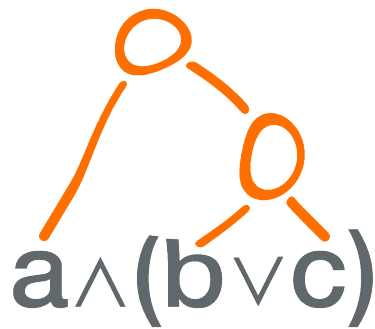Bachelor Thesis

Nyāya for iPad

Interactive Environment for BoolTool

Alexander Maringele (8517725)

alexander.maringele@gmail.com

$$a \wedge (b \vee c)$$

27 February 2013

Supervisor: Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

_____   _____
            Datum   Unterschrift

Abstract

If wishes were horses
Beggars would ride:
If turnips were watches
I would wear one by my side.
And if if's and an's were pots and pans,
The tinker would never work!

This nursery rhyme from the 16th century has some propositions! But are they true? Propositional logic provides a toolset to answer this question and many other questions, where propositions are made and connected. Nyāya for iPad will introduce the formalism of propositional logic interactively. A series of tutorials will put the user in a position to formalize this poem and to assess if the implications made by this poem can be true.

Acknowledgments

I am grateful to my family for their support and encouragement.

I thank my employer World-Direct eBussiness Solutions GmbH for its understanding of certain absences.

Last but not least I want to thank Dr. Georg Moser for accepting and supervising the project.

Contents

1. Introduction

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such way that we can reason about them formally. [9]

Logic is used daily. Although the core of reasoning and the foundation of computer science, logic still takes time and practice to be mastered soundly and completely. Uncertainty in the formalisms of logic leads to fundamental inaccuracies in the modeling of various applications in computer science. Therefore logic is a mandatory part of every computer science curriculum.

Each tool that promotes a deeper and more accurate understanding of formalisms and applications of logic, or simply provides an easier introduction to formal logic, is a welcome addition to the toolbox of training.

Propositional logic is – to put it simply - the entry point and the innermost core of logic. It is a powerful formal language, which can be used in many fields. But it also has limits of expressiveness and so there are many more powerful formal languages of logic, which can not be mastered without a sufficient understanding of propositional logic.

BoolTool by the Computational Logic Research Group at the University of Innsbruck allows the manipulation and evaluation of Boolean functions. The tool supports different representations of Boolean functions and a variety of different algorithms. Propositional logic is one of many representations of Boolean functions.

Nyāya *"(sanskrit ny-āyá, literally 'recursion' used in the sense of 'syllogism, inference') is [...] one of the [...] schools of Hindu philosophy – specifically the school of logic."* [3] *"Its followers believed that obtaining valid knowledge was the only way to obtain release from suffering."* [4]

Nyāya for iPad provides an interactive environment, that allows the user to learn simple facts about the formalism of propositional logic and standard transformations of Boolean functions. The environment is self-explanatory, re-implements the functionality of BoolTool, provides a graphical editor for syntax trees of propositional formulas, and works without a back end on a server. Nyāya supports the most effective learning techniques – steadily learning and practice testing [7] – with its combination of small bits of learning content and seemingly countless exercises.

2. Related Work

Nyāya for iPad is not the first program, that processes expressions of propositional logic. Besides powerful SAT-solvers and automatic provers, which rarely can be used by beginners, there are some software products, that can be used on a basic level of experience with propositional logic and Boolean expressions. Most of them provide a (short) introduction to propositional logic, but they seldom offer tutorials or exercises.

The following list presents some software products accessible via website and apps available for iPad.

2.1. Websites

- **PLogic Applet** by S. Lukins, A. Levicki, and J. Burg at the Wake Forest University, is a tutorial program for propositional logic "that serves a double role as an educational tool and a research environment"[1] and focuses on interactive theorem-proving.

- **BoolTool** by the Computational Logic research group at the University of Innsbruck "is an interface to the program BoolTool which allows the manipulation and evaluation of boolean functions."[2]. It computes truth tables and binary decision diagrams and checks for satisfiability and validity.

- **Wolfram Alpha** by Wolfram Research – the makers of Mathematica – supports propositional logic by calculating truth tables and minimal forms, but does not draw syntax trees or binary decision diagrams. Instead it draws a logic circuit and a Venn diagram from the input. Additionally it calculates the truth density and the boolean operator number of the input formula.

  Wolfram Alpha does not calculate exactly for an arbitrary number of variables in propositional formulas. For example no truth table, conjunctive normal form, no disjunctive normal form is calculated for the input $a \veebar b \veebar c \veebar d \veebar e \veebar f \veebar g \veebar h \veebar i \veebar j$. The truth density is only sampled and does not match the exact result 0.5.

---

[1] http://www.cs.wfu.edu/~burg/papers/PropLogic.pdf
[2] http://cl-informatik.uibk.ac.at/software/booltool/?page=info

2.2. Apps for iPad

Some apps for iPad or iPod touch that handle aspects of propositional logic.

- **Constraints** by Davide Cucciniello is a "SAT based propositional (boolean) logic engine defined by a list of models, where a model includes a list of constraints (a Knowledge Base) each defined by a propositional formula (x and (y or not z)) including a set of propositional (boolean) variables (x,y,z) and operators (and,or,not)."[3] Constraints lets you build models in an interactive way. It also provides a short introduction in propositional logic, but it does not deliver any tutorials.

- **Truth Table Generator** by Mertz Werkz LLP "constructs truth tables for the Boolean expressions you enter".[4] It does that nicely by using standard symbols of propositional logic (and a limited set of atoms), but nothing more.

- **Boolean Algebra Cheat Sheet** by Clint Johnson presents two pages of logic "rules and laws" in low picture quality. So "Containing all of your Boolean logic needs."[5] does not feel quite right.

- **Logic Mania** by Imagination Creations presents seven logic gates in a list. Their semantics are shown in a per-gate-simulation where the user chooses the input and the app shows the result. A combination of gates is not possible. It's quite nice but the statement "THE [sic] reference application for students studying logic gates"[6] seems exaggerated.

- **Circuit Coder** by Trycycle Design HB is a game about building digital circuits, which is an application of propositional logic.

---

[3] `http://www.mysvc.it/myapps/constraints/`

[4] `http://www.mertzwerkz.com/truth.html`

[5] `http://itunes.apple.com/at/app/boolean-logic-cheat-sheet/id341959531?mt=8`

[6] `http://itunes.apple.com/at/app/logic-mania/id434019152?mt=8`

3. Concept

The core of this concept builds on tutorials with exercises and a playground where the acquired knowledge can be used in different modes. A glossary of terms and a formula calculator to check formulas will support the user in her learning.

3.1. Tutorials

Nyāya for iPad covers syntax and semantics of propositional logic, but only some basic equivalences. It introduces normal forms, truth tables and binary decision diagrams as different but equivalent representations of Boolean functions.

The structure of the tutorials and the content of the exercises are heavily based on related sections of the book *Logic in Computer Science* by M. Huth and M. Ryan. [9] Additional content was taken from the lecture *Logic* by A. Middeldorp. [12]

```
5   <array>    <!-- root with title and sections -->
6     <string>Tutorials</string>       <!-- title -->
7     <array>    <!-- section with section title and tutorials-->
8       <string>Introduction</string>     <!-- section title -->
9       <array>  <!-- tutorial with title and id -->
10        <string>Motivation</string>  <!-- tutorial title -->
11        <string>11</string>            <!-- tutorial id -->
12      </array>
13      □
```

Table 3.1.: Tutorials.plist – the configuration file for all tutorials

General and specific configurations for the tutorials and additional content for the exercises are stored in simple property lists (see Table B.1 on page 37) which are xml-files with some basic data types for data and collections, that can be easily read and interpreted by different programs. The file "Tutorials.plist" (see Table 3.1) provides the basic data about available tutorial sections and tutorial titles to build a navigation menu programmatically.

Every tutorial includes a teaching part with examples (HTML-files in UTF-8 encoding) and an interactive knowledge checker with exercises, where the user gets immediate feedback. Every interactive part will present some instructions to master the task.

