University of Malta

Faculty of Information and Communications Technology

Department of Intelligent Computer Systems

A Controlled Natural Language Interface for Electronic Contracts

John J. Camilleri

Final Year Project Report
23rd May 2010

Supervisors

Michael Rosner
Gordon J. Pace

Submitted in partial fulfilment of the requirements for the degree of B.Sc. I.T. (Hons.)

A Controlled Natural Language Interface for Electronic Contracts

John J. Camilleri

Supervised by Michael Rosner and Gordon J. Pace
Submitted in partial fulfilment of the requirements for the degree of B.Sc. I.T. (Hons.)
University of Malta
23rd May 2010

This document was typeset in Palatino using $\LaTeX\,2_\varepsilon$
Graphs plotted using MATLAB 2009b

for Claudia

Abstract

A contract is an agreement between two or more parties, defining the obligations of each and the consequences of violating them. Contracts often have ambiguous or even contradictory interpretations, motivating the study of formal contract representations in order to make them amenable to automated processing. Human interaction with electronic contracts generally requires the use of a natural language interface, for which controlled natural languages (CNLs) are frequently used.

We took the game of Nomic—a self-amending game based on government systems, where turns are played by changing the rules—as a case study for this project. Nomic combines contract-type game rules with the need for natural language interaction, thus providing ideal scope for investigating the design of a contract logic and CNL interface for the game.

We defined our own version of Nomic called *BanaNomic*, set in a rainforest where players are monkeys in a tree, competing to pick bananas and able to manipulate the rules of the forest to their advantage. A suitable contract logic was designed, and a corresponding contract evaluator implemented in Haskell. A CNL interface was built using the Grammatical Framework (GF), using its incremental parser to build guided input methods for simplifying the writing of CNL phrases.

Two games of *BanaNomic* were played by 14 players over nine days, and qualitative user evaluations were obtained through in-game feedback and a post-game questionnaire. The results of the evaluation indicated a positive response to the language used in the game, for both the automatically generated phrases and the guided input methods. The contract logic was found to be a little restrictive, but on the whole provided for a well-balanced game and allowed for a variety of interesting and devious rules to be created.

Despite some implementational flaws with the contract evaluator, the overall approach employed was well justified by the positive results obtained. The need for human language interfaces to electronic contracts is an important need, and this project has served to demonstrate the success of one such approach.

Acknowledgements

The author would like to thank the following people:

- Both my supervisors for all their guidance and feedback, and keeping me on my [mental] toes.
- The helpful people at the Grammatical Framework developers group[1], in particular Krasimir Angelov and Aarne Ranta.
- Joel Uckelman of `http://nomic.net` for pointing me to some useful Nomic literature.
- Stephen Blackheath for his help with the Hexpat[2] Haskell package.
- All the evaluators who volunteered a few minutes of every day to using the system and providing feedback.
- Nicola for her meticulous orthographical eye.

[1] `http://groups.google.com/group/gf-dev`

[2] `http://hackage.haskell.org/package/hexpat`

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Natural and formal languages . . . 1

1.1.2 Contracts and ambiguities . . . 1

1.2 Motivation . . . 2

1.2.1 Electronic contracts and their use . . . 2

1.2.2 The language problem . . . 3

1.2.3 Nomic . . . 4

1.3 Project aims . . . 5

1.4 Report overview . . . 5

2 Contract logic representations **6**

2.1 What is a contract? . . . 6

2.2 Deontic logic . . . 6

2.2.1 OPP-logic . . . 7

2.2.2 Paradoxes . . . 7

2.3 Issues . . . 8

2.3.1 Action-based vs. state-based . . . 8

2.3.2 Reparation clauses . . . 9

2.3.3 Internal and external choice . . . 9

2.3.4 Temporal aspects . . . 10

2.4 Comparison of works . . . 10

2.5 Chapter summary . . . 12

3 Controlled natural languages **13**

3.1 Processing natural languages . . . 13

3.1.1 The template approach . . . 14

3.2 Controlled natural languages . . . 15

3.2.1 Examples of CNLs . . . 16

3.3 Unification-based grammar formalisms . . . 18

3.4 The Grammatical Framework 19
3.4.1 Abstract & concrete syntaxes 19
3.4.2 Linearisation-centric records 21
3.4.3 Input possibilities 21
3.4.4 Resource Grammar Library 22
3.4.5 Projects using GF 22
3.4.6 GF and CNLs 23
3.5 Chapter summary 23

4 BanaNomic 24
4.1 The game of Nomic 24
4.1.1 Game objectives 25
4.1.2 Flavours of Nomic 25
4.1.3 Attempts at formalising Nomic 26
4.1.4 PerlNomic 26
4.2 BanaNomic 27
4.2.1 The setting 28
4.2.2 Rules of the rainforest 28
4.3 Chapter summary 29

5 A contract logic for Nomic 30
5.1 Formal grammar 30
5.2 Contract examples 33
5.3 Semantics 34
5.3.1 Deontic inference and precedence 34
5.3.2 Choice 34
5.3.3 Permanent contracts 35
5.3.4 Timely contracts 35
5.3.5 The generic player 36
5.3.6 Rule manipulation 36
5.4 Comparison with other works 37
5.5 Chapter summary 38

6 The language of BanaNomic 39
6.1 Designing the CNL 39
6.1.1 Template-based linearisations 39
6.1.2 Semantic disambiguation 40
6.1.3 Punctuation 40
6.1.4 Problem areas 41
6.2 Grammar design in GF 42

6.2.1 Abstract syntax 43
6.2.2 Concrete English syntax 43
6.3 Chapter summary 44

7 Implementation 45
7.1 System overview 45
7.2 Client-side web application 46
7.2.1 Google Web Toolkit 46
7.2.2 Grammatical Framework 46
7.3 Server-side backend & game evaluator 48
7.3.1 Java backend 48
7.3.2 Haskell game evaluator 49
7.4 Implementation notes 50
7.4.1 Hosting setup 50
7.4.2 Web applications & user interface 50
7.5 Development issues & unforeseen problems 51
7.5.1 Logic 51
7.5.2 Language 53
7.5.3 User interface 55
7.6 Chapter summary 55

8 Evaluation 56
8.1 Methodology 56
8.1.1 What answers are we seeking? 56
8.1.2 The questions we asked and how 56
8.1.3 Evaluation period 57
8.2 In-game feedback 58
8.2.1 Average overall ratings 58
8.2.2 User ratings over time 58
8.2.3 Enactment of new rules 58
8.2.4 Complete turns and passes 60
8.3 Post-game questionnaire 60
8.4 Discussion 62
8.4.1 The contract logic and the self-amending game 62
8.4.2 Language aspect 64
8.4.3 Overall reflections on the game 65
8.4.4 Limitations with the methodology 66
8.5 Chapter summary 67

9 Conclusions 68
9.1 Reflections on the project . . . 68
9.1.1 Project overview . . . 68
9.1.2 Summary of results . . . 69
9.2 Points of limitation . . . 70
9.2.1 Natural language . . . 70
9.2.2 Contract logic . . . 71
9.2.3 Nomic . . . 72
9.2.4 Evaluation . . . 72
9.2.5 Other observations . . . 73
9.3 Future work . . . 74
9.3.1 Unrealised ideas . . . 74
9.3.2 Ultimate goals . . . 74
9.4 Closing remarks . . . 75

A User guide 77
A.1 The basics . . . 77
A.1.1 What's this game all about? . . . 77
A.1.2 The setting . . . 77
A.1.3 Playing your turn . . . 78
A.1.4 Composing rules . . . 79
A.1.5 Feedback . . . 79
A.1.6 The initial rules . . . 80
A.1.7 Some gameplay tips . . . 81
A.2 Other issues . . . 81
A.2.1 What's b'clock? . . . 81
A.2.2 How actions are processed . . . 81
A.2.3 How rules are processed . . . 81
A.2.4 Permission, obligation and prohibition . . . 82
A.2.5 Sometimes & always . . . 82
A.2.6 Fixed laws . . . 82
A.3 Technical issues . . . 83
A.3.1 Browser compatibilities . . . 83
A.3.2 Bugs and glitches . . . 83

B Game transcripts 84
B.1 Initial rules . . . 84
B.2 Turn histories . . . 85
B.2.1 Banana Bonanza . . . 85
B.2.2 Potassium Paradise . . . 88

B.3 Final summaries 91
B.3.1 Banana Bonanza 91
B.3.2 Potassium Paradise 92
B.4 Interesting behaviour 93

C Evaluation forms & results 96
C.1 Evaluators 96
C.2 In-game feedback 97
C.2.1 Feedback form 97
C.2.2 Feedback responses 97
C.3 Post-game questionnaire 102
C.3.1 Questionnaire form 102
C.3.2 Questionnaire responses 103

