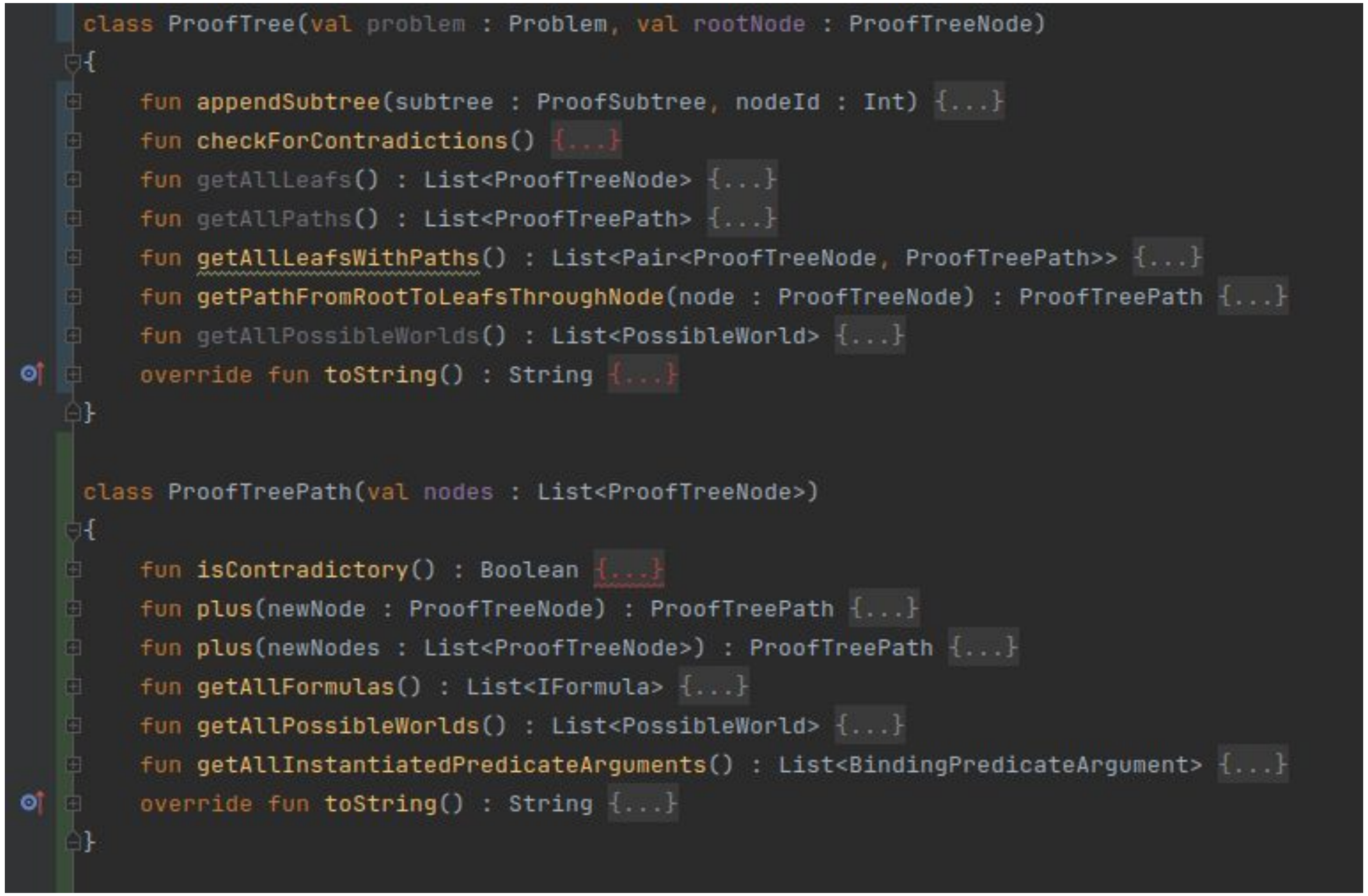
Data structures: ProofTree



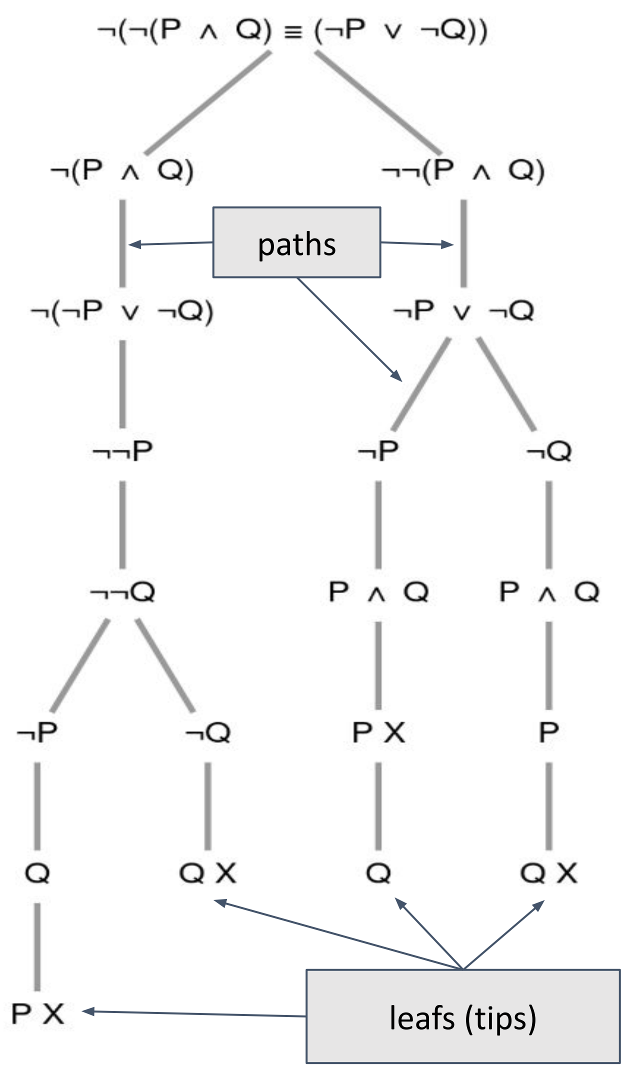


```
class Problem
(
    val logic : ILogic,
    val premises : List<IFormula>,
    val conclusion : IFormula,
)
{
    fun prove() : ProofTree {...}
}

class ProofTree
(
    val problem : Problem,
    val rootNode : ProofTreeNode,
)

class ProofTreeNode
(
    val id : Int,
    val formula : IFormula,
    var left : ProofTreeNode? = null,
    var right : ProofTreeNode? = null,
)
```

Data structures: ProofTree

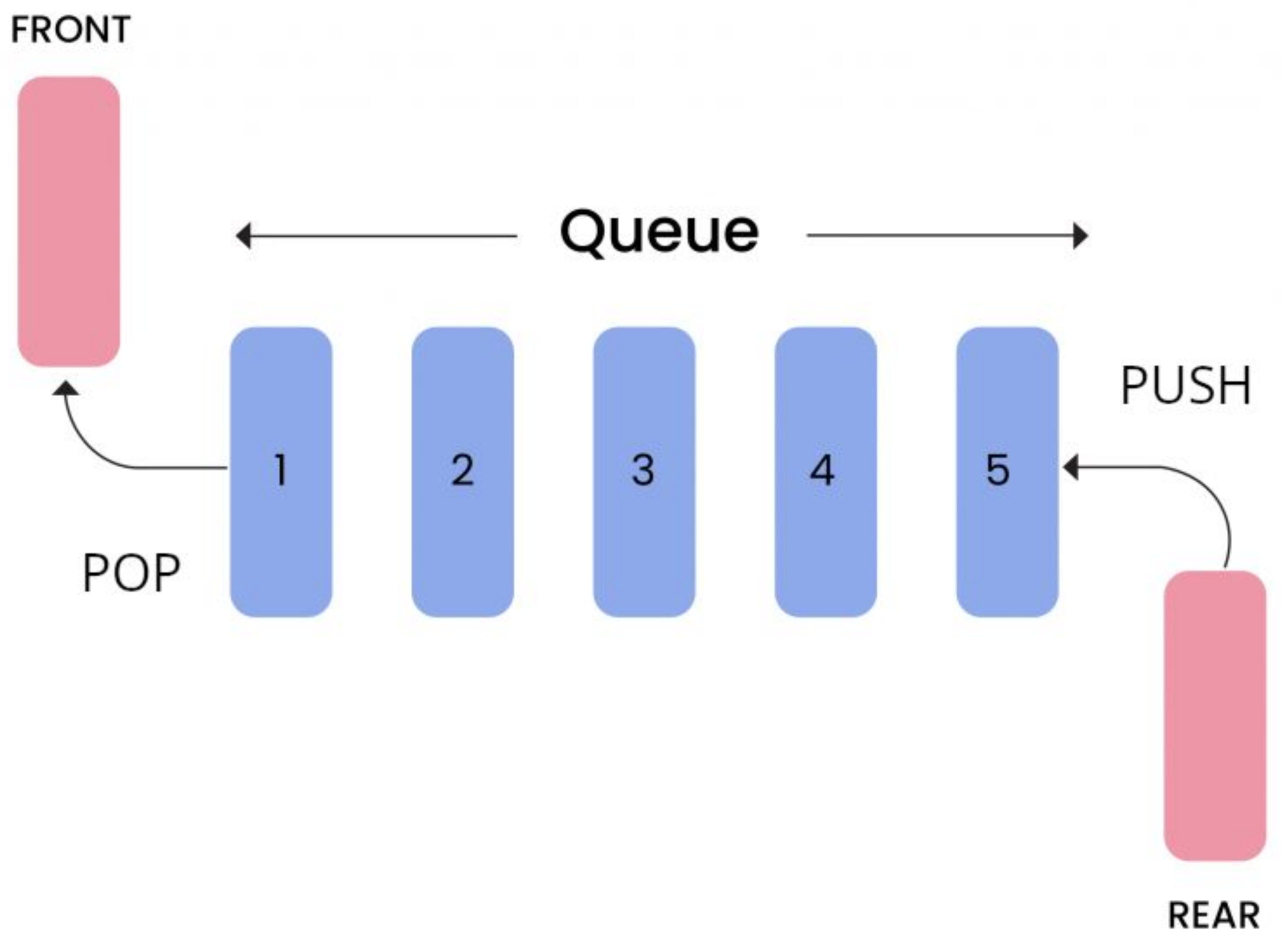


```
class ProofTree(val problem : Problem, val rootNode : ProofTreeNode)
{
    fun appendSubtree(subtree : ProofSubtree, nodeId : Int) {...}
    fun checkForContradictions() {...}
    fun getAllLeafs() : List<ProofTreeNode> {...}
    fun getAllPaths() : List<ProofTreePath> {...}
    fun getAllLeafsWithPaths() : List<Pair<ProofTreeNode, ProofTreePath>> {...}
    fun getPathFromRootToLeafsThroughNode(node : ProofTreeNode) : ProofTreePath {...}
    fun getAllPossibleWorlds() : List<PossibleWorld> {...}
    override fun toString() : String {...}
}

class ProofTreePath(val nodes : List<ProofTreeNode>)
{
    fun isContradictory() : Boolean {...}
    fun plus(newNode : ProofTreeNode) : ProofTreePath {...}
    fun plus(newNodes : List<ProofTreeNode>) : ProofTreePath {...}
    fun getAllFormulas() : List<IFormula> {...}
    fun getAllPossibleWorlds() : List<PossibleWorld> {...}
    fun getAllInstantiatedPredicateArguments() : List<BindingPredicateArgument> {...}
    override fun toString() : String {...}
}
```