3.1.1. Introduction

The first tutorial section with four tutorials gives an informal introduction to modeling, i.e. the translation from sentences in natural language to sentences in the formal language of propositional logic. Three of the four tutorials use the same content file to present some questions to check the user's comprehension. The fourth tutorial does not include an exercise.

```
 5  <array>
 6    <array>
 7      <string>If the barometer falls, then it will rain.</string>
 8      <string>if p then q</string>
 9      <string>the barometer falls; it will rain</string>
10      <string>p → q</string>
11    </array>
12  □
```

Table 3.2.: QA1.plist – content file for the first three tutorials with exercises

The content file "QA1.plist" (see Table 3.2) contains an array (starts at line 5) of arrays of strings that represents sentences and sample solutions for the first three exercises. The first sentence/solutions array of strings starts at line 7 and ends at line 10. The first string in the first array (at line 7) is a sentence, which must be analyzed in all three exercises. The next string (at line 8) is a correct solution for the first exercise, the third string (at line 9) is a correct solution for the second exercise, and the last string in the array (at line 10) is a correct solution for the third exercise. The next array of strings would start at line 13.

```
 5  <dict>
 6    <key>questionLabelText</key>
 7    <string>Guess the structure ...</string>
 8    <key>answerLabelText</key>
 9    <string>Use ...</string>
10    <key>solutionLabelText</key>
11    <string>Sample solution</string>
12    <key>questionsFile</key>
13    <string>QA1</string>
14    <key>solutionIndex</key>
15    <integer>1</integer>
16  </dict>
```

Table 3.3.: 11.plist – configuration file for the first exercise

Each tutorial has its own configuration file (see Table 3.3) with a dictionary that describes the exercise and defines the index of the sample solu-

tion, with which the user's answer will be checked with some tolerance. The `solutionIndex` expresses which string is to be used as sample solution in the exercise.

Applications – №11. *"The aim of logic in general is to reason about situations."* [9] The first tutorial shows, how two different paragraphs in natural language can be reduced to the same formal sentence *"If p and not q, then r. Not r. p. Therefore, q"*. If the reasoning is correct, the user can use it in both situations.

In the interactive part the user has to 'guess' the formal structure of composite sentences in natural language. The sentence 'If the barometer falls, then it will rain' from line 7 of `QA1.plist` (see Table 3.2 on the preceding page) has to be translated into 'if p then q' from line 8 (the solution index is defined as 1 in `11.plist`, see Table 3.3 on the previous page).

Propositions – №12. The concept of propositions – simple declarative sentences – as indivisible building blocks of propositional logic is explained in more detail. Some counter-examples are presented and the ambiguity of natural language is demonstrated.

The user has to find the propositions in composite sentences. The correct answer for 'If the barometer falls, then it will rain' would be 'the barometer falls; it will rain' from line 9 (the solution index is defined as 2 in `12.plist`).

Connectives – №13. The basic symbols of propositional logic to connect propositions are introduced. The symbols "→", "¬", "∧", and "∨" will replace "if then", "not", "and" and "or".

The user hast to find a matching formula in propositional logic. For the sentence 'If the barometer falls, then it will rain' a correct answer is 'p → q' from line 10 (the solution index is defined as 3 in `13.plist`), but '$\neg p \lor q$' would be accepted too.

Limits – №14. Some examples are given to demonstrate that not every situation can be modeled in propositional logic. There is no exercise for this tutorial.

3.1.2. Syntax

After the informal introduction into the atoms and connectives of propositional logic, the formal aspects of propositional logic will be introduced to the user. Again each exercise is configured by a configuration file similar to `11.plist` (see Table 3.3 on the preceding page), but without a `questionsFile` or a `solutionIndex`, because the content of the syntax exercises will be generated by the app. The user's solution will be checked without tolerance.

Definition – №21. The definition of well formed formulas is explained by a strict grammar with mandatory parenthesis.

$$P \quad ::= \quad p \quad | \quad (\neg P) \quad | \quad (P \land P) \quad | \quad (P \lor P) \quad | \quad (P \to P)$$

The user has to check whether a formula matches this strict definition of a propositional formula. If the formula is well formed, the user has to write down the name of the root connective, i.e negation, conjunction, disjunction or implication. If the formula is not well formed, the user has to leave the answer field empty.

Conventions – №22. The conventions of precedences and associativity of connectives to save parentheses are introduced. The user has to rewrite formulas from strict syntax to formulas using conventions and vice versa.

$$
\begin{array}{llll}
P & ::= & D & | & D \to P \\
D & ::= & C & | & D \vee C \qquad\qquad N \quad ::= \quad T \quad | \quad \neg N \\
C & ::= & N & | & C \wedge N \qquad\qquad T \quad ::= \quad p \quad | \quad (P)
\end{array}
$$

Sub-formulas – №23. Sub-formulas (starting from atoms) are used to build bigger formulas. Big formulas are build from many sub-formulas. In the exercise the user has to extract the set of sub-formulas from composite formulas. A correct answer for '$a \vee b \wedge c$' would be '$a, b, c, b \wedge c, a \vee b \wedge c$'. The order of the sub-formulas does not matter and the user can use parentheses as she likes it. But no sub-formula may occur more than once. So the answer '$c, b \wedge c, b, a \vee b \wedge c, a$' would be right too, but '$a, b, c, b \wedge c, b, a \vee b \wedge c$' would be a wrong answer.

Syntax Trees – №24. Syntax trees are introduced as graphical representations of well formed formulas. The user has to write formulas for presented syntax trees. The user can use parentheses as she likes, but still the written formulas have to match the syntax trees exactly. The formula '$a \vee b$' does not match the syntax tree representing the formula '$b \vee a$', where the atom node 'a' is on the right branch of the disjunction node '$\vee$'.

Top and Bottom – №25. The symbols "$\top$" (top) and "$\bot$" (bottom) are defined as abbreviations of conjunctions and disjunctions of formulas with their negation). Additional basic equivalences are introduced. The user has to simplify given formulas using tautologies, contradictions and basic equivalences.

$$
\begin{array}{lll}
P \vee \neg P \equiv \neg P \vee P \equiv \top & P \wedge \neg P \equiv \neg P \wedge P \equiv \bot & \\
P \vee P \equiv P & P \wedge P \equiv P & \\
P \vee Q \equiv Q \vee P & P \wedge Q \equiv Q \wedge P & \neg \neg P \equiv P
\end{array}
$$

3.1.3. Semantics

After the introduction of well formed formulas the meaning of logical connectives is defined. Configuration files are used again, the exercise context is generated randomly by the app, but exercise №33 also uses a context file with exemplary entailments that would not be likely generated.

Valuation – №31. The truth assignment for atoms $(p, q, r, \dots)$ is introduced. The extended valuation for negation, conjunction, disjunction and implication is defined. The valuation of arbitrary formulas by evaluating all sub-formulas is described. The user has to guess or calculate the evaluation of a given formula with a given truth assignment for the atoms of the formula.

Truth Tables – №32. The concept of truth tables is explained and demonstrated with the basic connectives. The valuation of arbitrary formulas by evaluating all sub-formulas is demonstrated again. The user has to fill in truth tables for formulas with a maximum of three atoms.

Entailment and Equivalence – №33. Semantic entailment and equivalence is defined and explained with various examples. The user has to guess or calculate whether a given semantic entailment or equivalence holds.

Satisfiability and Validity – №34. The properties satisfiability and validity of propositional formulas are defined and demonstrated. The relationship of these properties with top and bottom is mentioned. The user has to guess or calculate whether a given formula is satisfiable or valid.

3.1.4. Normal Forms

The advantages of propositional formulas in special shapes will be emphasized. The definitions of implication free forms, negation normal form, conjunctive normal forms and disjunctive normal forms will be provided. A recipe to transform arbitrary propositional formulas into specific normal forms will be delivered. Rules to detect satisfiability or validity from normal forms will be mentioned.

Implication Free Form – №41.

By removing the rule $P ::= D \to P$ in 3.1.2 on the previous page and optimizing the resulting grammar the implication free form is defined for propositional formulas. The equivalence transformation for removing implications is presented and proved with a truth table.

$$
\begin{array}{llll}
D & ::= & C & | & C \vee D \\
C & ::= & N & | & N \wedge C
\end{array}
\qquad
\begin{array}{llll}
N & ::= & T & | & \neg N \\
T & ::= & p & | & (D)
\end{array}
$$

The user has to check whether a given formula is implication free. If the formula is not implication free the user has to transform it into an implication free form using the equivalence transformation.

$$
P \to Q \equiv \neg P \ \vee \ Q
$$

Negation Normal Form – №42.

By removing all rules for arbitrary negations and optimizing the resulting grammar, the negation normal form is defined for propositional formulas. The equivalence transformations to transform negations of conjunctions, negations of disjunctions, and double negations into negation normal forms are presented and proved with truth tables.

$$
\begin{array}{lclcl}
D & ::= & C & | & C \vee D \\
C & ::= & N & | & N \wedge C
\end{array}
\qquad
\begin{array}{lclcl}
N & ::= & (D) & | & L \\
L & ::= & p & | & \neg p
\end{array}
$$

The user has to check whether a given formula is in negation normal form. If the formula is not in negation normal form the user has to transform it into negation normal form using the equivalence transformation for implications and the equivalence transformations for negations. The rule for double negations is already known from 3.1.2 on page 7.

$$
\neg(P \wedge Q) \equiv \neg P \vee \neg Q \qquad \neg\neg P \equiv P \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q
$$

Conjunctive Normal Form – №43.