D Selected code fragments 105
D.1 Game evaluator 105
D.1.1 Contract logic grammar 105
D.1.2 XML gamestate representation 107
D.2 Grammatical Framework 113
D.2.1 Abstract contract grammar 113
D.2.2 `complete()` JavaScript extension 114

Glossary of terms 118

Bibliography 121

List of Tables

3.1 Examples of CNL phrases and their natural equivalents 15
3.2 Examples of phrases in ACE 17

5.1 Time limits for *sometimes* clauses 35
5.2 Time limits for *always* clauses 36
5.3 Example showing the abolition of an obligational clause 37

6.1 Examples of straightforward canned text linearisations 40
6.2 Contract logic examples and their equivalents in GF 44
6.3 Different possible linearisations of a single lexical function 44

8.1 Available rating levels 57

B.1 Initial game rules 84
B.2 Transcript for *Banana Bonanza* 85
B.3 Transcript for *Potassium Paradise* 88
B.4 Final rules for *Banana Bonanza* 91
B.5 Final player standings for *Banana Bonanza* 92
B.6 Final rules for *Potassium Paradise* 92
B.7 Final player standings for *Potassium Paradise* 93

C.1 List of evaluators 96
C.2 In-game feedback evaluation data 97
C.3 Post-game questionnaire responses 103

List of Figures

3.1 Vauquois Triangle for machine translation and CNL triangle 14
3.2 The GF drop-down suggestions input method 22

7.1 System block diagram 46

8.1 Overall in-game feedback ratings 59
8.2 Average user ratings per day 59
8.3 Enactment of new rules and input methods used 60
8.4 Complete turns and passes played 61
8.5 Summary of post-game questionnaire responses 61
8.6 Post-game questionnaire responses about input methods 62

A.1 "Play Turn" panel 78
A.2 Suggest panel input method 79
A.3 Fridge magnets input method 79
A.4 In-game feedback dialog 80

Chapter 1

Introduction

1.1 Background

1.1.1 Natural and formal languages

Communication is one of the most fundamental aspects of human life, one which we have evolved to be particularly good at thanks to the development of language. There are thousands of spoken, written and even signed languages which humans use to communicate, but one feature which they share is that practically all human languages evolve *naturally*. Natural languages are the direct products of the people who use them, growing and adapting continuously and organically.

Yet, not all languages are formed this way. For all the expressivity of natural languages, there are many situations in which they are just too imprecise and ambiguous for the task at hand. The areas of mathematics and computer science in particular require means of expression which are not just precise and free from ambiguity, but which must also be amenable to automated reasoning. In such cases, domain-specific language grammars are purposely constructed to meet these needs. Such languages are known as *formal* languages, and include all computer programming languages, such as C or Java, as well as the languages of mathematical logic such as predicate calculus.

Formal languages are often used to represent real-life phenomena as theoretical models, which can be reasoned upon and simulated in virtual scenarios. This project will look at the formal representation of natural language contracts, and the issues involved in translating between the two.

1.1.2 Contracts and ambiguities

A *contract* is some form of agreement between two or more parties, defining what is expected of each and the implications of failing to satisfy that agreement. Many things are contracts: ISP service level agreements, the promise to meet your friend for coffee, business partnerships, a marriage, software requirement specifications, governmental laws and international treaties. Though these examples all display different levels of importance, they are all based on the same basic notions of permission, obligation and prohibition.

Whether verbal or written, real-world contracts are generally expressed using natural languages. The more important a contract is, the more careful the wording used to describe it—resulting in a particular form of language often called *legalese*. Because of the important legal implications that contracts often represent, an entire profession exists whose sole job is the writing and understanding of these documents. Notwithstanding this, the fact that contracts are defined using natural language often leads to ambiguities and contradicting interpretations. It is common that opposing parties do not agree about what a particular clause in a contract implies, or whether or not a contract has been breached in a given situation. This often ends up leading to legal proceedings where rulings must ultimately be made by precedent, jury or judge.

1.2 Motivation

1.2.1 Electronic contracts and their use

The major problem with using natural languages to describe contracts is their inherent fallibility. Natural languages make it very hard to express *anything* with the absolute certainty that it may not be misconstrued—which is exactly what formal languages are designed to avoid. Provided a suitable grammar is used, by describing a contract in formal terms one would be able to completely avoid the ambiguities that normally arise out of natural language descriptions. Not only could contracts become completely unambiguous, but when represented electronically they could also be verified in an automated way. Violations to a contract would not need to be determined by lawyers or courtrooms, but could be checked quickly and with confidence using standard verification tools.

Detecting conflicts In addition to automated analysis of actions against an existing contract, the formal representation of contracts would also facilitate their checking for conflicts and contradictions. This is of particular use when combining contracts from various sources; for example in the ratification of an international treaty, where checks must be made to ensure that local law does not contradict that of the treaty. This ability becomes even more useful in dynamic settings such as service-oriented architectures, where the contracts requiring analysis are only available at runtime (see the next section for a typical example).

Service-oriented architectures

One of the most often-mentioned applications for electronic contracts is in service-oriented architectures (SOA), for example when applied to an ISP service level agreement (SLA). Such agreements typically describe the quality-of-service (QoS) properties which are to be provided by the system for the consumer, such as *the average bandwidth should be more than 20kb/s* and so forth (Prisacariu and Schneider, 2007).

Services such as internet provision are often composed of a number of sub-services from different providers, for example the telecommunications company who owns the physical network and an intermediary enabler. If each system in the hierarchy is governed by its own QoS contract, then in order for the ISP to provide their own SLA "package" they must be able to successfully combine each of these intermediary contracts, ensuring that they do not conflict with one another.

In addition, one can appreciate that if these QoS agreements were to have a dynamic nature to them, then the ISP's means of ensuring contractual compatibility would also need to be able to handle these on-the-fly changes in order to guarantee a consistent SLA contract with its customers.

Legal uses

Another potential application for electronic contracts is in legal settings, ranging from private contracts between individuals or corporations to state or even international legislation. Currently these areas tend to be the exclusive domain of lawyers, who are professionally trained to do this particular job. At the very least, electronic contract systems could become powerful tools for lawyers, allowing them to double-check the consistency of their work in an automated fashion.

Further to this, if backed by widely accepted standards then electronic contracts could potentially bring a new level of transparency to the legal process, by allowing non-professionals to analyse and verify contracts *themselves* without necessarily needing legal consultation. This is not to say that the role of lawyers could be done away with, but the benefits to users of such potential tools would be considerable.

1.2.2 The language problem

Despite their potential benefits, a major issue with formally defined contracts is the language barrier they create for humans working with them. While *natural* languages are by their nature easy for people to learn and use, the same cannot be said of formal languages. Rather than being human-centric, formal languages are specifically mathematics- or computer-oriented, and generally seem awkward and unnatural to the average person. Although mathematicians and programmers must simply learn to think and express themselves in a "formal" way, this cannot be expected of the public at large. At the end of the day, people communicate using *natural* languages—not formal ones—so a means of translating between a formal contract and a natural language representation of it would be essential.

Yet as already discussed above, natural languages bring with them a host of well-known problems such as ambiguity, syntactic complexity and context sensitivity (Pace and Rosner, 2010)—some of the very problems which motivated the use of electronic contracts in the first place. So how can we keep using formal contracts and all their associated benefits, yet at the same time use natural language representations for them without becoming vulnerable to their traditional problems?

Controlled natural languages

The route which shall be investigated in this project is the use of "stripped down" or *controlled* natural languages (CNLs), which try to strike a balance between being precise enough to facilitate their automated processing, yet at the same time natural enough to be easily usable and adequately expressive for the average person. Controlled languages are *designed* rather than evolved, and are produced by artificially restricting a subset of a natural language with a specific purpose already in mind (Pace and Rosner, 2010). CNLs are discussed in more detail in chapter 3.

1.2.3 Nomic

Nomic is a game of self-amendment loosely based on governmental systems (Suber, 1990). Starting with an initial rule set, each player takes their turn by changing the game's rules through a system of rule proposals and player voting. What makes Nomic so particular is that *everything* is theoretically up for amendment during the game, including the voting system itself and what players need to do to win. This produces a totally unique style of play, which has spawned hundreds of games being played by thousands of players worldwide for over 20 years (Phair and Bliss, 2005).

Despite the popularity of the game, only one Nomic variant could be found which uses automated rule processing, and was actually played directly in the Perl programming language (Phair and Bliss, 2005). The contract-style rules system found in Nomic and the potential for automated analysis were of particular relevance to this project, which when combined with the need for natural language interaction provided an ideal case study for this project. More details on Nomic can be found in chapter 4.

1.3 Project aims

Goals

The primary goal of this project was to explore the use of CNLs as a human interface for formal contract logics. Secondly, this project aimed to investigate the issues involved in the design and automated analysis of formal languages for electronic contracts. Using the game of Nomic as a case study, a suitable formal contract logic and corresponding natural language grammar were to be designed and incorporated into a playable version of the Nomic game. The effectiveness of this approach with respect to the natural expressivity of the game would then evaluated by a group of independent test users.