Data structures: DecompositionPriorityQueue

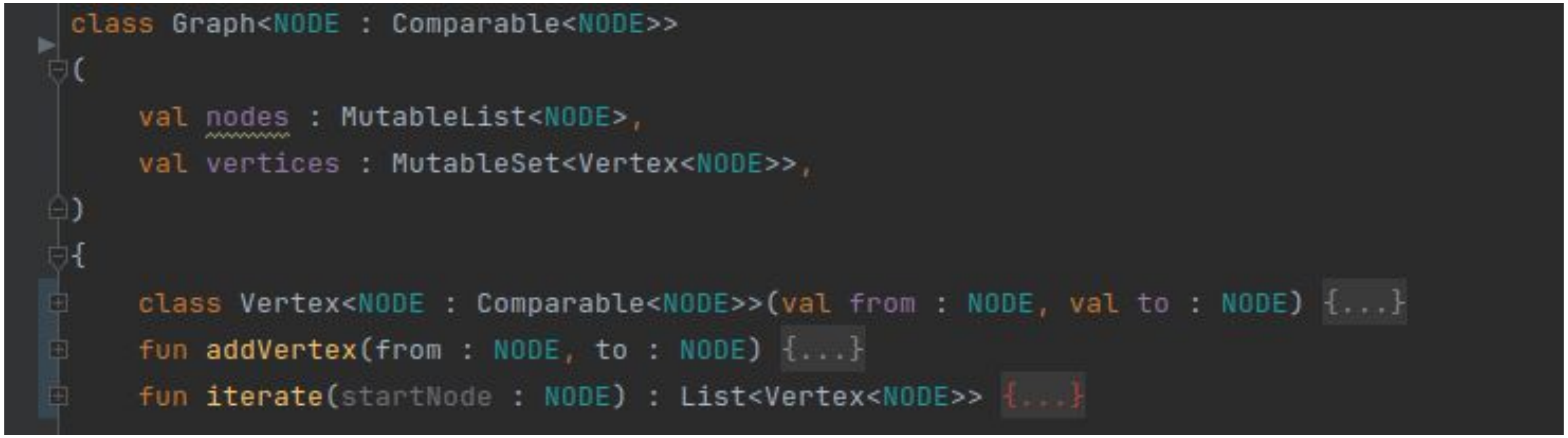
- It is a FIFO (first in first out) data structure.
- On the rear of the queue, we push formulas (starting with the conclusion and premises reversed).
- From the front of the queue, the algorithm will sequentially pop and process formulas.
- The algorithm stops when the queue becomes empty (when there are no formulas left to process).



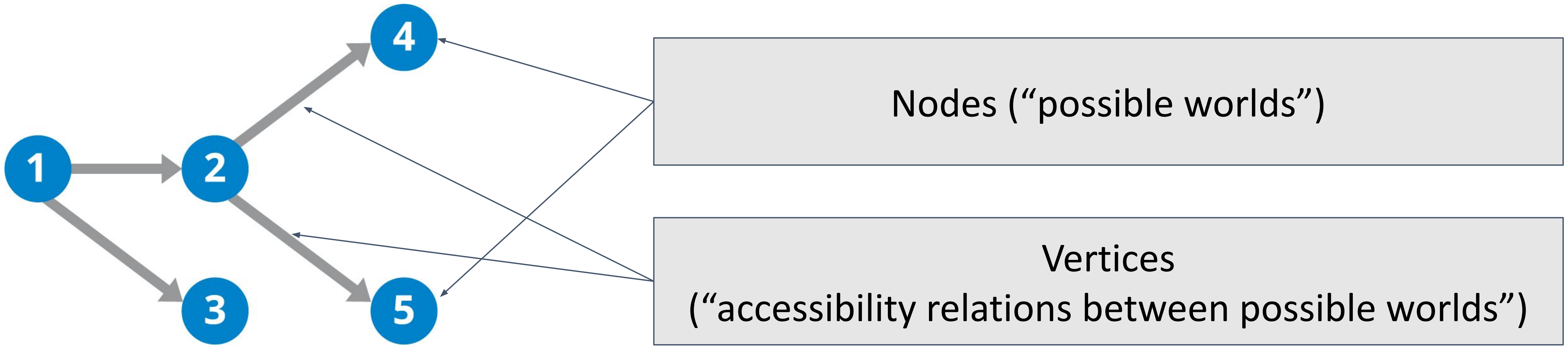
Data structures: DecompositionPriorityQueue

- A priority queue: some formulas have lower priority than other formulas
  - Formulas that start with the □ operator have lower priority than the rest
  - High priority formulas will be popped and processed first
  - □(...) formulas will be popped and processed last.
- The queue also implements a □(...) reuse strategy
  - The queue remembers all the □(...) formulas: even if a □(...) formula is popped and processed, it will be remembered for reusability
  - When the queue gets empty (when there are no formulas left to be processed), all the remembered □(...) formulas will be re-processed.
- The queue implements a □(...) banning strategy
  - To prevent the situation when □(...) gets re-processed again and again.

Data structures: (directed) Graph<T>



```
class Graph<NODE : Comparable<NODE>>
(
    val nodes : MutableList<NODE>,
    val vertices : MutableSet<Vertex<NODE>>,
)
{
    class Vertex<NODE : Comparable<NODE>>(val from : NODE, val to : NODE) {...}
    fun addVertex(from : NODE, to : NODE) {...}
    fun iterate(startNode : NODE) : List<Vertex<NODE>> {...}
```


1
2
3
4
5
Nodes ("possible worlds")
Vertices
("accessibility relations between possible worlds")

Showcase

Classical logic rules

$$\begin{tikzcd} \neg\neg P \arrow[d, no head] \\ P \end{tikzcd}$$

$$\begin{tikzcd} & P \vee Q \arrow[dl, no head] \arrow[dr, no head] & \\ P & & Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \vee Q) \arrow[d, no head] \\ \neg P \arrow[d, no head] \\ \neg Q \end{tikzcd}$$

$$\begin{tikzcd} P \wedge Q \arrow[d, no head] \\ P \arrow[d, no head] \\ Q \end{tikzcd}$$

$$\begin{tikzcd} \neg(P \supset Q) \arrow[d, no head] \\ P \arrow[d, no head] \\ \neg Q \end{tikzcd}$$

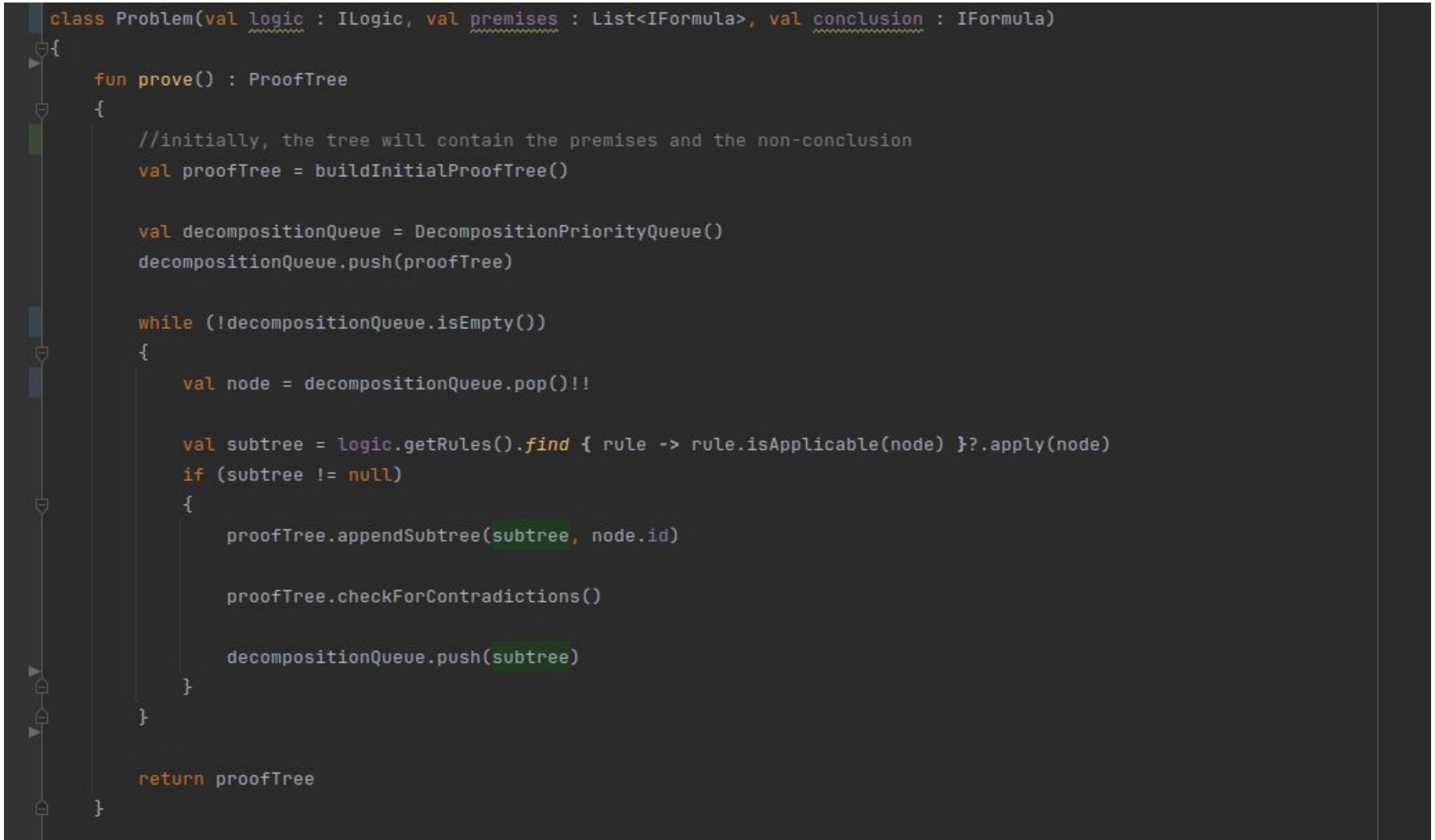
$$\begin{tikzcd} & \neg(P \equiv Q) \arrow[dl, no head] \arrow[dr, no head] & \\ P \arrow[d, no head] & & \neg P \arrow[d, no head] \\ \neg Q & & Q \end{tikzcd}$$