A formula in conjunctive normal form is a conjunctive concatenation of disjunctive clauses. A disjunctive clause is a disjunctive concatenation of literals. A literal is an atom or the negation of an atom. The distributions of disjunctions over conjunctions is presented and proved with a truth table.

$$
\begin{array}{lclcl}
C & ::= & (D) & | & (D) \wedge C \\
D & ::= & L & | & L \vee D
\end{array}
\qquad
\begin{array}{lclcl}
L & ::= & p & | & \neg p
\end{array}
$$

The user has to rewrite formulas to conjunctive normal forms. All presented formulas are in negation normal form but may already be in conjunctive normal form. So at most the user have to apply the distributions of disjunctions over conjunctions multiple times as the equivalence transformations towards conjunctive normal form.

$$
P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R) \qquad (P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)
$$

Disjunctive Normal Form – №44.

A formula in disjunctive normal form is a disjunctive concatenation of conjunctive clauses. A conjunctive clause is a conjunctive concatenation of literals. A literal is an atom or the negation of an atom. The distributions of conjunctions over disjunctions is presented and proved with truth tables.

$$
\begin{array}{lclcl}
D & ::= & C & | & C \vee D \\
C & ::= & L & | & L \wedge D
\end{array}
\qquad
\begin{array}{lclcl}
L & ::= & p & | & \neg p
\end{array}
$$

The user has to rewrite formulas to disjunctive normal forms. All presented formulas are in negation normal form but may already be in disjunctive normal form. So at most the user have to apply the distributions of conjunctions over disjunctions multiple times as equivalence transformations towards disjunctive normal form.

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge Q) \qquad (P \wedge Q) \wedge R \equiv (P \wedge R) \vee (Q \wedge R)$$

3.1.5. Binary Decision Diagrams

The definition of Boolean functions is introduced. Propositional formulas and syntax trees from the previous sections are representations of Boolean functions. Boolean expressions and binary decision diagrams are additional representations of Boolean functions.

Boolean functions - №51. N-ary Boolean functions are introduced as mapping from n-tupels of 0s and 1s to the set $0, 1$. The Boolean operators $^- \ + \ \cdot \ \oplus$ are introduced as special representations of specific unitary and binary Boolean functions. For convenience in the exercises an exclamation mark $!(B)$ will be used for the negation of a Boolean expression B instead of the bar $\bar{B}$. The user has to guess or calculate the results for composite expressions.

Boolean expressions - №52. The grammar for Boolean expressions is defined. The equivalence relation of Boolean expressions is explained.

$$B \quad ::= \quad p \quad | \quad (\overline{B}) \quad | \quad (B \cdot B) \quad | \quad (B + B) \quad | \quad (B \oplus B)$$

The user has to rewrite formulas with propositional symbols into Boolean expressions. The implication may be a challenge for some users, but is doable because the implication free form was introduced in an earlier tutorial.

Binary Decisions - №53. Binary decision nodes are introduced. The valuation of binary decision trees is explained. The user has to fill in the result nodes in a binary decision tree for a given Boolean expression.

Binary Decision Diagrams - №54. Arbitrary binary decision diagrams are explained. Ordered and reduced binary decision diagrams are introduced. The user has to check whether a binary decision diagram is ordered and/or reduced.

3.2. Playground

On the playground the user can create arbitrary propositional formulas by building syntax trees interactively. Truth assignments can be applied and will be visualized. The playground supports two modes for working with a syntax tree.

3.2.1. Free mode

In the free mode an arbitrary formula can be manipulated by adding, replacing and removing connectives and atoms. The process starts with an atom, which can be expanded or replaced. Connectives can be changed at any time, and sub-formulas can be reordered. The smallest possible formula consists of one atom with a symbol from the set $\{\top, \bot, p, q, r, ...\}$.

Truth values can be assigned to leaf nodes, which represent the atoms of the corresponding propositional formula. By default no truth values are assigned to atoms, which is visualized by a blue color of the node. A green color represents 'true', and a red color 'false'. The valuation of connective nodes is calculated automatically and displayed using the same blue, green and red colors.

3.2.2. Locked mode

In the locked mode only equivalence transformations (see Table 3.4 on the next page) can be applied to nodes of a syntax tree. The offered transformations depend on the type of node selected by the user.

In the locked mode the valuation of atoms can not be changed, but the automatic valuation of sub-formulas is still active. The user can observe that equivalence transformations do no change the valuation of the root node.

3.2.3. Remarks

Connective nodes of obvious tautologies like $P \vee \neg P$ or $P \rightarrow P$ are displayed with a green color – independent form the valuation status of any sub-node.

Connective nodes of obvious contradictions like $P \wedge \neg P$ are displayed with a red color – independent from the valuation status of any sub-node.

While working with a syntax tree, the corresponding propositional formula is shown on top of the playground.

While selecting a sub-node of the syntax tree – the root of a sub-tree – for manipulation the corresponding sub-formula will be highlighted in the corresponding propositional formula.

3.3. Glossary

Terms, definitions and truth tables from the tutorials are listed in the glossary in a searchable and hyperlinked form.

P	$\rightarrow$	Q			$\neg$	$\neg P$
$\neg P$	$\vee$	Q			$\boldsymbol{P}$	

	$\neg$	$(P \vee Q)$			$\neg$	$(P \wedge Q)$
$\neg P$	$\wedge$	$\neg Q$		$\neg P$	$\vee$	$\neg Q$

P	$\vee$	P		P	$\wedge$	P
	$\boldsymbol{P}$				$\boldsymbol{P}$	

P	$\vee$	$\neg P$		P	$\wedge$	$\neg P$
	$\top$				$\bot$	

P	$\vee$	$\top$		P	$\wedge$	$\top$
	$\top$				$\boldsymbol{P}$	

P	$\vee$	$\bot$		P	$\wedge$	$\bot$
	$\boldsymbol{P}$				$\bot$	

P	$\vee$	Q		P	$\wedge$	Q
Q	$\vee$	P		Q	$\wedge$	P

P	$\vee$	$(Q \wedge R)$		P	$\wedge$	$(Q \vee R)$
$(P \vee Q)$	$\wedge$	$(P \vee R)$		$(P \wedge Q)$	$\vee$	$(P \wedge R)$

$(P \wedge Q)$	$\vee$	R		$(P \vee Q)$	$\wedge$	R
$(P \vee R)$	$\wedge$	$(Q \vee R)$		$(P \wedge R)$	$\vee$	$(Q \wedge R)$

P	$\vee$	$(Q \vee R)$		P	$\wedge$	$(Q \wedge R)$
$(P \vee Q)$	$\vee$	R		$(P \wedge Q)$	$\wedge$	R

P	$\leftrightarrow$	Q		P	$\underline{\vee}$	Q
$P \rightarrow Q$	$\wedge$	$Q \rightarrow P$		$\neg(P \rightarrow Q)$	$\vee$	$\neg(Q \rightarrow P)$

Table 3.4.: Available equivalence transformations in locked mode

3.4. BoolTool

The interactive environment contains the functionality of the web front end of
`BoolTool`[1] developed and provided by the Computational Logic Research Group
of the Institute of Computer Science at the University of Innsbruck. Nyāya's
reimplementation is referenced as Nyāya's BoolTool in this document.

3.4.1. Input

Nyāya's BoolTool supports Latin letters as identifiers for atoms. It reads the
symbols `T` as top, `F` as bottom, `!` as negation operator, `&` as conjunction operator,
`+` as disjunction operator, `>` as implication operator, `^` as exclusive disjunction
operator, and `(` and `)` as parentheses. These ASCII-symbols can be found
on almost every keyboard. Thereby any formula of propositional logic can be
created completely using only a keyboard attached to an iPad running Nyāya.

Nyāya's BoolTool also accepts standard and additional symbols of proposi-
tional logic and Boolean expressions (see Appendix A on page 36 for a complete
list of accepted ASCII characters and symbols), which are provided by extended
and customized on-screen keyboards. Nyāya's BoolTool does not limit identi-
fiers to Latin characters, since, for example, Chinese symbols are meaningful
names for atoms (see Figure 3.1).

Figure 3.1.: Implication with Chinese symbols for rain and wet

3.4.2. Output

Nyāya's BoolTool calculates validity and satisfiability, a conjunctive normal
form, a disjunctive normal form, the truth table and a ordered and reduced
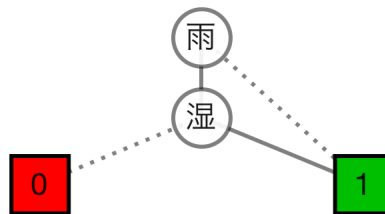binary decision diagram (see Figure 3.2) of the user's input .

Figure 3.2.: Reduced ordered binary decision diagram

---

[1] cl-informatik.uibk.ac.at/software/booltool

4. Tools and Processes

4.1. Tools

4.1.1. Developing for the iPad

First steps in developing applications for iPad can be made on a computer running Mac OS X (OS) and Xcode (IDE), which is a free download from the Mac App Store. Knowledge of object-oriented programming, object-oriented design patterns, and the C programming language builds a good base to learn the main platform-specific technologies Objective-C, Cocoa, and memory management with automatic reference counting.