Objectives

1. Design a formal contract logic for representing the Nomic game and facilitating its automated verification.
2. Design a CNL interface for this logic which is naturally expressive yet amenable to parsing and generation.
3. Build a playable version of the Nomic game using the items described above.
4. Evaluate the suitability of the contract logic and the expressivity of the CNL qualitatively, by employing a group of users to play the game and comment on their experiences with it.

1.4 Report overview

Following is a brief outline of the structure of this report. Chapters 2 and 3 cover the background of contract logics and CNLs respectively. Chapter 4 looks at the game of Nomic, and goes on to outline the version of Nomic designed for this project, *BanaNomic*. Chapters 5 and 6 cover the design of the contract logic and natural language for *BanaNomic*. Chapter 7 takes a closer look at the implementation details of the system, while chapter 8 moves on to the method of evaluation and a discussion of the results obtained. Finally, the conclusion in chapter 9 looks at the overall success of the project, discussing its major limitations and ideas for future work.

The appendices section of this project includes a user guide to the *BanaNomic* game (appendix A), full transcripts of the games played along with some interesting observations (appendix B) and listings of all the feedback received during the evaluation period (appendix C). Appendix D contains selected code fragments from the project implementation.

Chapter 2

Contract logic representations

In this chapter we examine the various possibilities for formal contract representations. With a particular focus on the deontic notions of obligation, permission and prohibition we explore the issues arising from formal contract languages and how they are tackled by different authors.

2.1 What is a contract?

The concept of a contract can be realised in a number of different ways, from simple pre/post-conditions to quality-of-service (QoS) properties (Fenech et al., 2009c)—or in other words, the view of a contract as a set of conditions which a system must satisfy. This approach may seem like a natural one, however viewing a contract simply as a set of logical properties contains much implicit information and does not allow for the expression of exceptional cases, i.e. when contract clauses are violated. In addition, this approach may actually hide contractual conflicts altogether (Fenech et al., 2009b).

The approach usually favoured in most of the literature is to use deontic logic to enable the explicit automated reasoning of both normative and exceptional contract behaviour, as well as facilitate conflict detection and analysis (Fenech et al., 2009b). The rest of this chapter will deal mainly with this deontic approach and its various realisations.

2.2 Deontic logic

Deontic logic is concerned with the concepts of obligation, permission and prohibition, or the logic "of ideal . . . versus actual behaviour" (Meyer et al., 1994). Most of the literature in the area tends to agree that these deontic notions provide an ideal basis for formal contract languages. While in real-world contracts we often make distinctions between "rights", "permissions" and "authority" and between "must", "ought to" and "should", these distinctions are not necessarily required for analysis. (Pace and Schneider, 2009)

These deontic notions alone, however, are not sufficient to fully capture the diversity of real-world contracts and so are often supplemented with temporal, reparational, and other operators (Pace and Schneider, 2009). Numerous issues arise in the construction of these combined grammars, the most predominant of which are discussed in § 2.3.

2.2.1 OPP-logic

To avoid generalising the term "deontic logic", Pace and Schneider (2009) coin the phrase *OPP-logic* to refer to a formal language based on the deontic concepts of obligation, permission and prohibition (the "OPP" part) and allowing for the specification of reparation clauses (§ 2.3.2), temporal/causal aspects (§ 2.3.4), and more. OPP-logic is defined over actions, providing a number of operators over them (such as choice $+$, composition $\cdot$ and so forth).

2.2.2 Paradoxes

Starting from its formalisation into Standard Deontic Logic (SDL) (Pace and Schneider, 2009), the field of deontic logic has been plagued by a number of infamous paradoxes—expressions in deontic logic which are technically valid, yet seem counter-intuitive to standard reasoning. Some of the most well-known deontic paradoxes include (Meyer et al., 1994):

Ross' paradox Expressed formally as $O\varphi \rightarrow O(\varphi \vee \psi)$, this expression can be reached by using a simple disjunction-introduction. However when illustrated with an instance like

> *If one is obliged to mail a letter, then one is obliged to mail a letter or burn it.*

this quickly becomes paradoxical, and clearly undesirable.

No conflicting obligations From the laws of standard logic, the conjunction of something with it's own negative must itself be false, or in deontic terms: $\neg(O\varphi \wedge O\neg\varphi)$. In natural language we could express this as:

> *It is impossible to be obliged to go to work and also be obliged not to go work.*

However in real life, conflicting obligations are very much possible—just think of a union worker on strike.

Good Samaritan paradox This aptly-named paradox states that the implication of something which is obligated, is itself obligated. Formally defined as $\varphi \rightarrow \psi \vdash O\varphi \rightarrow O\psi$, this could easily be expressed as:

> *If George helps Ringo, this implies that Ringo is injured;*
> *Thus if it is obliged that George helps Ringo, then it is obliged that Ringo is injured.*

> In this case, what started out with good intentions clearly ended up with a less-than-desirable outcome.

Yet more paradoxes

Further to the deontic paradoxes described above, the introduction of temporal concepts into contract logics introduces yet more paradoxes (Pace and Schneider, 2009). Consider the following example adapted from Pace and Rosner (2010):

> The law of a country says that: *One is obliged to hand in Form A on Monday and Form B on Tuesday, unless officials stop you from doing so.*
> On Monday, Linda did not hand in Form A. On Tuesday she was arrested, and brought to justice on Wednesday.
> The police argue: 'The defendant was obliged to hand in Form A on Monday, which she did not. Hence she should be found guilty'.
> But Linda's lawyer argues back: 'To satisfy her obligation the defendant had to hand in Form B on Tuesday, but she was stopped from doing so by officials. Therefore she is innocent'.

Both of these arguments seem valid, yet they are obviously inconsistent with each other. In this case, the problem stems from different definitions of *when* the obligation becomes violated; is it violated as soon as the first action is not carried out, or when the relevant time period has elapsed?

Despite various attempts at handling these paradoxes through axiomatizations, the most successful approach is to simply restrict the contract syntax (Pace and Rosner, 2010; Fenech et al., 2008). Since often the eventual goal is to model real-world contracts which are expressible in natural language (and free from such paradoxes), this restriction is a justifiable solution.

2.3 Issues

2.3.1 Action-based vs. state-based

Contract languages can be defined in one of two ways; either expressing what the parties involved should or should not *do* (action-based), or what state of affairs should or should not exist. The two approaches are also known as "ought-to-do" and "ought-to-be" respectively, for example:

> *Paul is obliged to rake the leaves on Sunday.* — Action-based (ought-to-do)
> *On Sunday the leaves must be raked by Paul.* — State-based (ought-to-be)

Both approaches may be defended, and the choice of which to use ultimately depends on the domain in question (Pace and Schneider, 2009). In many contracts it may seem more natural to use ought-to-do clauses where the subject and their actions are explicitly stated; however in some settings like QoS contracts it makes more sense to use an ought-to-be approach, for example *the average bandwidth should be more than 20kb/s* (Prisacariu and Schneider, 2007).

2.3.2 Reparation clauses

In a contract where the obligations of different parties are specified, it is equally important to specify what reparations will follow if those obligations are not fulfilled. These are known as contrary-to-duty (CTD) and contrary-to-prohibition (CTP) clauses, and are commonplace in the types of contracts we encounter in everyday life, such as:

> *Shoppers are forbidden from taking items without paying for them;*
> *Violating this will result in prosecution.*

Note how such clauses are based on the concepts of causality and hence temporality. Some different approaches to handling reparation are discussed in § 2.4.

2.3.3 Internal and external choice

It is common for contracts to contain an element of choice between clauses, although in many cases it is often unclear *who* is allowed or expected to make that choice. Consider the following example:

> *Next spring, John will be obliged to either (i) tour with the band; or (ii) stay home with Yoko.*

This sounds like a reasonably straightforward clause, although what it fails to specify is *who* should make the choice between the two options. In this example, there are at least three possibilities: John, Yoko, or some other non-deterministic entity[1]. The former two would constitute *external* or *angelic* choice, with the latter being *internal* or *demonic* (note that here the terms internal/external are in reference to the contract, not the performer of the action). Clearly each possibility favours a different party, so a clearly defined system for determining *who* can ultimately make the decision must exist.

[1] Perhaps John's manager or the other band members should have a say too.

2.3.4 Temporal aspects

Another common feature of "practical" contracts is ability to write clauses which refer to time in either a relative or an absolute sense. Consider the following examples:

> *You are obliged to fill in the form and sign it.* (no temporal reference)
> *You are obliged to fill in the form, after which you are obliged to sign it.* (relative)
> *You are obliged to fill the form and sign it by Monday 13th.* (absolute)

Different approaches to the handling of temporal aspects are compared in § 2.4.