$$\begin{tikzcd} & P \equiv Q \arrow[dl, no head] \arrow[dr, no head] & \\ P \arrow[d, no head] & & \neg P \arrow[d, no head] \\ Q & & \neg Q \end{tikzcd}$$

$$\begin{tikzcd} & P \supset Q \arrow[dl, no head] \arrow[dr, no head] & \\ \neg P & & Q \end{tikzcd}$$

$$\begin{tikzcd} & \neg(P \wedge Q) \arrow[dl, no head] \arrow[dr, no head] & \\ \neg P & & \neg Q \end{tikzcd}$$

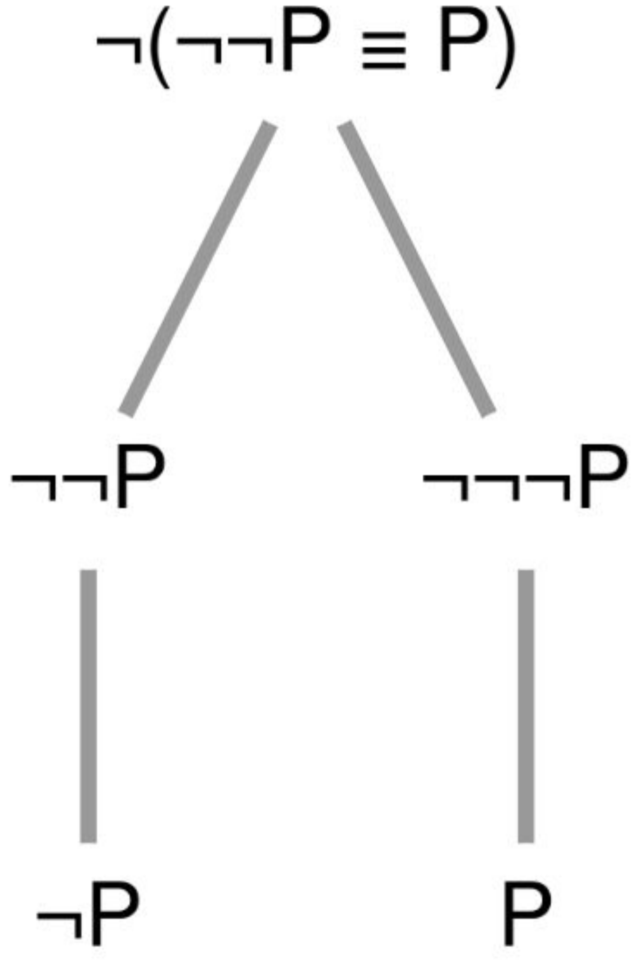
The algorithm in a nutshell - 1/4



```
class Problem(val logic : ILogic, val premises : List<IFormula>, val conclusion : IFormula)
{
    fun prove() : ProofTree
    {
        //initially, the tree will contain the premises and the non-conclusion
        val proofTree = buildInitialProofTree()

        val decompositionQueue = DecompositionPriorityQueue()
        decompositionQueue.push(proofTree)

        while (!decompositionQueue.isEmpty())
        {
            val node = decompositionQueue.pop()!!

            val subtree = logic.getRules().find { rule -> rule.isApplicable(node) }?.apply(node)
            if (subtree != null)
            {
                proofTree.appendSubtree(subtree, node.id)

                proofTree.checkForContradictions()

                decompositionQueue.push(subtree)
            }
        }

        return proofTree
    }
```

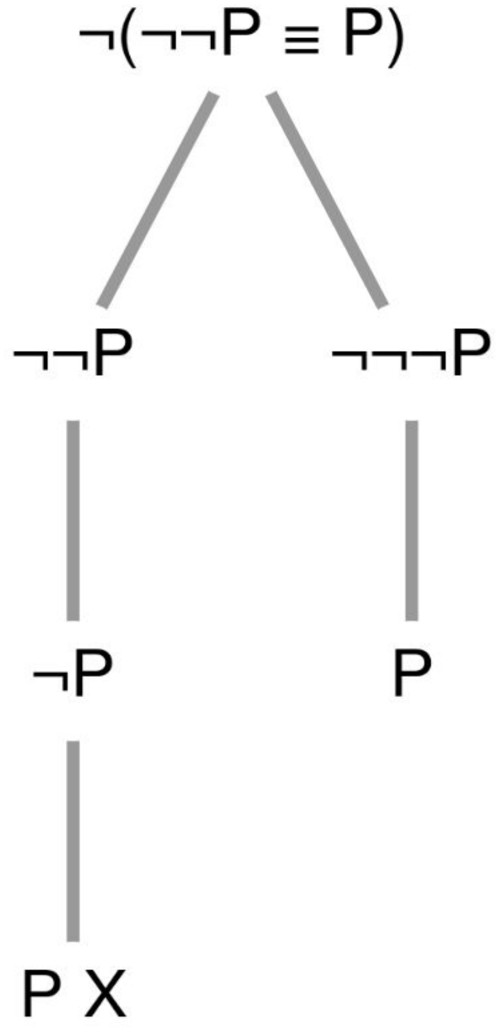
The algorithm in a nutshell - 2/4

- For instance, let’s prove that ¬¬P≡P
- Initialization:
    - proofTree = { rootNode: ¬(¬¬P≡P) }
    - queue = { ¬(¬¬P≡P) }
- First iteration: since queue is not empty:
    - The ¬(¬¬P≡P) formula is popped from the queue
    - The queue becomes empty
    - The ¬≡ rule (NotBiImplyRule) is applied
    - The result subtree is appended to the tree
    - Check for contradictions: there are no contradictions
    - All resulting formulas are pushed to the queue
    - The queue becomes { ¬¬P, ¬P, ¬¬¬P, P }



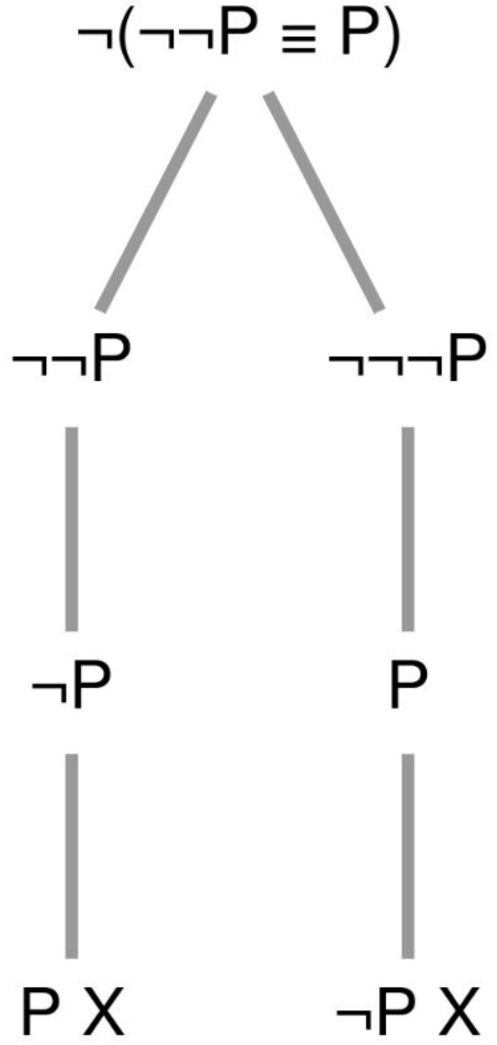
The algorithm in a nutshell - 3/4

- Second iteration
  - The ¬¬P formula is popped from the queue. The queue becomes { ¬P, ¬¬¬P, P }
  - The double negation rule is applied
  - Check for contradictions: found a contradiction
  - Add P to queue. The queue becomes { ¬P, ¬¬¬P, P, P }
- Third iteration
  - The ¬P formula is popped from the queue. The queue becomes { ¬¬¬P, P, P }
  - There is no rule for ¬P. Skip this iteration.