In addition to the technical skills, an overview of the necessary steps for Development and Distribution (p. 15) for the iPad platform should be obtained.

Xcode is Apple's integrated development environment to create software for OS X and iOS devices. It includes an iPad simulator, complete API documentations and development guides to various topics. Xcode supports the source management systems Git and Apache™ Subversion®. Git will be installed along various other development tools with Xcode.

Objective-C is a reflective, object-oriented extension of the C programming language, which was developed by Tom Love and Brad Cox in the early 1980s (a few moments earlier than C++). The syntax for objects and methods is based on Smalltalk. Objective-C method calls will be bound to functions at runtime with no need to be defined formally at compile time. Objective-C code is organized in header `*.h` and implementation `.m` files with `@interface` and `@implementation` sections for the declaration and definition of classes, properties, fields and operations. The Objective-C keyword `@interface` must not be mixed up with the Java/C# `interface` directive. The latter one corresponds to `@protocol`.

Cocoa Touch is the name for the object-oriented APIs for iOS (the operating system for iPhone and iPad). Cocoa Touch covers the platform specific Objective-C runtime and a platform specific set of libraries, the frameworks.

- The Foundation framework provides the basis for programming with Objective-C. In addition to the memory and exception handling it includes the base classes for strings, values, lists, sets and files.

- The UIKit framework provides a set of classes and functions for implementing (touch based) graphical user interfaces. The architecture of this

framework follows the Model-View-Controller pattern and the implementation provides a myriad of view and controller classes.

Memory Management by the Objective-C run time is based on reference counting. The run time does not offer garbage collection – which is surprising and annoying specifically for Java or C# developers. But the necessary retains and releases are added automatically at compile time. Most of the time memory management by automatic reference counting is as easy to use as garbage collection. But one important consequence is that strong object graphs must be implemented as directed acyclic graphs. For example a circular strong linked list would never be released.

The advantages are the deterministic and economical run-time behavior. Objects are destroyed at a defined time and there is no need for a separate thread for garbage collection. The main disadvantage is that deallocated objects can still be referenced, which would corrupt the event loop without the possibility of error recovery by exception handling.

Development and distribution for Apple's platforms includes the coding effort and expenses for administration and configuration. To install software on iOS devices – especially for distribution via the App Store – applications must be cryptographically signed. The necessary certificates are created in the paid members section on Apple's Developer Portal. Certificates, development and distribution profiles contain a bunch of identifiers for the identification and differentiation of developers (Team ID), applications (App-ID) and a list of permissions (entitlements).

4.1.2. Source Code Management

Every software project includes the need for source code management, which minimizes the risk of losing working code by human error or technical failure. This should be done by powerful, yet lightweight tools.

Git [1] is a distributed source code management and revision control system - originally developed by Linus Torvalds for Linux kernel development. Git supports local repositories with full source code history for development. Local repositories can be synchronized and merged with remote repositories for collaboration and backup [6].

Since Apple's IDE Xcode works with Git's local repositories out of the box, it was obvious to use Git as source control for the project.

Github [2] is a hosting service for projects that use Git. GitHub offers free accounts for public repositories, which can be seen and downloaded by anyone using a web-browser or a Git client. Committing changes is restricted to users chosen by the owner of the repository on GitHub.

As Nyāya is a student project and the source code should be publicly accessible, GitHub seemed suitable to house the project.

4.2. Project execution

Projects with only one developer are easy to manage, because all work must be done by the same person and there is no coordination overhead. Nevertheless small projects need some structure too for a successful outcome.

4.2.1. Phases

The development of Nyāya for iPad was roughly divided into four phases, which are substantially coincident with four check-in blocks (see Figure 4.1).

Exploring user interface capabilities. After the informal specification of the feature set and the initial presentation of the project the capabilities of graphical visualization and user interaction were explored on iPad. In this phase several prototypes were developed for interactive syntax trees and exercises.
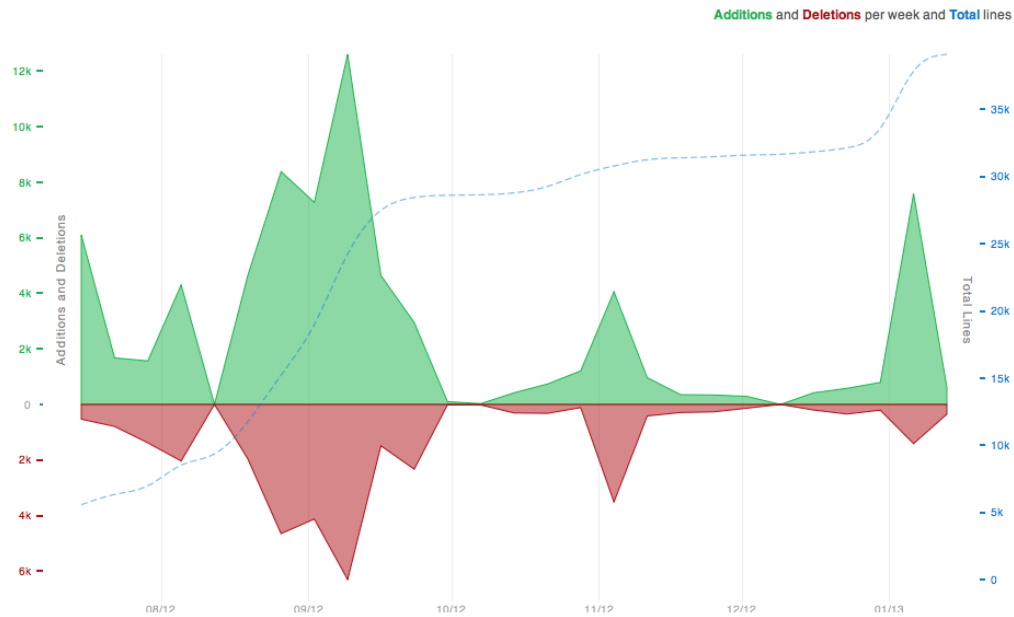
Figure 4.1.: Additions and deletions per week on GitHub

Developing core components. After the formulation of the general concept with tutorials including exercises, a playground, a glossary, and the embedding of BoolTool, the main classes for the representation of abstract syntax trees and binary decision diagrams were implemented. After several failed attempts to cross-compile `BoolTool`'s sources `OCaml` and to use the binaries on iPad, the functionality of `BoolTool` was reimplemented as Nyāya's BoolTool in Objective-C.

Adding content, controllers, and configurations. Although the general concept was developed, content had to be expounded. The tutorials had to be written and the exercises had to be generated. Content and configuration files had to be added to the project. The different views had to be controlled and the navigation between the different use cases had to be added.

Finishing. In the final phase all parts of the program were reviewed. Nyāya's content was amended and this document was written. Nyāya was prepared for distribution in the App Store.

4.2.2. Development model

The development of this software did no follow a specific development model, but did borrow some principles from agile development.

Fail-Fast. The interaction with syntax trees and the integration of `BoolTool` were identified as essential parts with a high risk potential. The first one was cleared by exploring the user interface capabilities. The second one failed at the beginning of developing core components and was substituted with a native implementation of the functionality of `BoolTool` under direct control of the project.

Test-driven development. Parsers, trees and operations on trees were the ideal application for test driven development. First, the expected results were defined in test cases, then the functionality was implemented. So it was ensured that at every point of the development cycle the core of the application was still working as defined, when all unit tests had succeeded.

Refactoring. Due to occurring memory and performance issues on the iPad some implementations had to be reconsidered and some functionality had to be reimplemented. This could be done without complications because the local repository provided a complete development history and unit tests provided a quick check, whether the reimplementation was correct.

Use cases. With the written concept (see chapter 3 on page 4) of an interactive environment for BoolTool that enables the learning of simple facts about propositional logic the collection of use cases was defined. These use cases represent the real value of the project and they mark the criterion for completion of development.

5. Implementation

5.1. Architecture

The general design of Nyāya follows the Model-View-Controller pattern (MVC), which is originated in Smalltalk [8, p.4], that separates the representation of data from the user interaction. Cocoa Touch (see section 4.1.1 on page 14) encourages the use of MVC by providing a rich set of useful views and controllers within the UIKit framework. These views and controllers cover many use cases of data presentation and user interaction.

5.2. Model

Nyāya must hold different representations of Boolean functions, for example propositional formulas, abstract syntax trees and binary decision diagrams.

Since propositional formulas and Boolean expressions are formal languages and therefore subsets of the set of all Unicode strings, they are easily represented by the standard string classes of actual frameworks in general and Objective-C in particular.

Due to the unambiguous relationship between propositional formulas and Boolean expressions Nyāya allows the use of a mixed syntax and additional symbols as input for the user's convenience. The input expression $!a \wedge b + c \oplus a$ translates into the propositional formula $\neg a \wedge b \vee c \veebar a$ (see section 5.2.2 on page 23).