2.4 Comparison of works

In this section we mainly compare the works of the contract language $\mathcal{CL}$ as defined in Prisacariu and Schneider (2007), and the OPP-logic as used in Pace and Schneider (2009) and Pace and Rosner (2010).

Both works are designed with the same intention, that is the formal representation of contracts based on OPP notions and temporal aspects, while avoiding the major paradoxes often encountered in the area. $\mathcal{CL}$ and OPP-logic both choose to take the action-based "ought-to-do" approach.

Reparation

In the original definition of OPP-logic Pace and Schneider (2009) the authors propose the use of dedicated operators $CTD(\alpha, C)$ and $CTP(\alpha, C)$ which explicitly specify an obliged/prohibited action α and the contract C that must be satisfied if the former clause is violated.

The version proposed in Pace and Rosner (2010) takes a slightly different approach, by using an if-then-else type of notation $c_1 \triangleleft \alpha \triangleright c_2$, whereby if the action α is carried out then c_1 comes under affect; otherwise c_2 becomes applicable. This is known as the *conditional* operator.

The implementation of CTDs in $\mathcal{CL}$ borrows the $[\alpha]\,\phi$ syntax from Propositional Dynamic Logic (PDL) to represent that after performing action α, clause ϕ must hold. Following from this, CTDs are expressed using the $O_\varphi(\alpha)$ operator which equates to $O(\alpha) \wedge [\overline{\alpha}]\,\varphi$, or "$\alpha$ is obliged, and if α is not carried out then φ must hold" (Prisacariu and Schneider, 2007).

Choice

The issue of internal/external choice (§ 2.3.3) deals with *who* is allowed/expected to make the decisions defined in a contract.

As noted in Fenech et al. (2009a), the Communicating Sequential Processes (CSP) language has explicit operators for internal and external choice—$\sqcap$ and $[]$ respectively, eliminating the internal-external problem altogether.

On the other hand, $\mathcal{CL}$ makes no distinction between the two options (Prisacariu and Schneider, 2007), leaving it up to the formal semantics of the grammar to define which type of choice to apply.

Pace and Schneider (2009) suggest that internal and external choice be inferred by the position of the choice operator with respect to the deontic ones. Consider the difference in these two examples:

a. $\mathbb{O}(a+b)$
John is obliged to either (i) tour with the band; or (ii) stay home with Yoko.

b. $\mathbb{O}(a)+\mathbb{O}(b)$
John is either (i) obliged to tour with the band; or (ii) obliged to stay home with Yoko.

A seemingly fair way to interpret these contracts is to say that in the former, the obligation is already enacted and thus it is up to John to decide which option to choose (external), while in the latter case John is not obliged to choose, thus the choice lies with some other non-deterministic entity (internal). The authors however admit that this approach may at times be questionable, as being able to enact concurrent obligations as in contract (a) can allow for contractual anomalies.

Furthermore, this creates additional problems when used with prohibitions. Consider the clause:

$\mathbb{F}(a+b)$
George is forbidden from either (i) watching the game; or (ii) staying out all night.

It this case it becomes unclear how to interpret the contract; should George be forbidden from doing both actions (even individually), or should he only be prohibited from carrying them *both* out?

Time

When it comes to the inclusion of temporal aspects in contract logics, Fenech et al. (2009a) look at the use of Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) by testing them on a set of use cases from the Common Component Modelling Example (CoCoME). Their findings however indicate that they are both inadequate for encoding capturing the traditional deontic notions in an analysable way.

Furthermore, $\mathcal{CL}$ is already in-built with the temporal notions $\mathcal{U}$ (until), $\bigcirc$ (next), $\square$ (always) and $\Diamond$ (eventually)—as taken from temporal logic (TL) (Prisacariu and Schneider, 2007).

The OPP-logic grammar defined in Pace and Rosner (2010) goes a step further, with the incorporation of timed regular expressions within discrete domains.

Verifiability

Finally, we move onto the verifiability of formally encoded contracts. Using a set of CoCoME use cases as mentioned in the previous section, Fenech et al. (2009a) find that while LTL/CTL and CSP are amenable to formal analysis within their own right, major difficulties arise when using them to represent contracts. With the temporal logics (LTL/CTL) the issue lies with the encoding of the deontic notions themselves, while with the operational CSP there is no clear way to correctly represent CTD actions within the language. They conclude that for the analysis of full contracts, the deontic-based $\mathcal{CL}$ proves most suitable.

The verifying of contracts written in $\mathcal{CL}$ has been successfully demonstrated by converting the $\mathcal{CL}$ specification into the extended μ-calculus $\mathcal{C}\mu$ and using the NuSMV verifier (Pace et al., 2007).

Fenech et al. (2009b) point out that while $\mathcal{CL}$ includes a trace semantics for runtime monitoring, these are inadequate for the detection of conflicts as they do not capture the deontic aspects of the contract. They go on to present their own version of the trace semantics suitable for conflict analysis (Fenech et al., 2009c) and demonstrate a custom-built tool for this purpose named CLAN ($\mathcal{CL}$ ANalyser) (Fenech et al., 2009b).

2.5 Chapter summary

In this chapter we first tackled the basic problem of how to represent a contract and went on to introduce the field of deontic logic, showing the extent of its adoption in the area of formal contract representation. We explained the temporal and reparational extensions often added to deontic logic (referred to as OPP-logic) and pointed out some of the paradoxes that have plagued the field, along with how they are generally avoided. Some of the salient issues with OPP-logics were discussed, and comparisons were made between some of the major works in this area.

Chapter 3

Controlled natural languages

In this chapter we briefly discuss the background of natural language handling in computer applications, moving on to focus on controlled natural languages and the arguments for their adoption. After going over some of the classical grammar formalism used in natural language processing, we introduce the Grammatical Framework and describe the benefits it offers. Finally mention some of the successful usages of the framework, and its relevance to the area of controlled natural languages.

3.1 Processing natural languages

Ever since the creation of computers there has existed a very fundamental communication problem—humans and computers don't talk the same language. Human knowledge generally is represented and shared using natural languages, but computers can only process information which is represented formally. Both types of language have their purpose, and there exists quite a "conflict between the wish to use natural languages and the need to use formal languages" (Fuchs et al., 2008)—which is one of the major concerns of natural language processing (NLP).

This issue is traditionally split into two separate but related problems; the *generation* of natural language phrases from formal statements, and their *analysis* into formal expressions:

$$\text{natural language} \underset{generation}{\overset{analysis}{\rightleftarrows}} \text{formal representation}$$

Figure 3.1(a) shows the so-called Vauquois triangle (Vauquois, 1968), which demonstrates the concept of machine translation (MT) via the processes of analysis and generation. The top vertex of the triangle represents the intermediary formal representation of the information under translation, with the vertical position of the 'Transfer' arrow providing some indication as to the linguistic depth of this representation (Pace and Rosner, 2010).

In this section we first look at the simple *template-based* approach to handling natural languages, followed by an introduction to the concept of controlled of natural languages

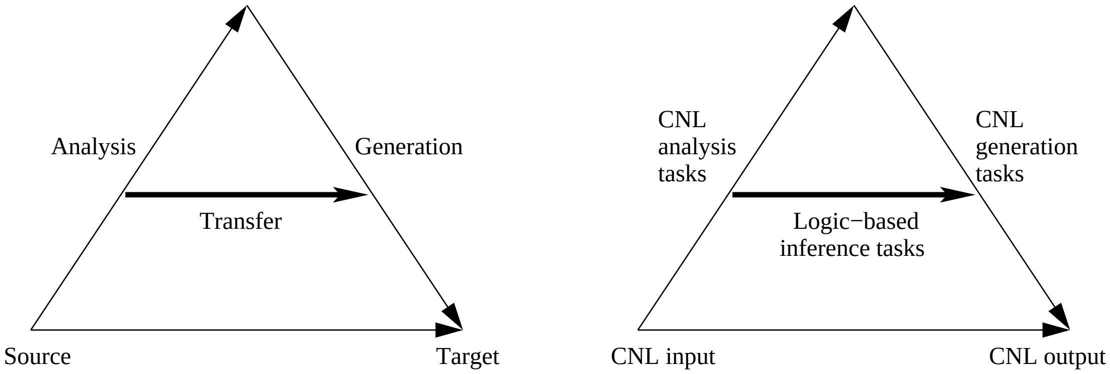


Figure 3.1: (a)Vauquois Triangle for machine translation; and (b) Controlled natural language triangle (see § 3.2). Taken from Pace and Rosner (2010).

and a discussion of their suitability for machine translation to and from contract logics.