The algorithm in a nutshell - 4/4

- Fourth iteration
  - The ¬¬¬P formula is popped from the queue. Queue: { P, P }
  - The double negation rule is applied
  - Check for contradictions: found another contradiction
  - Add ¬P to queue. The queue becomes { P, P, ¬P }
- 5th / 6th / 7th iteration
  - Pop the queue: { P, P, ¬P }. There are no rules to apply. Skip.
- The queue is empty now. Nothing left to do.
- We have contradiction on all branches
  - Initially we assumed ¬(¬¬P≡P)
  - This can't be right, thus ¬¬P≡P is true.
  - ¬¬P≡P was proved! ▌



Modal logics

Modal logic rules

$$\begin{tikzcd}[row sep=small] \Box A, i \\ irj \arrow[d] \\ A, j \end{tikzcd} \qquad \begin{tikzcd}[row sep=small] \Diamond A, i \arrow[d] \\ irj \\ A, j \end{tikzcd}$$

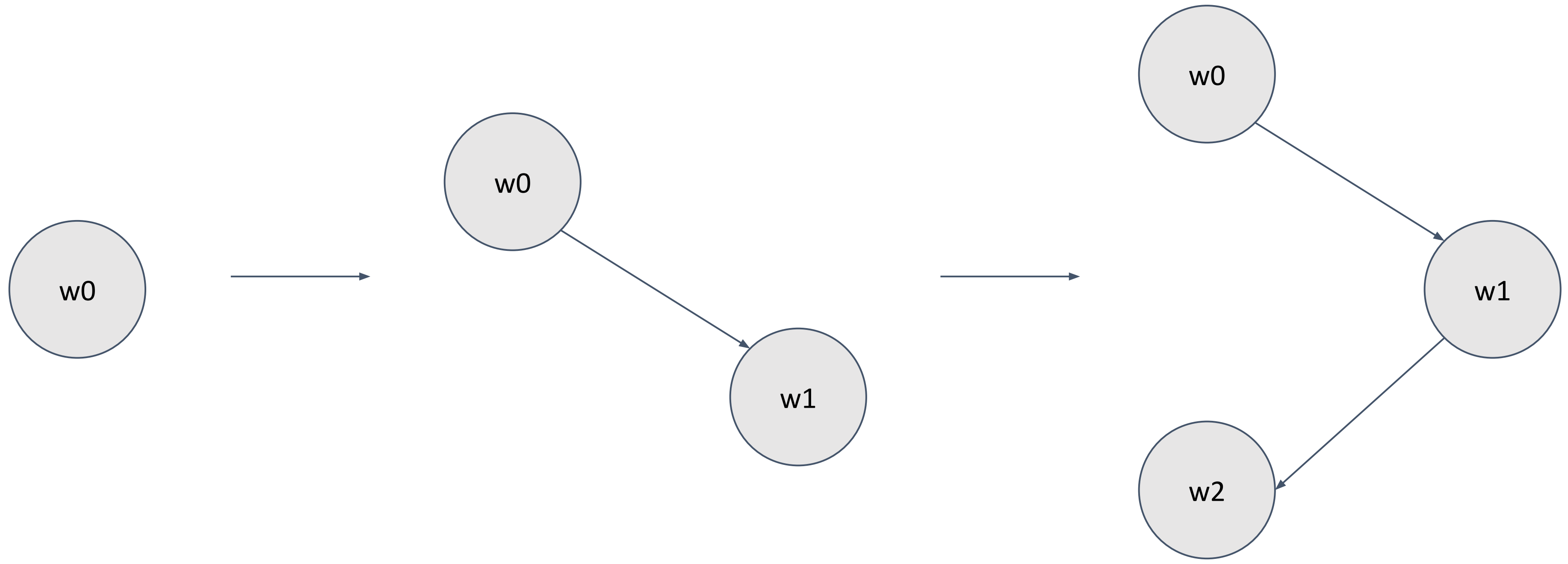
$$\begin{tikzcd} \neg \Diamond P \arrow[d, dash] \\ \Box \neg P \end{tikzcd} \quad \begin{tikzcd} \neg \Box P \arrow[d, dash] \\ \Diamond \neg P \end{tikzcd} \quad \begin{tikzcd} P ⥽ Q \arrow[d, dash] \\ \Box (P \supset Q) \end{tikzcd} \quad \begin{tikzcd} \neg (P ⥽ Q) \arrow[d, dash] \\ \neg \Box (P \supset Q) \end{tikzcd}$$

Modal logic rules: ◇ possible rule

$$\begin{tikzcd}
\Diamond \Diamond P\ w0 \arrow[d, no head] \\
w0Rw1 \arrow[d, no head] \\
\Diamond P\ w1 \arrow[d, no head] \\
w1Rw2 \arrow[d, no head] \\
P\ w2
\end{tikzcd}$$

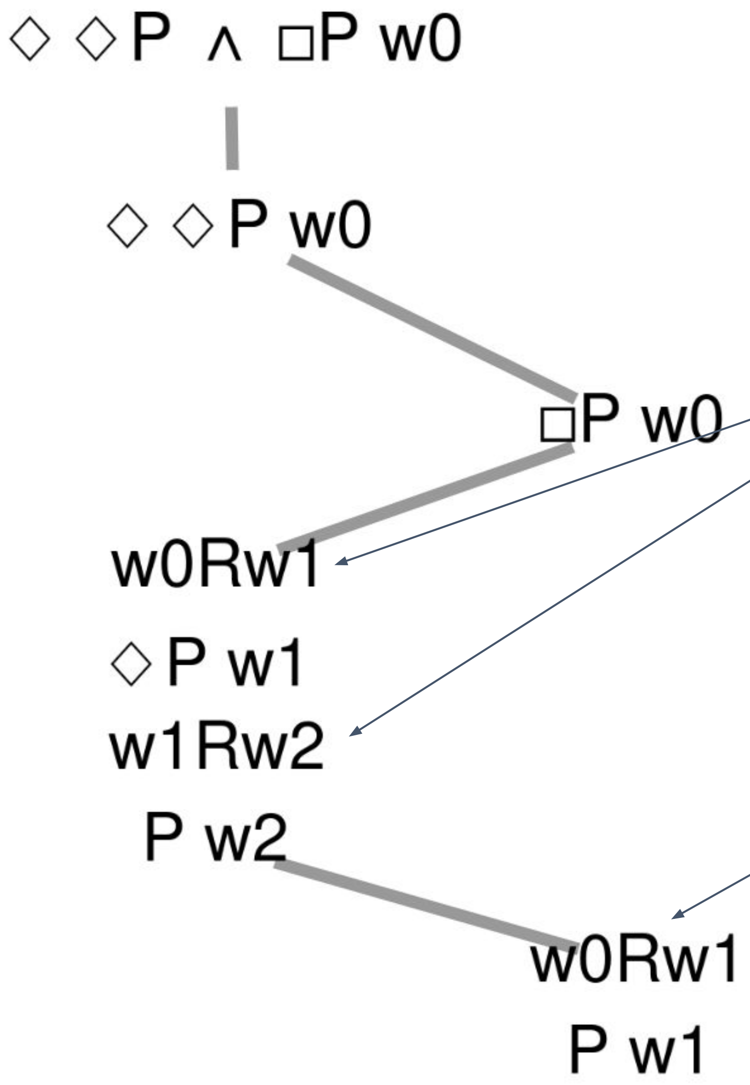
◇ rule will spawn a new world from the current world

(◇ will "build" the graph)



Modal logic rules: □ necessary rule

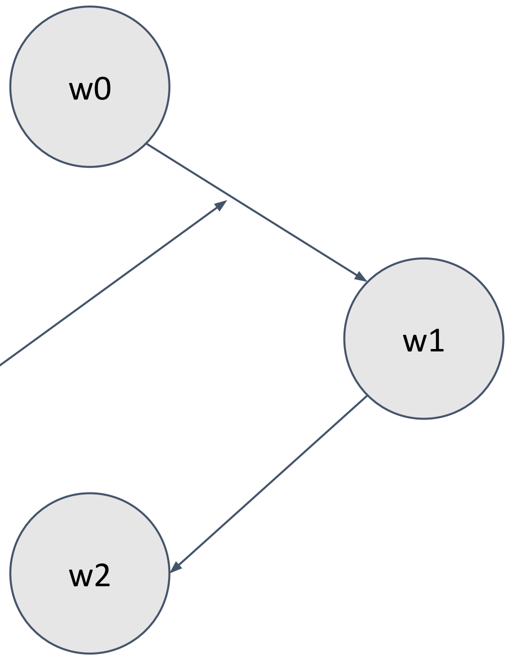
□ rule will iterate all graph vertices that start from the current world



The ◇ rule will build the graph

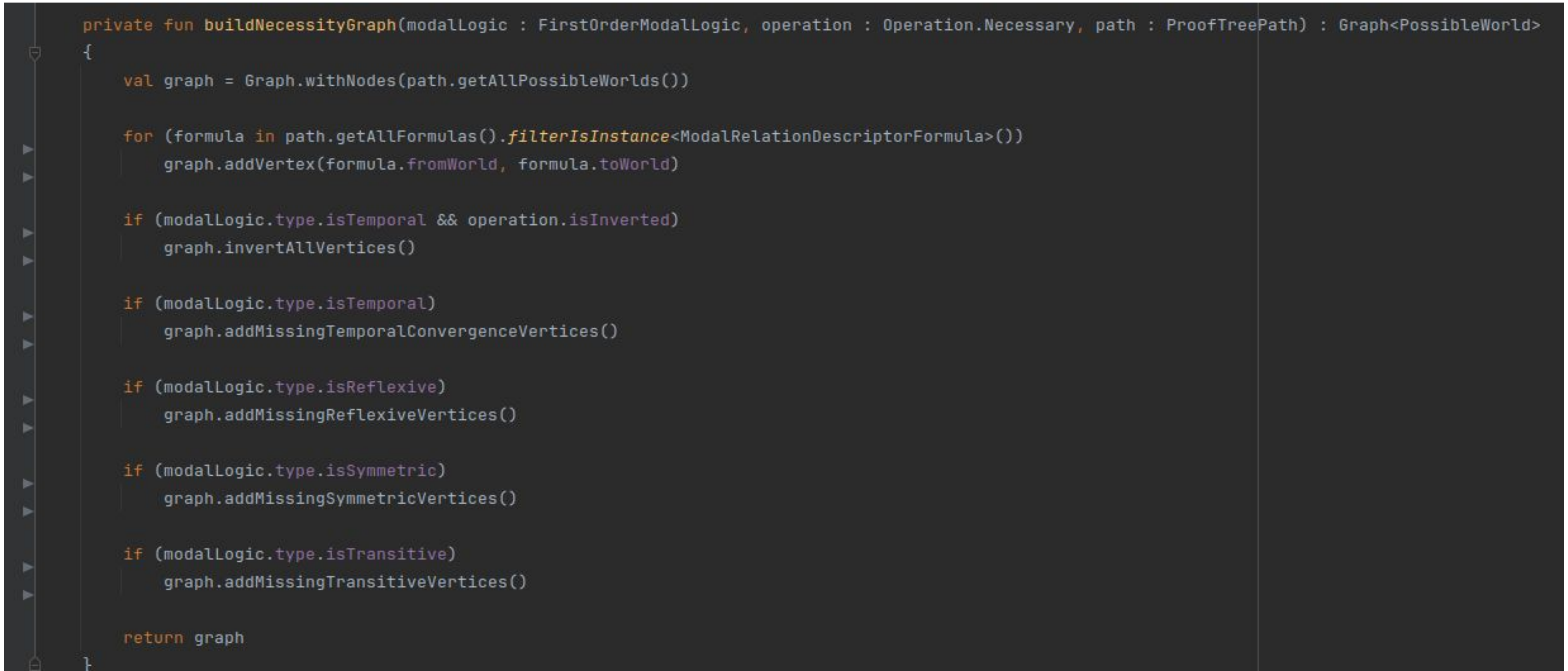
□P w0 is applied here

The graph contains a w0⟶w1 vertex. The □ rule will add: w0Rw1, P w1 to the tree.