5.2.1. Abstract Syntax Trees

Cocoa does not provide a class to represent trees, but with instances of a custom class (see Figure 5.1), that implements the composite pattern [8, p.163ff] arbitrary graphs and therefore syntax trees can be represented easily .
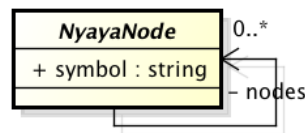

Figure 5.1.: Node class to represent an abstract syntax tree

Node Class Cluster

The node class is implemented as a class cluster, which is a Cocoa design pattern [5, p.282ff] – an adaptation of the the abstract factory design pattern [8, p.87ff]. The abstract and public class `NyayaNode` provides a set of static and public methods (see Figure 5.3 on the following page), that returns instances of non public sub-classes (see Figure 5.2) of `NyayaNode`.
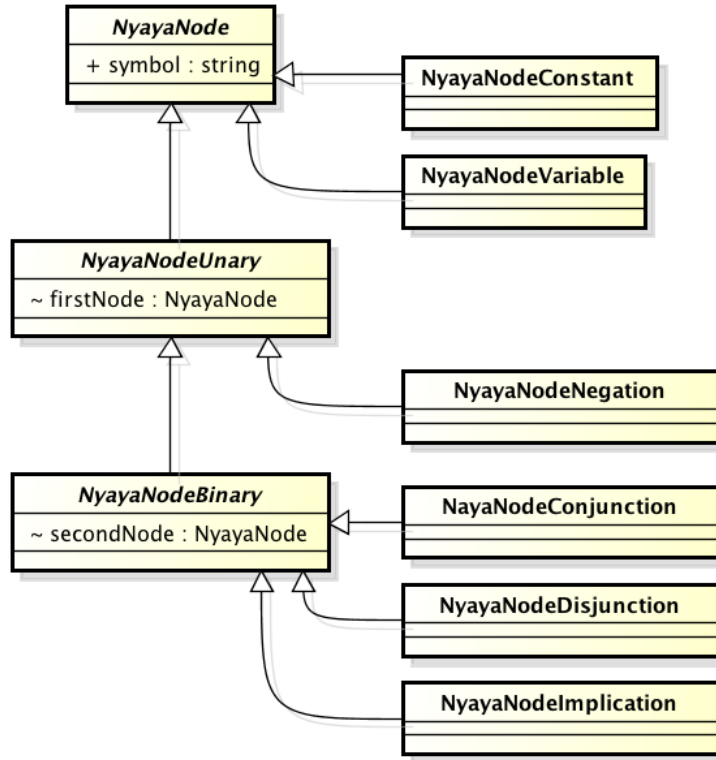

Figure 5.2.: Class hierarchy with abstract and concrete classes

Node Class Categories

Actually the creational methods were implemented as one node class category in a separate source file. Class categories are a language feature of Objective-C [10, p.225ff] – an adaptation of the decorator design pattern [8, p.175ff] – to extend or organize the functionality of given classes beyond their core purpose.

There are base implementations of category methods in the root node class, that will be overridden in subclasses. For example `isLiteral:bool` returns false for all node instances except for instances of variables and instances of negations of variables.

The variable node class adds setters for the categories `Valuation` (see Figure 5.4 on page 21) and `Display`.

"Attribute" provides information about the number of sub-nodes of a node and the normal forms the sub-tree matches.

"Creation" (see Figure 5.3) implements the production rules of the grammar (see Table 5.2 on page 23) to create syntax trees recursively . Since the constructors of all classes in the cluster are hidden, the production of invalid syntax trees is prevented.
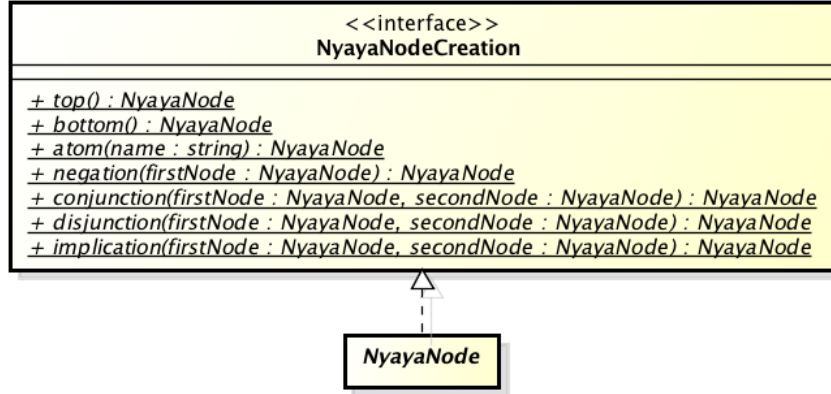

Figure 5.3.: Production rules

"Derivation" defines methods to derive semantically equivalent normal forms from a given syntax tree. These methods are used to create implication free forms, negation normal forms, conjunctive and disjunctive normal forms for Nyāya's BoolTool.

"Description" provides methods to convert a syntax tree into one of it's string representations – either a propositional formula in strict syntax with many parentheses or a formula using precedences and associativity. In most cases, the second form leads to shorter strings with less parentheses. In rare case, only the outer parentheses are omitted.

"Display" (see Figure 5.8 on page 27) allows extended valuations with incomplete truth assignments, i.e the value of an atom can be undefined. It is used for the interactive syntax tree view on the playground, where the user can assign truth values to no atom, some atoms or all atoms.

"Random" creates arbitrary syntax trees with a given set of connectives and atoms within a given range for the number of nodes. It is used to create formulas for the exercises.

"Transformation" provides equivalence transformations and methods to create semantically different syntax trees by replacing atoms, connectives or sub-trees with atoms, connectives or trees.

"Type" provides a single method, that returns an enum type of a node, i.e constant, variable, type of connective. This method is used to decide, which equivalence transformations are available.

"Valuation" (see Figure 5.4) provides fast valuations with complete truth assignments. It is used to create truth tables and binary decision diagrams.
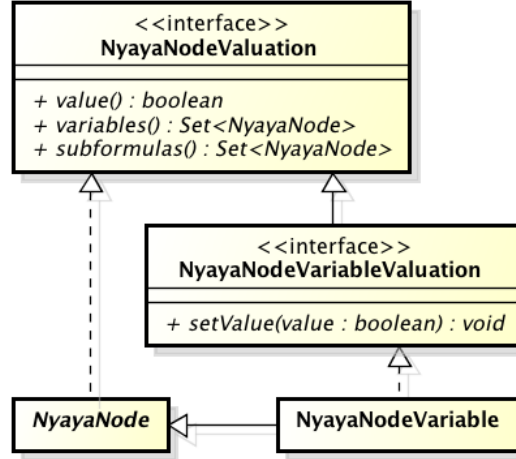

Figure 5.4.: Valuation of trees

Immutable syntax trees

Instances of node classes are not mutable regarding their function as nodes of abstract syntax trees. This means nodes can be created, but not modified. The symbol of an atom can not be changed, a connective node can not be altered into another one, and child nodes of a connective node can not be replaced or reordered.

Derivations or transformations do not change the structure of an existing syntax tree, but rather create a new independent syntax tree.

Obviously mutable truth assignments do not affect the immutable structure of a syntax tree, because truth assignments are semantics, not syntax.

Acyclic syntax graphs

Since the representation of abstract syntax trees is implemented immutable, there is no need to create multiple instances for syntactically equivalent sub-trees. At run-time syntax trees are represented by acyclic syntax graphs, where multiple sub-trees can be represented by the same instance-graph (Figure 5.5 on the next page).

This eases the detection of obvious tautologies $P \vee \neg P$, $P \rightarrow P$ and contradictions $P \wedge \neg P$, because the sub-tree P on the left side is represented by the same object-graph as the sub-tree P on the right side.
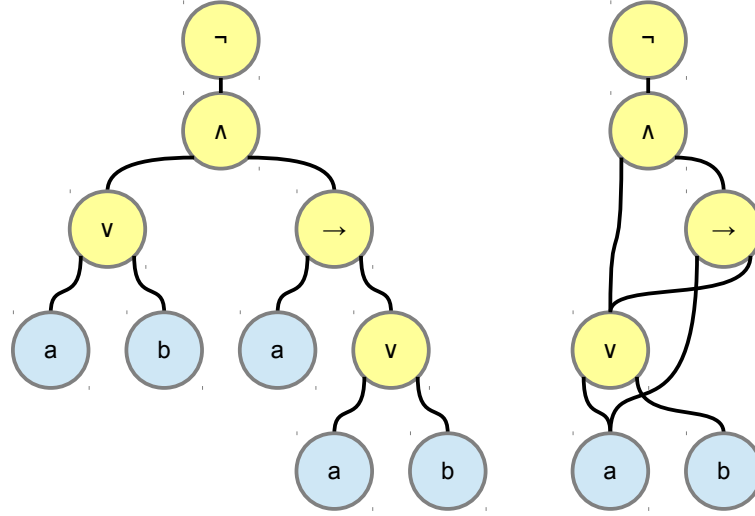
Figure 5.5.: Abstract syntax tree – acyclic graph
$$\neg((a \vee b) \wedge (a \rightarrow a \vee b))$$

Limitations

The derivation of normal forms will fail on some relatively small syntax trees. especially when the formula contains too many exclusive disjunctions.