3.1.1 The template approach

One of the most basic ways of handling natural language generation is the "template" or "canned text" approach (Dale et al., 2000), which essentially uses pre-defined phrases containing slots where names and values can then be entered as needed. A common social media example of such a template could easily be:

```
<number> people like <name>'s status.
```

By simply inserting the known values in the correct positions, a seemingly natural phrase can be generated:

```
Three people like Paul's status.
```

In restricted scenarios this simple approach can produce satisfactory results, however it is easy to see that its expressivity is limited entirely by the phrases that have been manually entered into the system. Furthermore, there is no inherent agreement between the canned text and the inserted values, meaning programmers need to explicitly check for values of *1*, names ending in *s* and so forth. Without such checks, something like the following erroneous phrase could easily be generated:

```
One people like James's status.
```

While in some settings these errors may be acceptable, they are clearly not correct use of natural language.

This template approach may also arguably be used for natural language "analysis", by matching an input phrase against a corresponding template and stripping away the canned text to obtain the desired values. However this is again highly limited, and such a system would only be able to parse phrases which match *exactly* with the original templates. This approach shows absolutely no level of linguistic understanding, and fidelity is cannot be guaranteed since the canned text has no "systematic relation to the underlying rules" (Pulman, 1996).

3.2 Controlled natural languages

A more involved way of tackling machine translation between formal and natural languages is through the use of *controlled* natural languages (CNLs). CNLs are artificially created subsets of regular natural languages which have a reduced vocabulary, syntax and semantics in order to minimise ambiguity and maximise clarity (Pace and Rosner, 2010; Pulman, 1996). While not as expressive as a full natural language, the main purpose of a CNL is to be adequately expressive for regular people to understand and use, yet precise enough to facilitate automated generation and analysis (Pace and Rosner, 2010).

Syntax restrictions

To give a clearer idea of the kinds of restrictions which CNLs usually involve, Pace and Rosner (2010) provide us with the a few examples of CNL phrases along with their fully natural equivalents, as shown in table 3.1. *More examples of phrases in CNL can be found in § 3.2.1.*

Natural	Upon accepting a job, the system guarantees that the results will be available within an hour unless cancelled in the meantime.
Controlled	`if SYSTEM accepts Job, then during one hour it is obligatory that SYSTEM make available results of Job unless SOMEONE cancels Job.`
Natural	Only the owner of a job has permission to cancel the job.
Controlled	`it is permitted that only owner of Job cancels Job.`
Natural	The system is forbidden from producing a result if it has been cancelled by the owner.
Controlled	`If owner of Job cancels Job, it is forbidden that SYSTEM produces result of Job.`

Table 3.1: Examples of CNL phrases and their natural equivalents, from Pace and Rosner (2010).

CNLs for translation

The main reason for designing and using controlled languages is their suitability for automatic translation to and from formal representations. The use of CNLs for automatic machine translation closely matches the traditional Vauquois model (Vauquois, 1968), as illustrated in figure 3.1(b). As such, the CNL model is a more specific version of the Vauquois one, where the transfer method is defined to operate using logic-based inference (Pace and Rosner, 2010).

Input issues

Reading and understanding phrases written in a CNL is relatively easy to do for anyone who is fluent in the "full" version of the language (Fuchs et al., 2008). However, composing sentences which would be considered valid by a given CNL is trickier. Because at heart they are formally defined, CNLs are not forgiving—the way people are—when it comes to the range of phrases they may include. Thus for an application to successfully allow user input of CNL phrases, a degree of user learning and some way of ensuring that valid statements are entered is required (Pulman, 1996).

For examples of the input methods enabled by the Grammatical Framework, refer to § 3.4.3.

3.2.1 Examples of CNLs

Computer Processable English (CPE)

CPE (Pulman, 1996) is a CNL designed for knowledge-representation applications, with the aim of improving the acquisition and management of knowledge in expert systems. In order to achieve portability between the controlled language and underlying logic representation, the Knowledge Interchange Format (KIF) (Patil et al., 1992) is used as an intermediate "canonical" representation. This extra layer avoids the difficulties encountered in direct translation from CNL to executable representation language, but requires that the latter is convertible into KIF.

The following is an example of KIF rule linearised into CPE (Pulman, 1996).

```
TASK: dispatch an RPV
INPUTS: a blocked route, B; a location, L.
PRECONDITIONS:
    there must be at least 1 available RPV
    there must be less than 3 active RPVs
ACTION:
    make a mission record, M,
    which has mission route, R,
    and which has blocked route, B,
    and which has blocked location L.
    R’s launch location is divisional HQ.
    R’s recce route is L.
    R’s search radius is 3000.
deploy an available RPV, X,
    which has mission route R,
    and which has mission type ‘single mission’.
change status of X from ‘available’ to ‘active’.
```

Note how the problem of pronouns is avoided by naming each entity in the rules by a letter-code. This is not a particularly natural aspect of the language, but is deemed to be an acceptable trade-off.

A tool built specifically for facilitating this conversion from controlled English into KIF formulas was produced in the Controlled English to Logic Translation (CELT) project (Pease and Murray, 2003). The CELT tool includes a large lexicon imported from WordNet[1], and allows sentences to have multiple possible parses. The parsing itself is achieved using definite clause grammars (§ 3.3) with feature grammar extensions.

Attempto Controlled English (ACE)

The Attempto project[2] produced its own knowledge representation CNL called 'ACE', with considerable success (Fuchs et al., 2008). Like CPE, ACE also uses an intermediary representation format known as discourse representation structures (DRS), which can then be easily translated into first-order logic. The ACE language "has been used in several applications, and was adopted as the controlled language of the EU Network of Excellence REWERSE (Reasoning on the Web with Semantics and Rules)" (Kaljurand, 2009).

ACE is one of the best examples of how convincing a CNL can be, as illustrated in the selection of examples in table 3.2.

Simple Sentences	*A trusted customer inserts two valid cards.* *A customer inserts two cards manually.* *John's customer who is new inserts 2 valid cards of Mary manually into a slot X.*
Composite Sentences	*If a card is valid then a customer inserts it.* *A customer inserts a VisaCard or inserts a MasterCard, and inserts a code.* *No customer inserts more than 2 cards.*
Commands	*John, go to the bank!* *John and Mary, wait!* *Every dog, bark!*

Table 3.2: Examples of phrases in ACE, taken from Fuchs et al. (2008).

In addition to the language itself, ACE is supported by a number of tools for parsing, reasoning, and conversion into semantic web languages. The parsing tool for ACE—the Attempto Parsing Engine (APE)—features both standard and user-defined lexicons, and can provide various kinds of parsing output to the user; APE can produce translations in various forms of first-order logic, the Web Ontology Language (OWL), Semantic Web

[1] `http://wordnet.princeton.edu/`

[2] `http://attempto.ifi.uzh.ch/`

Rule Language (SWRL) and RuleML (Fuchs et al., 2008). The language also enjoys an automatic reasoning tool named the Attempto Reasoner (RACE) along with the semantic web oriented ACE View, AceRules and AceWiki tools (Fuchs et al., 2008).

Processable English (PENG)

Taking the parsing of CNLs one step further, Processable English (PENG) is a CNL born out of ACE whose main focus is on producing a syntax-aware, predictive text editor and parsing tool which can be used without prior knowledge of the CNL grammar itself. This tool, called ECOLE, helps users write grammatical PENG sentences first time round by looking ahead and indicating in real-time whether a phrase is grammatical or not (Schwitter, 2002).

3.3 Unification-based grammar formalisms

A *grammar formalism* is a language for the definition of grammars, such as BNF (Backus-Naur Form). Many of the grammar formalisms used in computational linguistics belong to the family of *unification grammars*. Such grammars revolve around formal descriptions of grammatical units by using attribute-value pairs (Cole et al., 1997). These are called *feature structures*, and their merging together through the checking of grammatical information is known as *unification*. Unification grammars provide an elegant way of representing syntactical constraints that would be difficult to express with context-free grammars (CFGs) alone (Jurafsky and Martin, 2009).

Definite clause grammars (DCGs)

The DCG (Pereira and Warren, 1980) is one of the simplest and earliest examples of a unification grammar. Traditionally implemented in Prolog, DCGs provide the concept of grammar *features*, making them more powerful than CFGs (equivalent to context-sensitive grammars) and therefore more linguistically useful. Consider the following example of a DCG grammar adapted from Blackburn et al. (2006):

```
s --> pro(subject), vp.
vp --> v, pro(object).
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
v --> [likes].
```

By using features, this small grammar ensures that *he likes her* is valid, but *he likes she* is not. Despite their initial popularity, DCGs have fairly low-level descriptive capabilities, and their lack of types makes them ineffective for large scale tasks (Ranta, 2004).

PATR

The PATR-II formalism (Shieber, 1984) is a computer language created by the PATR group for encoding linguistic information. It was one of the first attempts to create "a declarative computer language with clear semantics designed specifically for encoding linguistic information" (*ibid.*). PATR-II was the first formalism to use records for representing complex grammatical objects (Ranta, 2004), and works using feature agreement.