Modal logic rules: □ necessary rule

Before applying the □ rule, the graph is populated with the missing graph vertices. Different modal logics requires specific graph types.



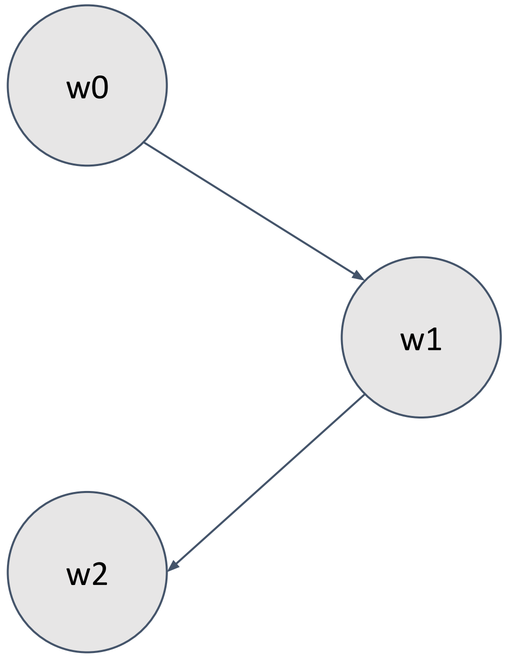
```
private fun buildNecessityGraph(modalLogic : FirstOrderModalLogic, operation : Operation.Necessary, path : ProofTreePath) : Graph<PossibleWorld>
{
    val graph = Graph.withNodes(path.getAllPossibleWorlds())

    for (formula in path.getAllFormulas().filterIsInstance<ModalRelationDescriptorFormula>())
        graph.addVertex(formula.fromWorld, formula.toWorld)

    if (modalLogic.type.isTemporal && operation.isInverted)
        graph.invertAllVertices()

    if (modalLogic.type.isTemporal)
        graph.addMissingTemporalConvergenceVertices()

    if (modalLogic.type.isReflexive)
        graph.addMissingReflexiveVertices()

    if (modalLogic.type.isSymmetric)
        graph.addMissingSymmetricVertices()

    if (modalLogic.type.isTransitive)
        graph.addMissingTransitiveVertices()

    return graph
}
```

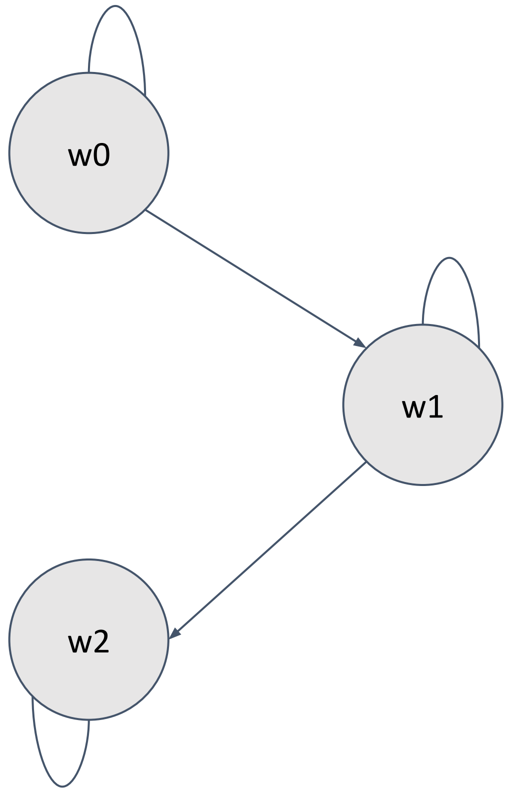
Modal logic rules: □ necessary rule

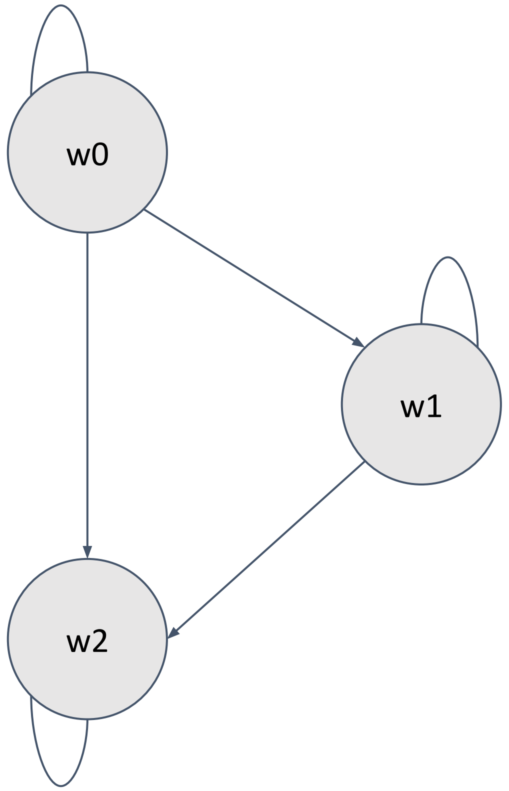
- K modal logic requires no changes to the graph.
- T(Kρ) modal logic requires a reflexive graph.
  - The algorithm adds the missing reflexive vertices as follows: for all nodes w_i, a $w_i \longrightarrow w_i$ vertex will be added to the graph, if missing.
- S4(Kρт) modal logic requires a reflexive and transitive graph.
  - The algorithm adds the missing transitive vertices as follows: for all nodes w_i, $w_\square$, $w_\square$, if the graph contains $w_i \longrightarrow w_\square$ and $w_\square \longrightarrow w_\square$ vertices, and the graph does not contain a $w_i \longrightarrow w_\square$ vertex, then a $w_i \longrightarrow w_\square$ vertex will be added to the graph.
- S5(Kυ) modal logic requires a reflexive, symmetric and transitive graph.
  - The algorithm adds the missing symmetric vertices as follows: for all nodes w_i, $w_\square$, if the graph contains a $w_i \longrightarrow w_\square$ vertex and the graph does not contain a $w_\square \longrightarrow w_i$ vertex, then a $w_\square \longrightarrow w_i$ vertex will be added to the graph.

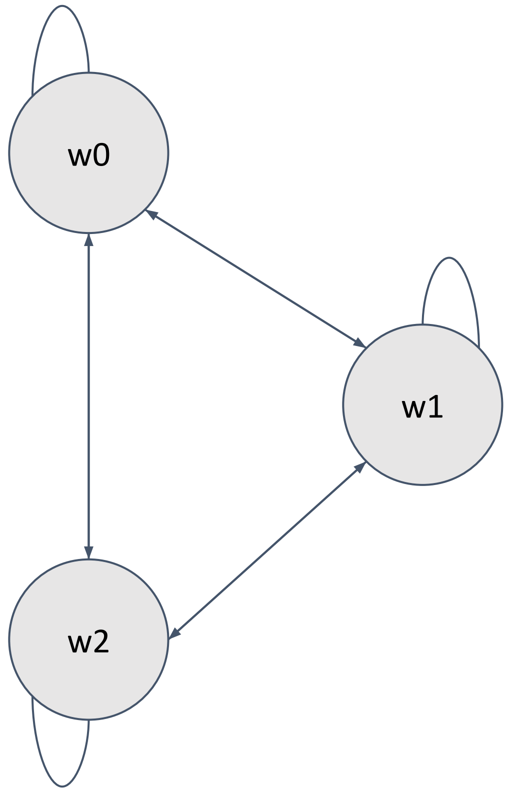
Modal logic rules: □ necessary rule

K	T(Kρ) (reflexive)	S4(Kρτ) (r+transitive)	S5(Kυ) (r+t+symmetric)



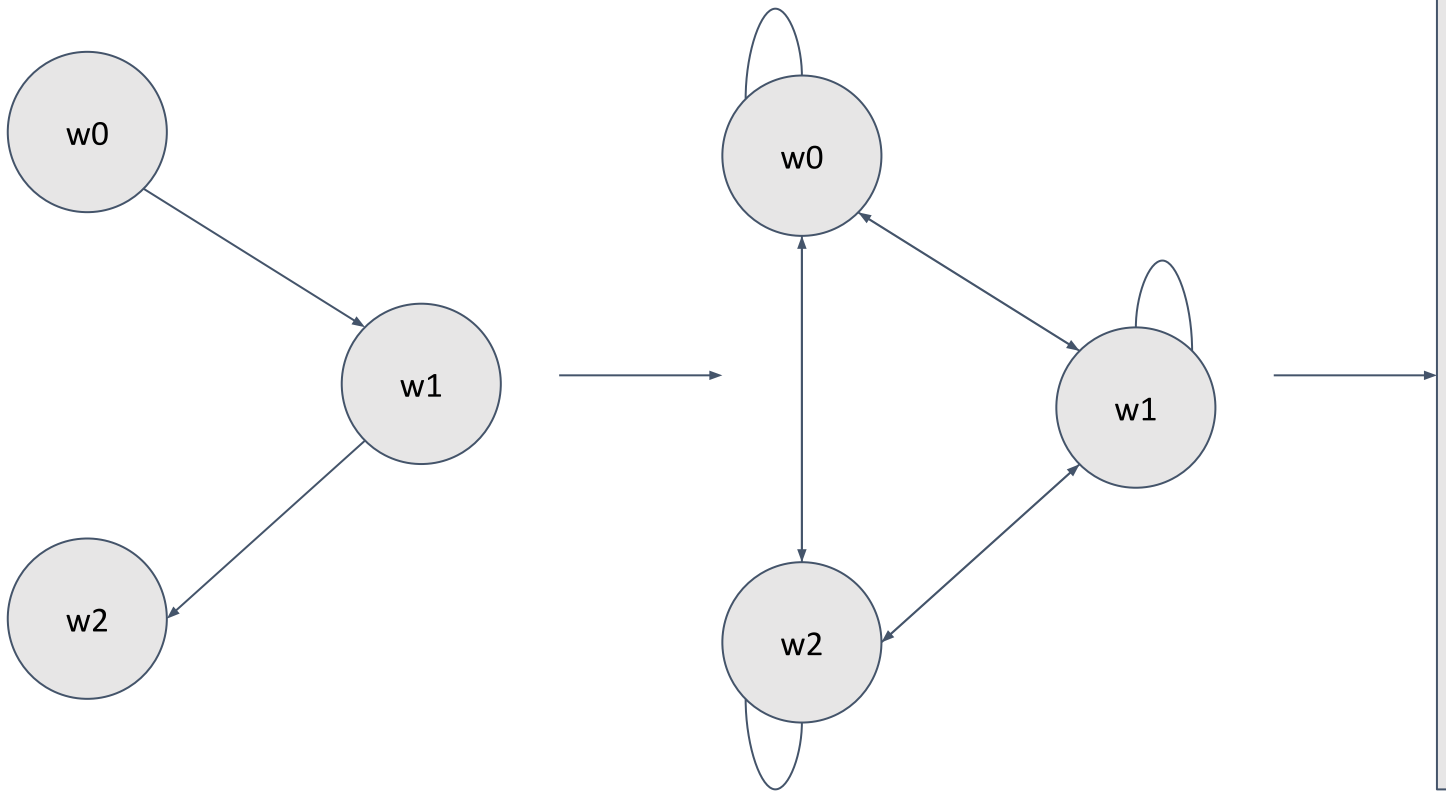






Modal logic rules: □ necessary rule

For instance, on S5(Kᴜ) modal logic, □P w0 will be applied as follows:



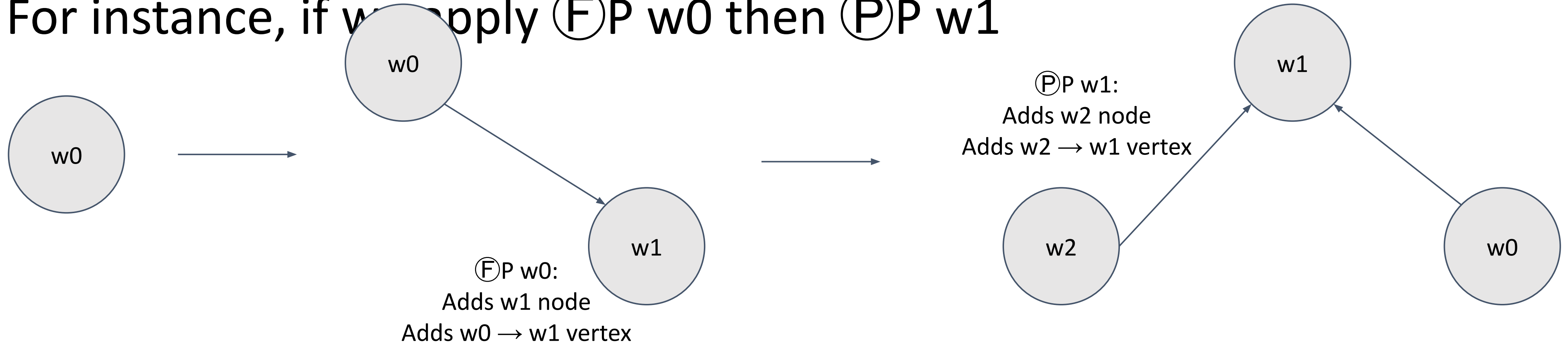
There are 3 vertices that start with w0:
w0 ⟶ w0, w0 ⟶ w1, w0 ⟶ w2.

□P is true at w0, thus P
should be true at w0, w1, w2.

The algorithm will append the following nodes
to the tree:
w0Rw0
P w0
w0Rw1
P w1
w0Rw2
P w2

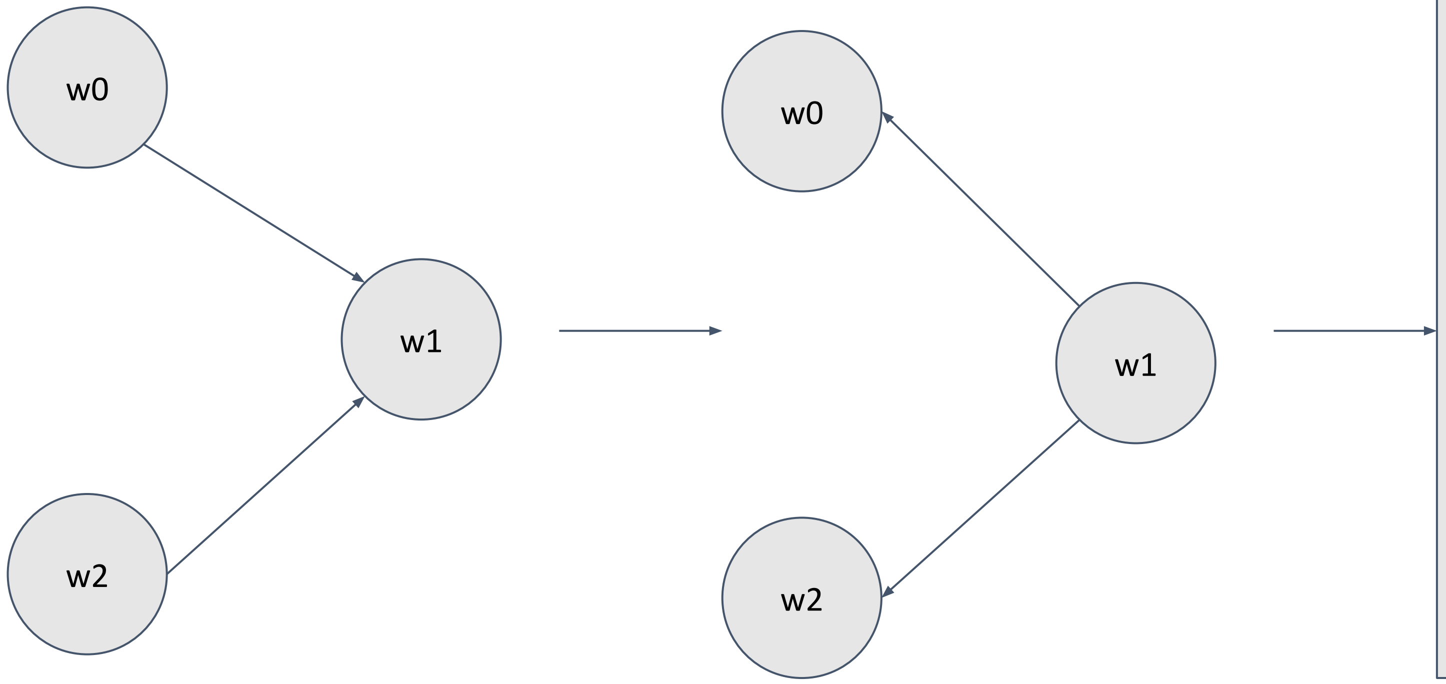
K^t modal logic (tense logic) rules

- On K tense modal logic, there are four different modal operators instead of two:
  - Ⓕ possible in future: will act the same as ◇, will add a w□ node and a w_i → w□ vertex
  - Ⓟ possible in past: will add a w□ node and a w□ → w_i vertex
  - 🄵 necessary in future: will act the same as □
  - 🄿 necessary in past: will reverse graph vertices before applying the □ rule
- For instance, if we apply ⒻP w0 then ⓅP w1



K^t modal logic rules

Then, if we apply $\boxed{P}$P w1, vertices will be reversed before the iteration:



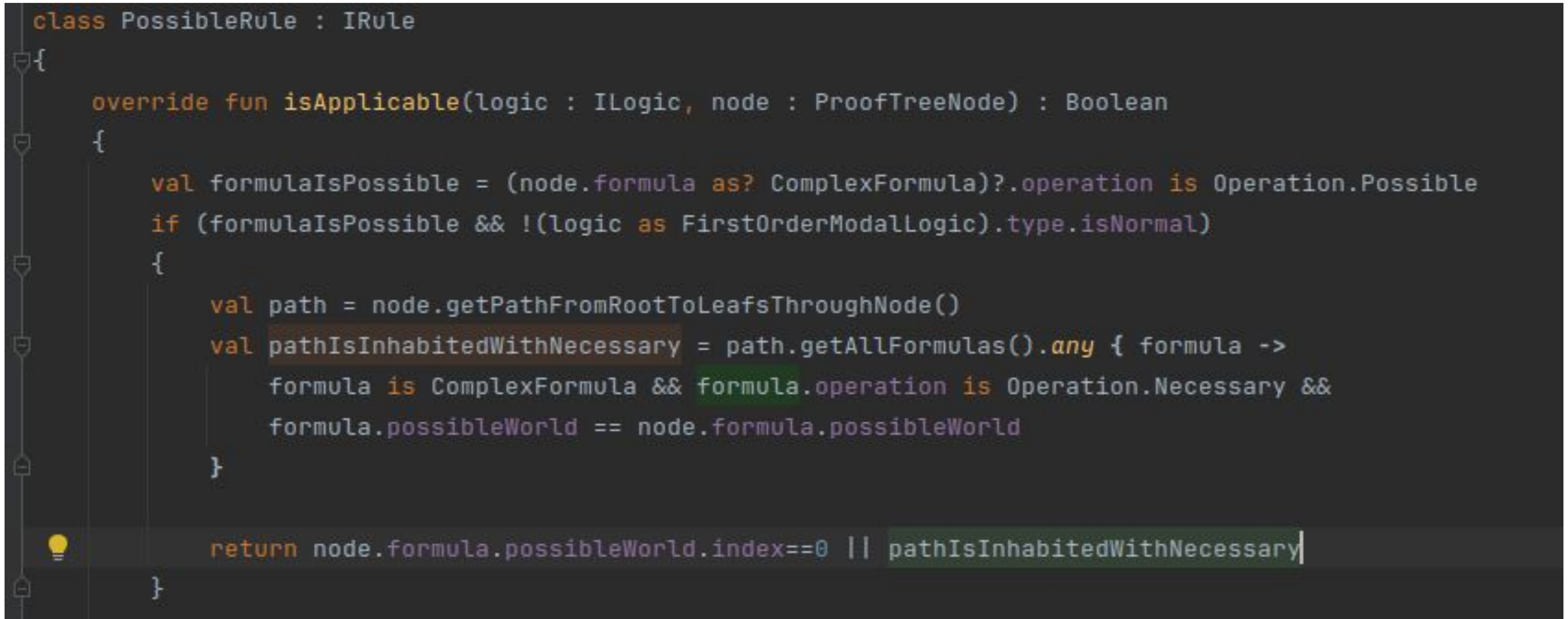
There are two vertices that start with w1:
w1 ⟶ w0, w1 ⟶ w2

$\boxed{P}$P is true at w1, thus P
should be true at w0 and w2.