With every exclusive disjunction of a formula P and an atom p, that is not already used in P, the number of nodes will at least double when deriving an equivalent implication free form. "Implication free" means also the absence of exclusive disjunctions.

$$
\begin{aligned}
\#(P \oplus p) &= 1 \cdot (\#(P) + 2) \\
\#((P \wedge \neg p) \vee (\neg P \wedge p)) &= 2 \cdot (\#(P) + 2) + 3 \\
\#((P \vee p) \wedge (\neg P \vee \neg p)) &= 2 \cdot (\#(P) + 2) + 3
\end{aligned}
$$

The syntax tree for an exclusive disjunction of 20 different atoms is build from 39 nodes. The syntax tree of a semantically equivalent implication free form will already contain over four million nodes. Deriving the negation normal form does not increase or decrease the number of nodes significantly, but the necessary distributions of disjunctions will multiply the number of nodes again.

To ensure the stability of Nyāya and since it is near impossible to present a propositional formula with thousands of symbols in a meaningful way to the user, the derivation of normal forms is aborted, when the results surpass a defined size.

5.2.2. Parser

The orginal backend of `BoolTool`[1] – parsing and transformation of Boolean functions – is implemented in `OCaml`[2], a functional programming language, which is well suited for this kind of task.

After several failed attempts to use the `OCaml` sources in the Xcode project, i.e to cross-compile the sources to iPad's processor architecture (ARM), to add the object code to the project's build configuration correctly to bridge and call `BoolTool`'s functions from C, to pass data to these functions and to read return data from these functions calls, the implementation of a simple translator for propositional formulas (a subset of all possible strings) into syntax trees looked more promising and was successful. `BoolTool`'s transformations were implemented as node class category (see Section 5.2.1 on page 20).

The generation of a syntax tree from a formula is carried out in two steps.

Scanning

In the first step the input string is transformed into an array of strings – the list of input tokens – using the standard regular expression class of Cocoa Foundation with a suitable expression (see Table 5.1).

$$\top \,|\, \bot \,|\, \neg \,|\, ! \,|\, \wedge \,|\, \& \,|\, . \,|\, \vee \,|\, \backslash / \,|\, \backslash + \,|\, \underline{\vee} \,|\, \oplus \,|\, \hat{} \,|\, = \,|\, <> \,|\, \leftrightarrow \,|\, > \,|\, \rightarrow \,|\, \models \,|\, (|)|, |; |\backslash w+$$

Table 5.1.: Regular expression for the scanner

Parsing

In the second step the list of input tokens is parsed top-down using a recursive-descent parsing algorithm [11, p.144ff] following an EBNF grammar outlined in Table 5.2, that defines precedence and associativity of connectives too.

```
1 formula      ::= entailment
2 entailment   ::= sequence [ '⊨' entailment]
3 sequence     ::= bicondition { ';' bicondition }
4 bicondition  ::= implication [ '↔' bicondition ]
5 implication  ::= xdisjunction [ '→' implication ]
6 xdisjunction ::= disjunction { '⊻' disjunction }
7 disjunction  ::= conjunction { '∨' conjunction }
8 conjunction  ::= negation { '∧' negation }
9 negation     ::= '¬' negation | '(' formula ')' | identifier
```

Table 5.2.: Core EBNF grammar for the parser

---

[1] cl-informatik.uibk.ac.at/software/BoolTool/
[2] ocaml.org

Limitations

Nyāya's BoolTool does not parse as fast and (memory) efficient as `BoolTool`'s parser, primarily because recursion and token probing are more expensive operations in Objective-C than in `OCaml`. The creation of a syntax graph instead of a syntax tree adds more overhead. But Nyāya's implementation is *good enough*, because *real world* user input is limited to at most a few thousands characters.

The unit tests have shown that there are no memory or run-time issues, although the parsing can take a second or two on an iPad.

Enhancements

Despite the drawbacks mentioned above the benefits far outweigh the disadvantages of implementing its own parser.

- Nyāya parses strings in UTF8-encoding, therefore identifiers are not limited to Latin letters and $\alpha + \omega$ will be parsed correctly.

- The grammar rules for operators (see Table 5.3) are defined as sets of tokens, which are initialized at application start.

```
7 disjunction  ::= conjunction { OR conjunction }

15 OR           ::= '∨' | '|' | '+' | 'O'
```

Table 5.3.: Excerpts from a localized grammar (Italian)

- The symbols for operators are localized using the standard localization framework of Cocoa, which will return the localized string from the appropriate text file (see Table 5.4).

```
IMP = "→ > IMPLIES";        IMP = "→ > IMPLICA";
OR  = "∨ | + OR";           OR  = "∨ | + O";
AND = "∧ & . AND";          AND = "∧ & . E";
NEG = "¬ ! ~ NOT";          NEG = "¬ ! ~ NON";
```

Table 5.4.: Localizable.strings in en.lproj and it.lproj

- The parser generates syntax graphs with Objective-C run-time objects, which are well suited for the use cases in an interactive environment.

- Extensions can be added to the parser at any time, without the risk of breaking anything, because the correctness of the parser is substantiated with unit tests in the project itself.

5.2.3. Truth Tables

Truth tables are either small enough to be computed on demand or too big to be stored in memory. Either way there is no reason to store filled truth tables at run-time.

5.2.4. Binary Decision Diagrams

Binary decision trees and diagrams are represented by trees or acyclic directed graphs built from instances of a simple node class with attributes for a name, a left branch and a right branch. The name is either an identifier, i.e. the name

BddNode
+ name : string + leftBranch : BddNode + rightBranch : BddNode
+ obdd(ast : NyayaNode, order : Array<NyayaNode>, reduce : boolean) : BddNode

Table 5.5.: Attributes and factory method of BddNode

of a variable, "0" or "1". Nodes with the name "0" or "1" are leaf nodes or result nodes and must not have a left or right branch. The other nodes with the name of a variable are decision nodes and must have valid left and right branches.

The class provides a factory method to create (un)reduced ordered binary decision diagrams from abstract syntax trees (Figure 5.5).

5.3. Views and Controllers

Nyāya's main navigation – switching between Welcome, Tutorials, Playground, Glossary and Nyāya's BoolTool – is organized by a sub-class of `UITabBarController` (an heir of `UIViewController`). At start the tab bar controller will be instantiated and invoked by Nyāya's application delegate. The tab bar controller will present it's `UITabBar` (an heir of `UIView`) and will appoint of one of its view controllers, which will present its view such as the welcome screen.

Tabbing an `UITabBarItem` will cause the tab bar to delegate the event to its `UITabBarDelegate` – the tab bar controller. The tab bar controller will switch the selected view controller, which then will present its content.

Except for the welcome view controller all content view controllers inherits from split view controller, which provides a master view controller and a detail view controller. The tutorial master view controller and the Nyāya's BoolTool master view controller shares a list of persisted formulas.

5.3.1. Welcome

The welcome view controller is a simple subclass of `UIViewController` and presents its content in a not so simple `UIWebView`, which is a fully functional, but 'naked' web-browser, i.e. links are followed and there is a page history, but no back-button is presented. The behavior of the web view can be altered by implementing a web view delegate – usually the controller.

5.3.2. Tutorials

The tutorials master view controller presents a table view with five sections of tutorials. The sections organize the tutorial titles for introduction, syntax, semantics, normal forms and binary decision diagrams. Tabbing an entry will cause the detail view controller to present the corresponding tutorial in a web view with an additional exercise button on the top of the view.

Tabbing this exercise button will put the exercise view controller in charge. A suitable overlay view will be opened, and an instance of a matched class implementing of `NyTuTester` will be created.
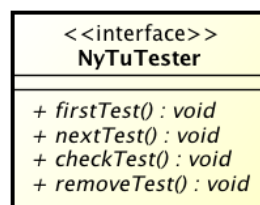

Figure 5.6.: Tester interface

The method `firstTest` initializes the test parameters and fills the test view with instructions and labels. Then it calls `nextTest`, which will fill the test

view with a new question with every call. The method `checkTest` evaluates the user's answer and displays the result in the test view. When the test view is closed by the user `removeTest` will dispose resources attached to the test view.