Head-driven phrase structure grammars (HPSG)

A lexical *head* is the most grammatically important word in a phrase (Jurafsky and Martin, 2009), and HPSGs (Pollard and Sag, 1994) work by annotating each sub-tree in a parse with that phrase's lexical head. HPSGs express complex grammatical objects as typed records or *signs*—inherited from PATR-II—which contain "both syntactic and semantic information" (Ranta, 2004). This concept means that the HPSG formalism is based on *lexicalism*, where the lexicon is made up of richly structured entities rather than just a list of entries (Pollard and Sag, 1994).

Through the DELPH-IN project[3] many large open-source HPSG grammars are being developed for various languages, including English, German and Japanese.

3.4 The Grammatical Framework

The Grammatical Framework (GF) (Ranta, 2004) is a specialised functional language for defining Montague-style grammars extended with dependent types. GF differs from unification-based grammar formalisms by having separate abstract/concrete syntax rules, a strong type system, and inherent support for multilingual grammars. GF grammars are declarative in nature, with a primary focus on the linearisation of syntax trees. By writing an abstract GF grammar and defining *how* it should linearise into one or more natural languages (concrete grammars), GF is able to derive both a generator *and* a parser for each of those languages. (Ranta, 2004).

3.4.1 Abstract & concrete syntaxes

When implementing a computer language it is common to make the distinction between its *abstract* and *concrete* syntaxes. The abstract syntax defines the language's hierarchical structure using a system of trees, while the concrete syntax dictates how the language "looks like" (Ranta, 2004; Angelov, 2009). This separation is based on the idea that type checking and semantics are more relevant to the abstract level, while syntax details are more a concrete concern (Ranta, 2004).

[3] Deep Linguistic Processing with HPSG Initiative (DELPH-IN). `http://www.delph-in.net/`

To demonstrate this duality, consider this "Hello World" abstract grammar taken from the GF Tutorial[4]:

```
abstract Hello = {
    flags
        startcat = Greeting ;
    cat
        Greeting ; Recipient ;
    fun
        Hello : Recipient -> Greeting ;
        World, Mum, Friends : Recipient ;
}
```

This grammar simply defines two linguistic categories (`Greeting` being the start category for parsing and generation) and a total of four functions for building meaning. To this, we can add any number of concrete syntaxes which define how this abstract structure can be linearised into text[5]:

```
concrete HelloEng of Hello = {
    lincat
        Greeting, Recipient = {s : Str} ;
    lin
        Hello recip = {s = "hello" ++ recip.s} ;
        World = {s = "world"} ;
        Mum = {s = "mum"} ;
        Friends = {s = "friends"} ;
}
```

```
concrete HelloMlt of Hello = {
    lincat
        Greeting, Recipient = {s : Str} ;
    lin
        Hello recip = {s = "bonġu" ++ recip.s} ;
        World = {s = "dinja"} ;
        Mum = {s = "mama"} ;
        Friends = {s = "ħbieb"} ;
}
```

Both concrete syntaxes define how the `Greeting` and `Recipient` *categories* are linearised as a single string `s`. The three `Recipient` *functions* are given direct linearisations which differ between languages, and the `Hello` function returns a complete phrase by concatenating the recipient string (`recip.s`) to the world "hello".

This allows a single, common abstract syntax to be linearised in as many different ways as there are concrete syntaxes and forms the basis for GF's approach to multilingual translation. Further to simple example shown above, each concrete syntax is able

[4] `http://www.grammaticalframework.org/doc/gf-tutorial.html`

[5] *Ibid.*

to define its own linearisation rules, to cover linguistic phenomena such as inflection and agreement on a per-language basis.

3.4.2 Linearisation-centric records

As shown in the previous example, grammatical objects in GF are represented as records, with a focus as to *how* each object is linearised. This is comparable to the use of *signs* in HPSG, however while signs only capture an *instance* of a word's possible forms and inflections, GF's *records* encode all such forms within a single object (Ranta, 2004). Consider the English noun form "integers"; since HPSG signs are obtained via string analysis, a sign for "integers" would be encoded as:

$$\{\ cat = Noun\ ;\ s = \text{“}integers\text{”}\ ;\ num = Plural\ \}$$

Note how this does not provide any information about the *singular* form of the word; merely that "integers" is a plural noun. On the other hand, GF would encode this same term as a *table* of possible linearisations:

$$\{\ cat = Noun\ ;\ s = \textbf{table}\ \{Singular \Rightarrow \text{“}integer\text{”}\ ;\ Plural \Rightarrow \text{“}integers\text{”}\}\ \}$$

Note how in contrast to the HPSG example above, this record contains the linearisation for both the singular "integer" and plural "integers" (example taken from Ranta (2004)).

3.4.3 Input possibilities

A major part of GF is its partial evaluation algorithm or *incremental parser* (Angelov, 2009), which gives rise to interesting guided-input possibilities. By presenting the user with a list of possible words which may come next in a partial sentence, they are able to construct grammatical sentences in an auto-complete fashion. This is highly useful as it ensures that only syntactically-correct phrases are entered first-time round, and will avoid user frustration of trying to construct valid sentences in free-text.

Two implementations of this method of input—the drop-down suggestions[6] (see figure 3.2) and the "fridge poetry magnets"[7]—have been developed by the GF team, and are distributed as examples with the GF source code. These input methods are of particular interest to the area of CNLs, as they help avoid the problem of users having to know what is grammatical in a particular CNL and what is not.

[6] Online demonstration at `http://tournesol.cs.chalmers.se:41296/translate/`

[7] Online demonstration at `http://tournesol.cs.chalmers.se:41296/fridge/`

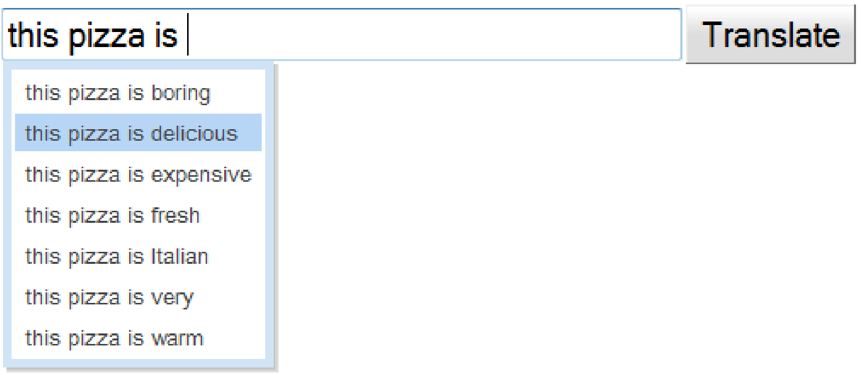


Figure 3.2: The drop-down suggestions input method as used in the “Translate” application; demonstrated online at `http://tournesol.cs.chalmers.se:41296/translate/`

3.4.4 Resource Grammar Library

In addition to the GF formalism itself, an important part of the framework is the Resource Grammar Library which is distributed with it. This library is a set of GF parallel grammars in a number of languages including Bulgarian, Catalan, Danish, English, Finnish, French, German, Italian, Norwegian, Polish, Romanian, Russian, Spanish, and Swedish (Ranta, 2009). All grammars in the library share the same abstract syntax (see § 3.4.1), allowing it to be used as a “resource for language processing tasks, such as translation, multilingual generation, software localization [and] natural language interfaces” (Ranta, 2009). The GF Resource Grammar Library is open-source[8].

3.4.5 Projects using GF

First created in 1998 and released under the GNU General Public License (GPL)[9], GF has been used in a number of projects which involve automated translation to and from natural languages, often within a multilingual setting.

Alfa

The Alfa proof editor (Hallgren and Ranta, 2000) is one such project which has incorporated GF as a plug-in for linearising formal proofs into various natural languages. This allows users to construct proofs interactively while instantaneously viewing them as natural language texts. In addition, users are able to extend this capability to new languages by writing additional GF grammars for the concrete syntax (Ranta, 2004).

[8] Available under the GNU Lesser General Public License (LGPL); `http://www.gnu.org/licenses/lgpl.html`

[9] `http://www.gnu.org/licenses/gpl.html`

Software specification

Another area in which GF has been used is the formal software specification. Semi-formal software specification languages such as the Object Constraint Language (OCL) are widely used in industry, but a gap exists between them and informal natural language specifications (Ranta, 2004). In (Hähnle et al., 2002) the authors describe the design of a computer-aided software engineering (CASE) tool which aims to bridge this gap by using GF to "combine linguistic and logical models" and provide a natural language interface to formal software specifications. This went on to form part of the KeY system, and has been used successfully in translating the formal specifications of the Java Card API from OCL into English (Johannisson, 2005).

WebALT

One of the largest projects to date to use GF successfully is the WebALT project. Concluded in 2007 and since turned into a commercial entity, WebAlt combined existing standards for representing mathematics with GF to create a body of language-independent mathematical exercises for educational purposes. Thus, mathematical content written using the WebAlt authoring tool can be easily localized to any European language/culture included in the WebAlt set of GF grammars (WebALT Consortium, 2007).