The algorithm will append the following nodes
to the tree:
w1Rw0
P w0
w1Rw2
P w2

Non-normal modal logic ◇ rule

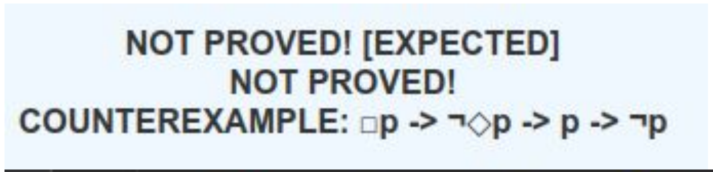
- On non-normal modal logic, the ◇ rule is applicable iff the current graph node (the current possible world) is normal.
- A possible world w_i is normal iff w_i = w0 or the current node's path is □-inhabited (it contains a node with □ as main operator).



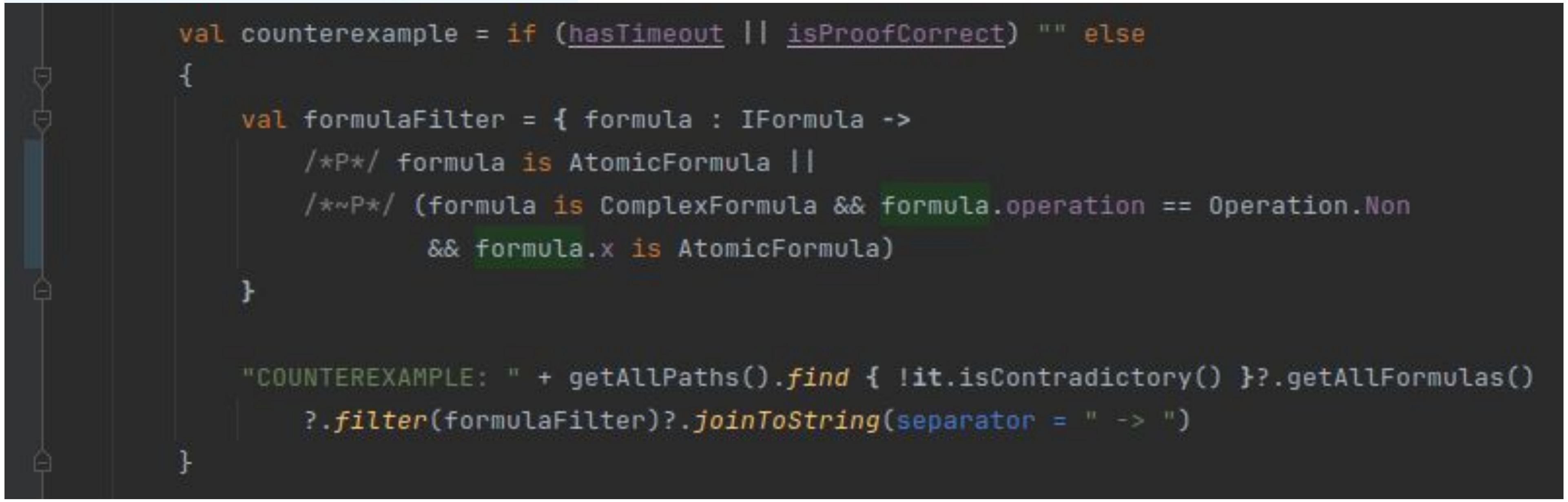
```
class PossibleRule : IRule
{
    override fun isApplicable(logic : ILogic, node : ProofTreeNode) : Boolean
    {
        val formulaIsPossible = (node.formula as? ComplexFormula)?.operation is Operation.Possible
        if (formulaIsPossible && !(logic as FirstOrderModalLogic).type.isNormal)
        {
            val path = node.getPathFromRootToLeafsThroughNode()
            val pathIsInhabitedWithNecessary = path.getAllFormulas().any { formula ->
                formula is ComplexFormula && formula.operation is Operation.Necessary &&
                formula.possibleWorld == node.formula.possibleWorld
            }

            return node.formula.possibleWorld.index==0 || pathIsInhabitedWithNecessary
        }
```

Reading the counterexample

- If the problem is not proved, the program shows a counterexample.
- The counterexample is built by simply extracting and filtering all the formulas from a non-contradictory path of the tree.



```
NOT PROVED! [EXPECTED]
NOT PROVED!
COUNTEREXAMPLE: □p -> ¬◇p -> p -> ¬p
```



```
val counterexample = if (hasTimeout || isProofCorrect) "" else
{
    val formulaFilter = { formula : IFormula ->
        /*P*/ formula is AtomicFormula ||
        /*~P*/ (formula is ComplexFormula && formula.operation == Operation.Non
                && formula.x is AtomicFormula)
    }

    "COUNTEREXAMPLE: " + getAllPaths().find { !it.isContradictory() }?.getAllFormulas()
        ?.filter(formulaFilter)?.joinToString(separator = " -> ")
}
```

Theoretical issues

Soundness and completeness

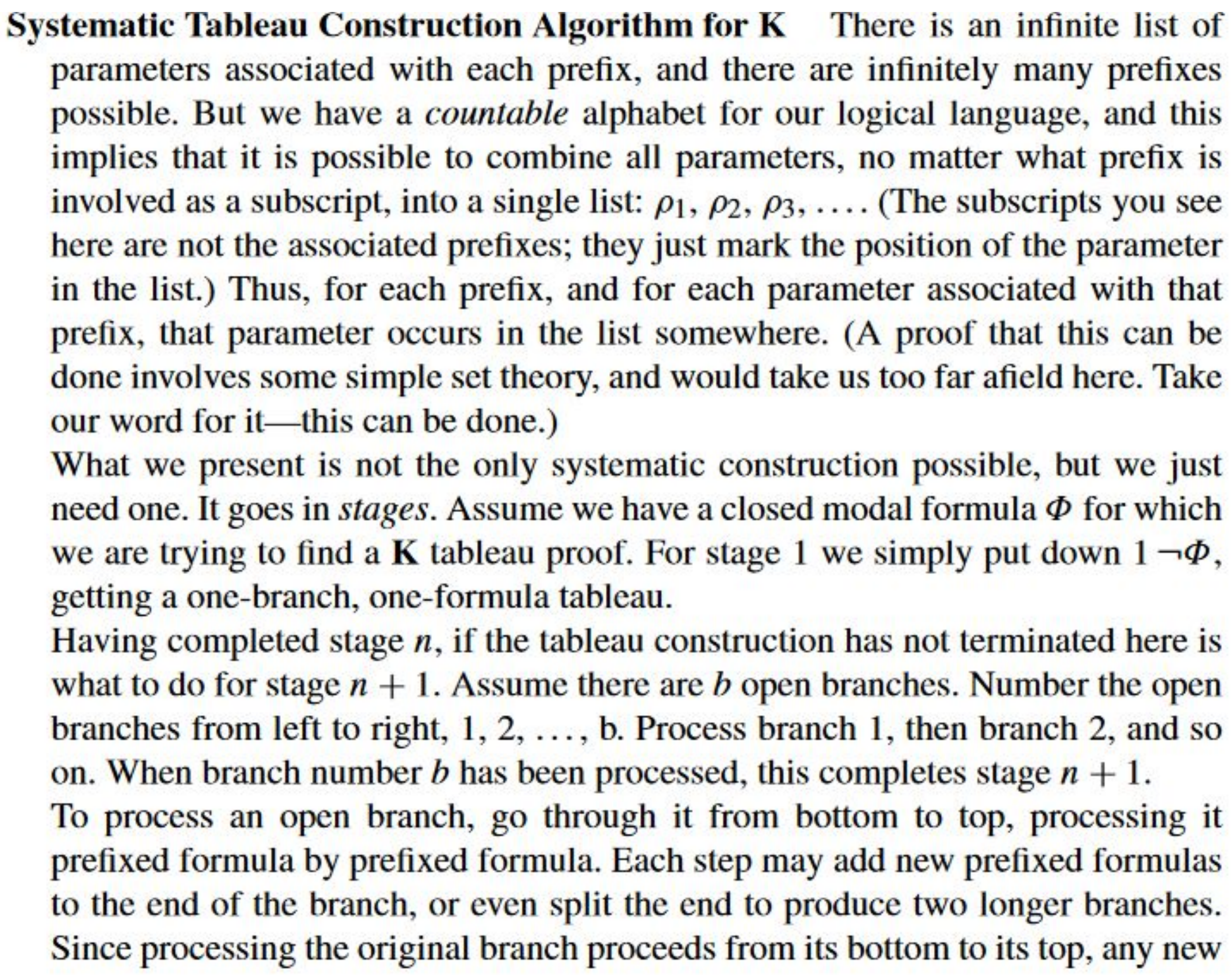
For **soundness**:

- It suffices to inspect the code and see that new nodes are not inserted where the original rules of Priest (2008) would not allow it and that there are no rules not allowed by Priest (2008) for any covered logic.
- We think this is immediate, barring programming bugs.

For **completeness**:

- Inspect the code and for each logic separately, derive a **fully-specified algorithm** in stages and steps, for example on the model of Fitting and Mendelsohn p.206-207 (next page)
- Then, the algorithm can be mapped on the completeness proofs of Priest (2008), using contraposition: if the procedure does not result in a closed tree, there is a counterexample readable out of it (even if infinite, although this gets tricky to prove).
- We think this is doable, similarly barring programming bugs for the first step.