5.3.3. Playground

The playground master view controller presents a table view with the list of stored and selectable formulas. The user can use a formula from this list to add a tree view to the detail view – the canvas view. Or the user can add a new tree view to the canvas view using a tap and hold gesture. The canvas view can hold multiple syntax trees.
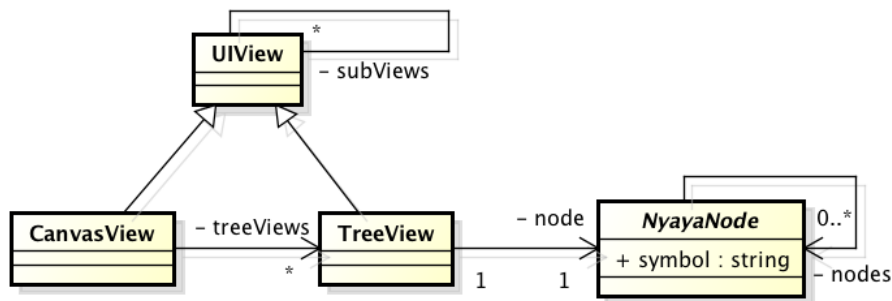
Figure 5.7.: Canvas view with multiple syntax tree views

The nodes of the syntax tree view are drawn as circles around the `symbol()` method defined in the category `Display` of the class `NyayaNode`. The color of the border is determined by the value of `displayValue()`.
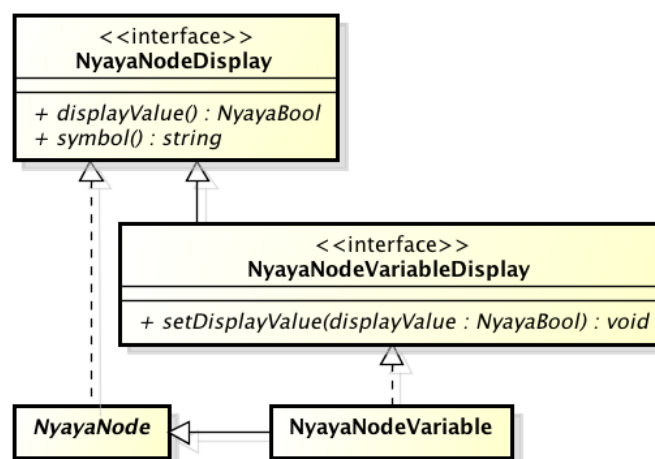

Figure 5.8.: Node display interface

5.3.4. Glossary

The glossary master view controller reads all elements with an id from the glossary content document, creates an alphabetically ordered list of technical terms using the texts included between the corresponding element-tags. This list of technical terms defined by the glossary document is presented in a table view. Tapping a term will scroll the glossary document view – a web view - to the chosen term, controlled by the glossary detail view controller.

5.3.5. BoolTool

The master view controller of Nyāya's BoolTool presents a table view with stored and selectable formulas. An editable text field with an attached customized keyboard will allow the user to enter their formulas.

The output will be presented in different views. Text views, which are configured to be read only, are used to present normal forms, a web form is used for the truth table and a customized view will draw the binary decision diagram.

5.4. Application content

Beside the classes to determines the run-time behavior of the application, the storage of application content must be defined.

5.4.1. Content on delivery

Configuration data, localization data and html documents are stored in UTF-8 encoded text files. All graphical content such as icons and diagrams are stored in portable network graphics. These content files are embedded in Nyāya's app bundle and will be downloaded along the application execution data, while receiving Nyāya for iPad from the App Store.

5.4.2. User created content

Formulas from the playground and Nyāya's BoolTool are persisted in the simple property list file `BoolToolData` in the documents folder of the app.

5.4.3. Export and import of content

Single formulas can be exported from or imported into editable text fields by simply using the standard copy and paste mechanism of iOS.

6. Retrospective

The first version of Nyāya is a fully functional eLearning App, which covers its purpose – an introduction to propositional logic - very well. The entire content of the application is stored in HTML-files in UTF-encoding. So the tutorials and the glossary are usable on any platform.

But Nyāya is still far from perfect. Some features have to be re-engineered before additional features can be implemented.

6.1. Important improvements

Although some technical aspects of Nyāya are working correctly, they need a revision, because they are too inflexible or too slow.

- At this time content updates can only be done with an application update in the App Store, which will lead to some delay in the distribution of the correction of typing errors or an update of the glossary.

- Syntax trees are stored memory optimized as acyclic syntax diagrams. Nevertheless the valuation does not use this fact and the same sub-tree will be evaluated multiple times.

- The creation of binary decision diagrams is very slow for formulas with more than 15 different atoms.

- Individual formulas can be imported and exported with the clipboard-mechanism of iOS. But this should also be possible with all persisted formulas at once.

Some aspects of Nyāya 's user interface does not use the full richness of Cocoa touch. Others lack the graphical sophistication that users are accustomed to on the iOS platform.

- Accessibility features of iOS have to be supported – especially `VoiceOver`, `Zoom`, `LargeText` and `AssistiveTouch`, which will enable Nyāya to reach a wider audience.

- Although English is the technical language of computer science, localizations of the user interface, the tutorials and the glossary will increase the reachable audience.

- Content and user interface should be presented in a visually more appealing manner. Firstly css-definitions should be developed, secondly additional images have to be created.

6.2. Outlook

To increase the number of potential users Nyāya could be extended with new features or Nyāya could be ported to other platforms.

6.2.1. Additional features

Even though it is unlikely the following features will be implemented in the foreseeable future, the following use cases would be obvious evolutions for Nyāya.

- Creation and editing of arbitrary binary decision diagrams similar to abstract syntax trees.

- Definition of arbitrary Boolean functions with truth tables, propositional formulas, abstract syntax trees or binary decision diagrams.

- Syntax trees, truth tables and binary decision diagrams should be exportable too.

6.2.2. Additional platforms

The author of Nyāya is an apologist of native client applications, although he admits that there are many useful use cases for modern web-applications, using html5, css3 and JavaScript. Unfortunately the freely available development tool chains for (graphical) html/javascript-applications are very limited and far from consolidated, although the underlining standards of html5, css3 and JavaScript 1.8 would provide everything needed. Still the workload to create sophisticated web-apps is much higher than to develop native apps on the same level.

So the most likely platforms to supported Nyāya are Mac OS X and Android.

- For a Mac OS X version some aspects of the user interface must be rethought and re-implemented, because Mac OS X does not support a touch interface.

- For an Android version the model must be translated into Java and the user interface must be re-implemented.

7. Conclusion and Summary

Besides the work as a .NET developer for administrative tools of a payment system, the software project Nyāya for iPad was a welcome diversion. Coping with an entirely different platform and mastering the memory restrictions and the very limited cpu speed of an original iPad (in comparsion to a modern pc) were exciting tasks.

List of Figures

List of Tables

Bibliography

[1] *Git.* http://git-scm.com, Nov 2012.

[2] *GitHub.* http://www.github.com, Nov 2012.

[3] *Nyaya.* http://en.wikipedia.org/wiki/Nyaya, Feb 2013.

[4] *Nyaya.* http://www.iep.utm.edu/nyaya/, Feb 2013.

[5] E. M. Buck and D. A. Yacktman. *Cocoa Design Patterns.* Addison-Wesley Professional, 1st edition, 2009.

[6] S. Chacon. *Pro Git.* Apress, Berkely, CA, USA, 1st edition, 2009.

[7] J. Dunlosky, K. A. Rawson, E. J. Marsh, M. J. Nathan, and D. T. Willingham. Improving students' learning with effective learning techniques: Promising directions from cognitive and educational psychology. *Psychological Science in the Public Interest*, 14(1):4–58, 2013.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, New York, NY, USA, 2004.

[10] S. Kochan. *Programming in Objective-C 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[11] K. C. Louden. *Compiler Construction: Principles and Practice.* PWS Publishing Co., Boston, MA, USA, 1997.

[12] A. Middeldorp. Logic (lecture). http://cl-informatik.uibk.ac.at/teaching/ws11/lics/content.php, 2011/2012.