3.4.6 GF and CNLs

The use of GF for building controlled languages shall be discussed in more detail in chapter 6, yet at this stage we will just indicate some work which has already been done in the area. Ranta and Angelov (2009) consider the use of GF as a tool for implementing CNLs, using the the ACE language (see § 3.2.1) as a case study. The authors claim to have found the implementation "quick and mostly smooth", and subsequently proceeded to port it to French, German, and Swedish. The main problem encountered was that certain ACE structures are not covered by the GF Resource Library, as they are in fact considered to be linguistically incorrect.

3.5 Chapter summary

This chapter covered the basics of controlled natural languages (CNLs), and the benefits they offer to the area of NLP. We listed some of the major works in the area of CNLs, and considered the issues involved in their use. In discussing language grammar formalisms we introduced the Grammatical Framework (GF), explaining its features, discussing some of the projects in which it has successfully been employed, and pointing out its relevance to the area of controlled languages.

Chapter 4

BanaNomic

This chapter introduces the game of Nomic and its unique concept of self-amending rules. We take a brief look at the many variants of Nomic, with particular focus on the only known automated version of the game, PerlNomic. We then go on to define our own version of Nomic called BanaNomic, *explaining its basic features and how the game is played.*

4.1 The game of Nomic

Games of all types are always governed by some set of rules which define what can and cannot be done, and what the consequences of various actions and game states may be. For example, one of the rules of Monopoly[1] famously states that:

> *When a player passes "Go" they may collect £200.*

Rules likes this are very much based on the basic contract concepts discussed so far (namely obligation, permission and prohibition), and are assumed to be static and infallible.

The game of Nomic, originally conceived by Peter Suber in his book *The Paradox of Self-Amendment* (Suber, 1990), is a somewhat unconventional game where the only move possible is to actually change the rules of the game themselves. The game starts with an initial rule set, and with every turn players propose additions and amendments to this list and vote on other players' proposals. All the rules in the game are subject to amendment, including how victory is defined and who is eligible to play. Loosely modelled on real-life government systems, Nomic is a game based on the fundamental contracts concepts with the additional concepts of self-reference and self-amendment.

[1] The game of Monopoly is trademark of Hasbro, Inc.

4.1.1 Game objectives

Regular types of games are played with fixed, clearly defined goals such as earning the most points, being the last player standing, and so on. To this end, all players play with exactly the same objectives. However the same cannot be said of Nomic; since the victory conditions for the game are themselves written as rules—and therefore open to modification—the game's objectives are actually "moving targets", and different players may have different sets of goals to each other. Consider the following situation:

Rule 1 *The player with the most points wins the game.*
Rule 2 *Players earn 15 points for successfully amending a rule.*
George *has 20 points. It is his turn to play.*
Olivia *has 10 points.*

I this game George is currently winning, but he is worried that Olivia will quickly overtake him if she simply amends one rule successfully. Thus, he cunningly changes rule 2 to:

Rule 2 *Players earn* ***no*** *points for successfully amending a rule.*

Now Olivia can never earn enough points to have more than George—but she doesn't need to. By simply changing rule 1 to:

Rule 1 *The player with the* ***least*** *points wins the game.*

Olivia has instantly put herself in a winning position!

This small example is just one of many ways in which playing Nomic is really quite unlike any other game. Every single game of Nomic can turn out differently to the last, and no one can tell ahead of time how a game will turn out. In fact, it has even been said that a game of Nomic has the potential to become any other game (Vreeswijk, 1995).

4.1.2 Flavours of Nomic

A large number of Nomic variants exist, and hundreds of games have been played by thousands of players worldwide for over 20 years (Phair and Bliss, 2005). Originally maintained through electronic mailing lists, most Nomic games are now maintained online through bulletin boards and wikis[2].

One example of an active Nomic game (at the time of writing) is *B Nomic*[3], which dates back ten years to 2000. The game action happens via a mailing list, and a summary of the state of the game is maintained as a wiki. The layout of the game is quite elaborate, with different "ministries" taking care of specific tasks; These include an electoral

[2] Judging by the Nomic games lists at `http://www.nomic.net/~nomicwiki/index.php/NomicDatabase` and `http://www.nomic.net/archive.html`

[3] Online at `http://b.nomic.net/index.php/Main_Page`

commission, a judiciary and "ministry of ministries". Changes to the game are enacted through a proposition-voting process.

Despite the fact that most Nomic games are maintained electronically, very few attempts seem to have been made at formalising the game such that its rules and actions may be automatically verifiable.

4.1.3 Attempts at formalising Nomic

Vreeswijk (1995) was the first to speculate about the possibility of formalising and automating Nomic. The author's interest lay in creating a completely automatic Nomic simulator that would allow the analysis of self-modifying communications protocols. Much like voting systems used in everyday life, Nomic's voting system has a self-regulating aspect to it which protects the rules from "weird proposals" (Vreeswijk, 1995). The interest in studying such protocols is their potential use in distributed computing and multi-agent systems, however the authors do not pursue the building of any actual systems.

Nomic as ontologies

Some work has in fact been carried out in representing Nomic rules as an OWL ontology. As part of the ESTRELLA project an, ontology of Nomic rules was implemented in LKIF-Core, aiming to "organize the structured information" present in a set of Nomic rules and to "allow reasoning and problem solving" over that knowledge (Klarman et al., 2008). Some interesting issues arose regarding the concepts of change and version control within the represented knowledge, however this system as such does not allow active playing of the game itself.

4.1.4 PerlNomic

PerlNomic (Phair and Bliss, 2005) is the only computer-based Nomic variant found which employs automatic rule checking. Rather than being played in a natural language like English, rules in the game are actually arbitrary snippets of code written in the Perl programming language. The game turned out to be quite a success, with five games running over 3 years and attracting roughly three thousand visitors worldwide (Phair and Bliss, 2005).

Avoiding ambiguity

The idea of playing a game purely in Perl code may seem cumbersome, but in fact there is a major advantage to this kind of approach—the inherent lack of ambiguity in computer languages. Unlike in the real world where natural language ambiguities in the law must be assessed by human judges, in PerlNomic the only judge required is the Perl interpretor itself. By taking advantage of the strict structure of Perl, the authors were able to build a formal version of Nomic with modest implementational requirements and a sizeable user base (Phair and Bliss, 2005).

Rules as code

The idea that every rule in the game is actually a Perl script opens up some interesting possibilities. Apart from the ability to patch any other script running on the server hosting the game, players would easily be able to write proposals which replace the game interpreter with their own version, or even create completely unrelated services on the host machine (such as chat or email server) (Phair and Bliss, 2005). In this sense the game is quite flexible in terms of what players are allowed to create.

Laws & limitations

Despite this apparent flexibility of PerlNomic, it of course is not without its limitations. Since the rules are simply snippets of Perl code, the scope of the game is essentially limited to the actions that are performable from within the Perl interpreter; in the words of the authors, "there can be no rules about hot dogs in *this* game" (Phair and Bliss, 2005).

While players of PerlNomic are able to add any scripts they like, these scripts are still ultimately limited by a hierarchy of lower-level structures—the Perl interpretor, operating system kernel, processor instruction set and so on. However—without getting too philosophical—it is pertinent to note at at *some* level there will always be a limiting factor to self-amendment capabilities, whether it is technical or rooted in human communication itself.

4.2 BanaNomic

In line with the aims of the project, a version of Nomic was to be implemented using a suitable contract logic, upon which a suitable CNL interface could be built. Rather than attempt to implement an already existing flavour of Nomic, we went about designing our own take on the game, tailored to our specific needs and restrictions. Thus—inspired by the traditional geek affinity for all things simian—*BanaNomic* was born.

For a full explanation of the game, refer to the User Guide in appendix A. Examples of *BanaNomic* gameplay, taken directly from the project's evaluation period, can be found in appendix B.

4.2.1 The setting

Of bananas, monkeys and trees

The setting for *BanaNomic* was chosen to provide some light-hearted fun. Each user of the game plays the role of a monkey living in a tree with other monkeys, fighting to pick bananas and defend their own stash. Governing rainforest life are a series of rules dictating what one can, cannot, and must do. These rules cannot be violated, but players are allowed to add and remove rules at will. Thus, as the tree contains only a finite number of bananas, players must use their wits to manipulate the rules to their advantage and collect the most bananas before the end of the game.

The tree itself is split into four horizontal sections, from bottom to top: the *forest floor*, *under story*, *canopy* and *emergents* level. Players can climb up and down the tree and pick bananas freely, except that no bananas grow at the forest floor level.

Playing turns

Players start the game on the forest floor with zero bananas each. With each turn, a player may (depending on rule constraints) carry out any of the basic actions below, enact a new rule and/or abolish a current one. The game is governed by banana-time—which is coincidentally synchronised to increment one *b'clock* every 24 hours—and players are allowed to play up to once per day.