Fair algorithm example in Fitting and Mendelsohn 2023 p.206-207

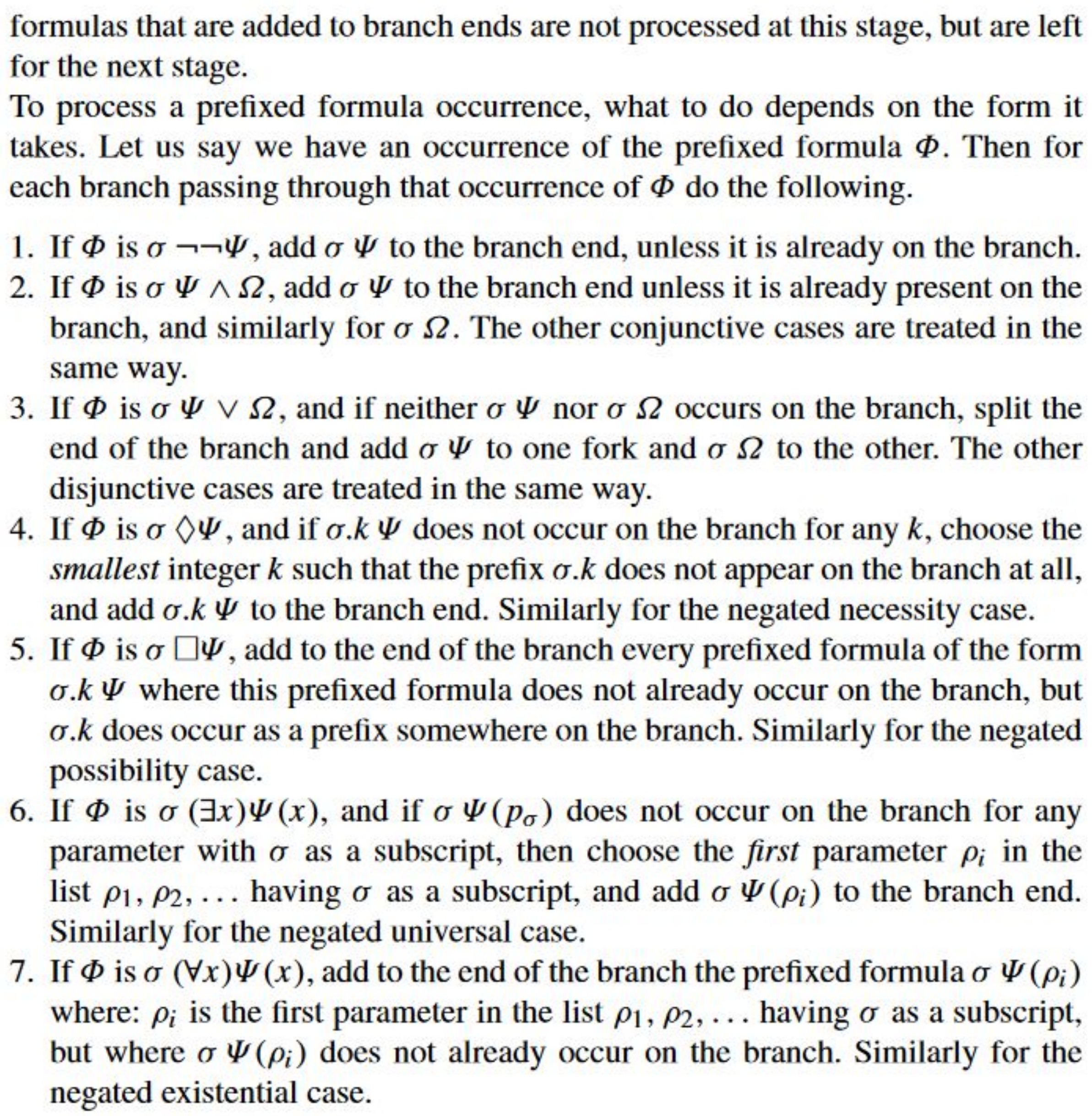


Systematic Tableau Construction Algorithm for K There is an infinite list of parameters associated with each prefix, and there are infinitely many prefixes possible. But we have a *countable* alphabet for our logical language, and this implies that it is possible to combine all parameters, no matter what prefix is involved as a subscript, into a single list: $\rho_1, \rho_2, \rho_3, \ldots$. (The subscripts you see here are not the associated prefixes; they just mark the position of the parameter in the list.) Thus, for each prefix, and for each parameter associated with that prefix, that parameter occurs in the list somewhere. (A proof that this can be done involves some simple set theory, and would take us too far afield here. Take our word for it—this can be done.)

What we present is not the only systematic construction possible, but we just need one. It goes in *stages*. Assume we have a closed modal formula Φ for which we are trying to find a **K** tableau proof. For stage 1 we simply put down $1\,\neg\Phi$, getting a one-branch, one-formula tableau.

Having completed stage n, if the tableau construction has not terminated here is what to do for stage $n+1$. Assume there are b open branches. Number the open branches from left to right, 1, 2, ..., b. Process branch 1, then branch 2, and so on. When branch number b has been processed, this completes stage $n+1$.

To process an open branch, go through it from bottom to top, processing it prefixed formula by prefixed formula. Each step may add new prefixed formulas to the end of the branch, or even split the end to produce two longer branches. Since processing the original branch proceeds from its bottom to its top, any new



formulas that are added to branch ends are not processed at this stage, but are left for the next stage.

To process a prefixed formula occurrence, what to do depends on the form it takes. Let us say we have an occurrence of the prefixed formula Φ. Then for each branch passing through that occurrence of Φ do the following.

1. If Φ is $\sigma\,\neg\neg\Psi$, add $\sigma\,\Psi$ to the branch end, unless it is already on the branch.
2. If Φ is $\sigma\,\Psi \wedge \Omega$, add $\sigma\,\Psi$ to the branch end unless it is already present on the branch, and similarly for $\sigma\,\Omega$. The other conjunctive cases are treated in the same way.
3. If Φ is $\sigma\,\Psi \vee \Omega$, and if neither $\sigma\,\Psi$ nor $\sigma\,\Omega$ occurs on the branch, split the end of the branch and add $\sigma\,\Psi$ to one fork and $\sigma\,\Omega$ to the other. The other disjunctive cases are treated in the same way.
4. If Φ is $\sigma\,\Diamond\Psi$, and if $\sigma.k\,\Psi$ does not occur on the branch for any k, choose the *smallest* integer k such that the prefix $\sigma.k$ does not appear on the branch at all, and add $\sigma.k\,\Psi$ to the branch end. Similarly for the negated necessity case.
5. If Φ is $\sigma\,\Box\Psi$, add to the end of the branch every prefixed formula of the form $\sigma.k\,\Psi$ where this prefixed formula does not already occur on the branch, but $\sigma.k$ does occur as a prefix somewhere on the branch. Similarly for the negated possibility case.
6. If Φ is $\sigma\,(\exists x)\Psi(x)$, and if $\sigma\,\Psi(p_\sigma)$ does not occur on the branch for any parameter with σ as a subscript, then choose the *first* parameter ρ_i in the list $\rho_1, \rho_2, \ldots$ having σ as a subscript, and add $\sigma\,\Psi(\rho_i)$ to the branch end. Similarly for the negated universal case.
7. If Φ is $\sigma\,(\forall x)\Psi(x)$, add to the end of the branch the prefixed formula $\sigma\,\Psi(\rho_i)$ where: ρ_i is the first parameter in the list $\rho_1, \rho_2, \ldots$ having σ as a subscript, but where $\sigma\,\Psi(\rho_i)$ does not already occur on the branch. Similarly for the negated existential case.

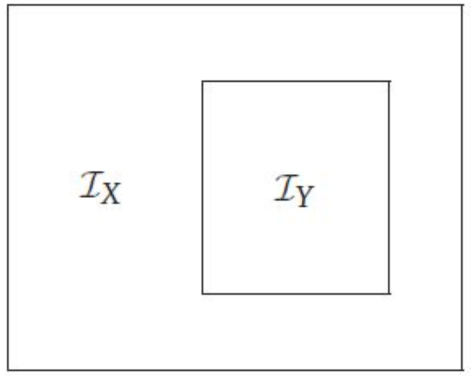
What is a logic - 1/2

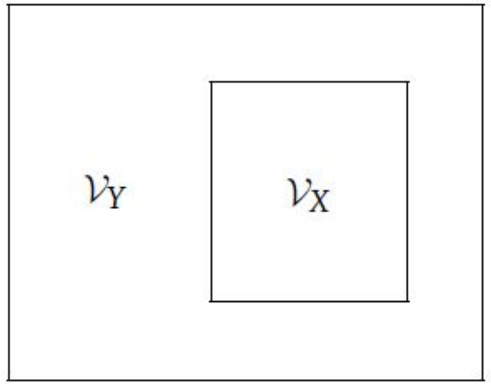
Standard definition: A logic is a formal language with a proof system (a calculus) and a semantics.

Priest (2008) favors semantical priority:

- "Most contemporary logicians would take the semantic notion of validity to be more fundamental than the proof-theoretic one, though the matter is certainly debatable"

- E.g. modal logic Kρ (T) is introduced as "We denote the logic defined in terms of truth preservation over all worlds of all ρ-interpretation, Kρ"





*"The logic determined by the class of interpretations Iy is an extension of that determined by the class Ix. [...] Vx and Vy are the sets of the inferences that are valid in the two logics.[..] **fewer interpretations, more inferences**" Priest (2008)*

What is a logic - 2/2

In the calculator, **a logic** is rather a calculus, a **combination** of:

a) Allowed operators (this determines the proof-theoretic rules)
b) Whether it is modal or not.
c) At modal systems:

- whether it is normal or not-normal (this changes the rule of ◇)
- whether it has a symmetric/reflexive/transitive accessibility relation (this changes the rule of □)
- whether it is temporal (this reverses the vertices of the world graph for [P] and <P>)

Thus, the functioning of tense logic K^t is now guaranteed by the *interaction of four checks*: that it has [P], <P>,[F], <F> as operators, that it is modal, that it is normal and that it is temporal.

To accommodate more logics from Priest (2008), we'll implement a stricter definition of a logic in code and move the rules inside each logic.

TODO list

1. Cover more logics from Priest (2008)

2. Some theoretical improvements
- Apply non-branching rules before branching rules (For readability)
- After timeout, have a *trial-and-error* module to look for finite countermodels.
- Detect infinite branches early by pattern (e.g. how many worlds are spawn)
- Implement a stricter definition of a logic in code and move the rules inside each.

3. Write a parser / extend the current one (to support quantifiers)

4. Develop better functionalities:
- Hovering a tree node with the mouse button should highlight the spawner node (the node from which the rule that appended the hovered node was applied)
- Clicking on a tree node should display the corresponding modal logic graph.
- The user will be able to select the applicable logic from a list.

Thank you!

Special thanks

- **The students at the 'Logic and software' course, 2024**, Faculty of Philosophy, University of Bucharest.
- **Graham Priest**

Bibliography

- Fitting, Mendelsohn 2023 - *First-Order Modal Logic*
- Girle 2010 - *Modal Logics and Philosophy*
- Johnson 2015 - *Tree Trimming: Four Non-Branching Rules for Priest's Introduction to Non-Classical Logic*
- Priest 2008 – *An Introduction to Non Classical Logic*
- Roy 2006 - *Natural Derivations for Priest, An Introduction to Non-Classical Logic*
- Roy 2017 - *More Natural Derivations for Priest*
- Smullyan 1968 – *First-Order Logic*