A. Symbols

Name	Atom	Negation NOT	Conjunction AND	Disjunction OR	Implication
ASCII	p,q,r	!	& .	| +	>
Symbol		¬	∧	∨	→
UTF-8		C2 AC	E2 88 A7	E2 88 A8	E2 86 92
Unicode		U+00AC	U+2227	U+2228	U+2192
HTML		¬	∧	∨	→
LaTeX		\neg	\wedge	\vee	\rightarrow

Table A.1.: Basic symbols in different representations

Tautology TOP	Contradiction BOTTOM	Exclusive Disjunction		Biconditional IFF
T 1	F 0	EOR	XOR	<>
⊤	⊥	$\underline{\vee}$	⊕	↔
E2 8A A4	E2 8A A5	E2 8A BB	E2 8A 95	E2 86 94
U+22A4	U+22A5	U+22BB	U+2295	U+2194
	⊥		⊕	↔
\top	\bot	\veebar	\oplus	\leftrightarrow

Table A.2.: Additional symbols

B. Definitions

```
1  <!ENTITY % plistObject "(array␣|␣data␣|␣date␣|␣dict␣|␣real␣|␣
       integer␣|␣string␣|␣true␣|␣false␣)" >
2  <!ELEMENT plist %plistObject;>
3  <!ATTLIST plist version CDATA "1.0" >
4
5  <!-- Collections -->
6  <!ELEMENT array (%plistObject;)*>
7  <!ELEMENT dict (key, %plistObject;)*>
8  <!ELEMENT key (#PCDATA)>
9
10 <!--- Primitive types -->
11 <!ELEMENT string (#PCDATA)>
12 <!ELEMENT data (#PCDATA)>
13     <!-- Contents interpreted as Base-64 encoded -->
14 <!ELEMENT date (#PCDATA)>
15     <!-- Contents should conform to a subset of ISO 8601 (in
           particular, YYYY '-' MM '-' DD 'T' HH ':' MM ':' SS 'Z'.
           Smaller units may be omitted with a loss of precision) -->
16
17 <!-- Numerical primitives -->
18 <!ELEMENT true EMPTY> <!-- Boolean constant true -->
19 <!ELEMENT false EMPTY> <!-- Boolean constant false -->
20 <!ELEMENT real (#PCDATA)>
21     <!-- Contents should represent a floating point number
           matching ("+" | "-")? d+ ("."d*)? ("E" ("+" | "-") d+)?
           where d is a digit 0-9. -->
22 <!ELEMENT integer (#PCDATA)>
23     <!-- Contents should represent a (possibly signed) integer
           number in base 10 -->
```

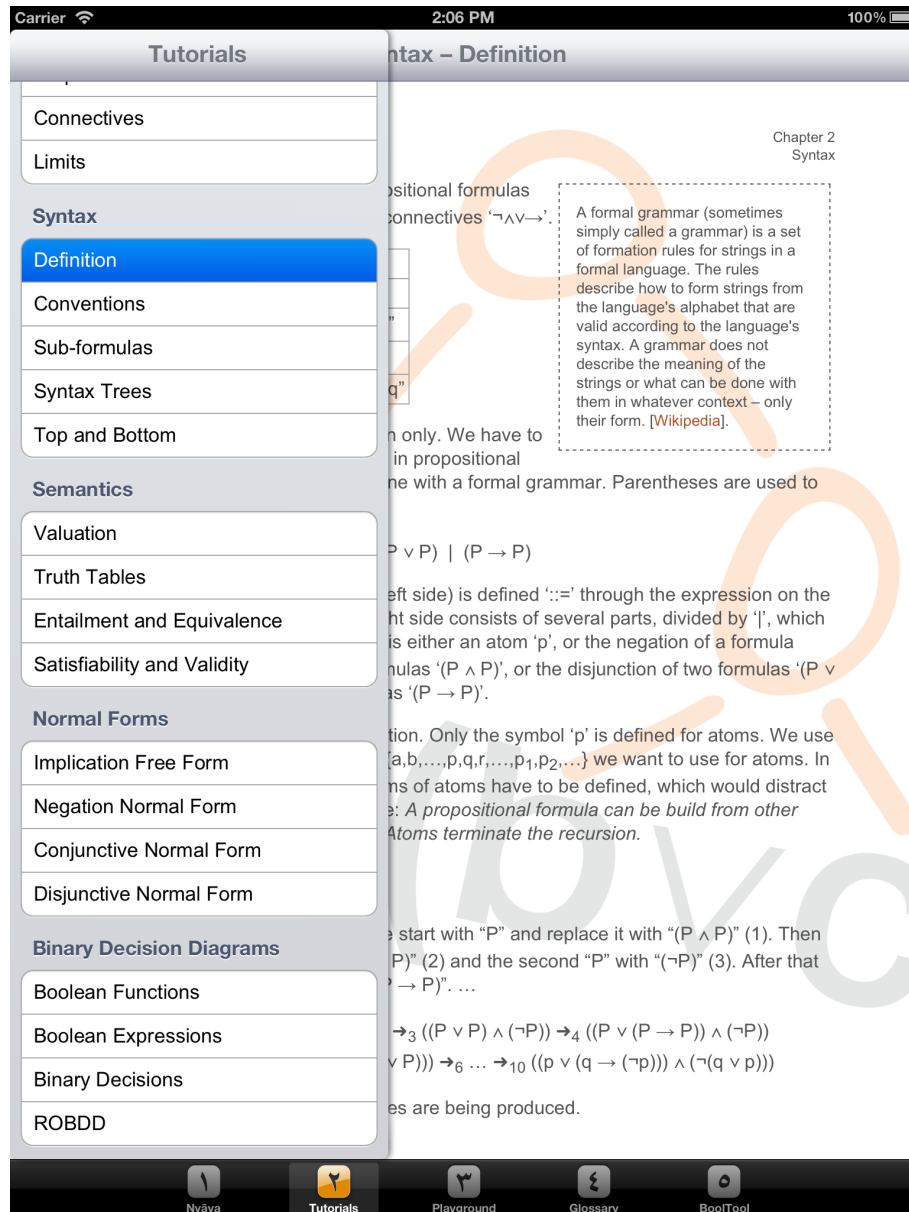Table B.1.: www.apple.com/DTDs/PropertyList-1.0.dtd
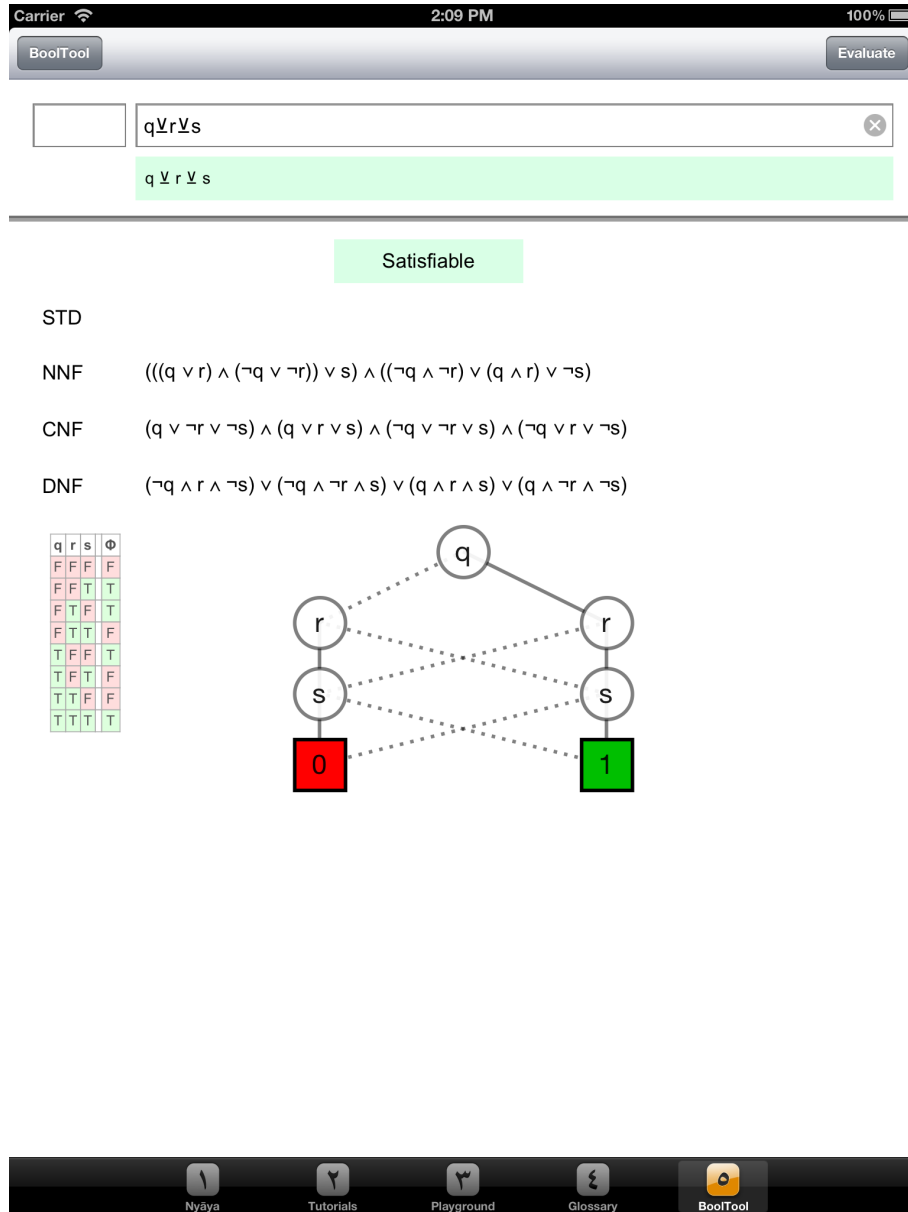
C. Screenshots

Figure C.1.: Tutorial

Figure C.2.: Playground

Figure C.3.: BoolTool