Game actions

While manipulating the rules is a vital part of *BanaNomic*'s gameplay, a number of basic actions are also included in order to make the game more accessible to users. These are:

- *Climb up* or *down* the tree to change one's current level.
- *Pick a banana* from the tree and add it to one's stash. Players cannot pick bananas from the forest floor, and so would need to climb up before being able to do so.
- *Throw a banana* at another player on the same level, which will cause the other player to lose all their bananas and end up on the forest floor again.

4.2.2 Rules of the rainforest

Rule manipulation and voting

While in Nomic the modification of rules is governed by a democratic voting process (at least initially), in *BanaNomic* this was dropped in favour of a direct enactment and abolishment system. Including a voting system in the game would not have been a problem from an implementation perspective; however because of the limited time available for evaluating the project, this choice was made to ensure that users can get their hands dirty with amending the rules from the get-go, without cumbersome voting processes taking up too much time.

Initial rules

The initial rules which are active at the start of the game were fairly basic:

1. All players are permitted to climb up the tree, climb down the tree, pick a banana, and throw bananas at other players.
2. At some point in the game, every player is obliged to enact a new rule.
3. At all times from 3 b'clock onwards, all players are permitted to abolish an existing rule.
4. If any player has more than 10 bananas, then all players are forbidden from throwing bananas at other players.

4.3 Chapter summary

We started this chapter by introducing the self-amending game of Nomic, discussing how and why it's played, its numerous variants and some approaches made at formalising it. After paying particular attention to the only known automated version of the game, PerlNomic, we defined our own take on Nomic called *BanaNomic* and explained its features. Chapters 5 and 6 go on to describe the design of a contract logic and CNL grammar for *BanaNomic*, respectively.

Chapter 5

A contract logic for Nomic

In this chapter we describe the design of a contract logic for the game of BanaNomic, *based on existing works but customised to our needs. After defining the grammar in detail, we then go on to discuss its informal semantics and finally comment on its differences from other contract languages encountered in the literature.*

5.1 Formal grammar

The contract grammar devised for *BanaNomic* is based on the *OPP-logic* as defined in Pace and Schneider (2009), with a number of simplifications which are justified in § 5.4. Where applicable, the same notation has been used.

Time

A simple category to represent a finite amount of time. It is also 'nullable', to allow for open-ended time clauses (see § 5.3.4). Time in *BanaNomic* is governed by a global clock which counts upwards from zero at some pre-defined interval, and all instances of *Time* in the logic represent absolute temporal values. Thus there is no concept of *x time units before* or *y time units from now*.

Player

A $PlayerName$ is used to refer to a player uniquely by their name and is either a type synonym for a *string* or the generic player ϕ. The generic player is a particular instance of this category used for building clauses which refer to all/any player in the game (see § 5.3.5). The *Player* itself is simply a function which given a $PlayerName$ will return that player's points and level as integers.

$$\begin{aligned} PlayerName &= String \mid \phi \\ Player &:: PlayerName \rightarrow \langle Int, Int \rangle \end{aligned}$$

Rule

This is a simple function which, given a unique *RuleID*, will return that rule's corresponding *Contract*.

$$Rule :: \langle RuleID \rangle \rightarrow Contract$$

Contract

A contract is a clause which ultimately terminates as a deontic expression (except for the empty contract ε). The *Contract* category allows operations over such clauses, providing choice ($+$) between two contracts, always ($\Box$) and sometimes ($\Diamond$) operators (optionally over a specified time period—see § 5.3.4), and two conditional operators. The single-triangles ($\lhd\ \rhd$) will evaluate to either the first or second contract, depending on whether the given deontic expression is active within the language or not. The double triangles ($\lhd\!\!\lhd\ \rhd\!\!\rhd$) behave similarly, by evaluating the given *Query*.

$$\begin{aligned} Contract = \varepsilon &\mid DeonticExp \mid Contract + Contract \\ &\mid \Box\,[t_1, t_2]\,Contract \mid \Diamond\,[t_1, t_2]\,Contract \\ &\mid Contract \lhd DeonticExp \rhd Contract \\ &\mid Contract \lhd\!\!\lhd\ Query \rhd\!\!\rhd Contract \end{aligned}$$

The *Contract* category contains no operators for negation, repetition, sequencing or checking for the presence of an action. In addition, CTD/CTP operators are not directly supported in the logic (for a discussion why, see § 5.4).

Deontic expression

Deontic expressions serve as the basic element for the logic, expressing a single obligation ($\mathbb{O}$), permission ($\mathbb{P}$) or prohibition ($\mathbb{F}$). Each deontic expression is tied to a player, although for convenience this could also be the generic player (ϕ), which is taken to represent *all players* or *any player*, depending on context (see § 5.3.5).

$$\begin{aligned} DeonticExp = {} & \mathbb{O}\ \ (PlayerName : Activity) \\ & \mid \mathbb{P}\ \ (PlayerName : Activity) \\ & \mid \mathbb{F}\ \ (PlayerName : Activity) \end{aligned}$$

Although the absence of permission (or obligation) will be interpreted as implicit prohibition, the $\mathbb{F}$ operator is still included for overriding other clauses. Thus, the deontic operators have the following order of precedence: $\mathbb{F} \geq \mathbb{O} \geq \mathbb{P}$

Query

The query will evaluate to *true* or *false*, based on its parameters and the current state of the game. The operators over queries provided are negation ($\overline{query}$), conjunction (&) and disjunction (+). Two helper functions *Points* and *Level* are defined for querying player information.

$$\begin{aligned} Query = {} & \overline{Query} \mid Query \,\&\, Query \mid Query + Query \\ & \mid Points\,(PlayerName) > Int \\ & \mid Points\,(PlayerName) < Int \\ & \mid Points\,(PlayerName) == Int \\ & \mid Level\,(PlayerName) == Int \\ & \text{where} \\ Points :: {} & PlayerName \rightarrow Int \\ Level :: {} & PlayerName \rightarrow Int \end{aligned}$$

Activity

An activity is a wrapper around one or more atomic actions. Only two operations over activities are provided—choice (+) and concurrency (&).

$$Activity = \varepsilon \mid Action \mid Activity + Activity \mid Activity \,\&\, Activity$$

The initial intention was to also include *sometimes* and *always* operators, however these were later omitted to avoid added complications with the semantics. Operators for negation, repetition and sequencing were also not included.

Action

The *Action* category represents a single, atomic action. The possible actions available in the game are "hardcoded" in the grammar, rather than having parametric constructors. Also no operations over actions are provided, as these are handled by the *Activity* category.

$$\begin{aligned} Action = {} & ClimbUp \mid ClimbDown \mid PickBanana \\ & \mid ThrowBanana\,(PlayerName) \\ & \mid Enact\,(Contract)^{\dagger} \mid Abolish\,(RuleID)^{\dagger} \end{aligned}$$

†: Note that when referring the general 'concept' of enacting and abolishing rules, we will use the *Enact* and *Abolish* constructors without parameters (see contract examples in § 5.2).

Gamestate

The *Gamestate* category represents an entire *BanaNomic* game at a given point in time. It is composed of a tuple containing the following items:

$Gamestate :: \langle T, B, P, R, O \rangle$

where

$T :: \langle Time \rangle$ is the current game time

$B :: Int$ is the number of bananas left on the tree

$P :: [\langle PlayerName, Int, Int \rangle]$ are the players with their points and levels

$R :: [\langle RuleID, Contract \rangle]$ are the currently active game rules

$O :: [DeonticExp]$ are the pending and satisfied obligations

5.2 Contract examples

Using the logic defined above, a few example contracts and their natural language readings are given below.

1. $\mathbb{O}\,(\text{“Paul”} : ThrowBanana\,(\text{“Ringo”}))$
   Paul is obliged to throw a banana at Ringo.

2. $\mathbb{P}\,(\text{“John”} : PickBanana) \lhd\!\lhd\ (Points\,(\text{“John”}) < 5) \rhd\!\rhd\ \varepsilon$
   If John has less than 5 bananas then he is permitted to pick one (no clause is defined otherwise).

3. $\varepsilon \triangleleft \mathbb{P}\,(\text{“George”} : \mathit{Enact}) \triangleright \Diamond\,[\varepsilon, 9]\,\mathbb{O}\,(\text{“George”} : \mathit{Abolish})$
   If George is not permitted to enact a rule, then he shall be obliged to abolish a rule at some point before 9 b'clock.

4. $\Box\,[3, \varepsilon]\,\mathbb{F}\,(\phi : \mathit{ClimbUp}\ \&\ \mathit{PickBanana})$
   At all points from 3 b'clock onwards, all players are forbidden from concurrently climbing up the tree and picking a banana.

5.3 Semantics

5.3.1 Deontic inference and precedence

Closed world

The grammar will operate under a *closed world* assumption, where the absence of explicit permission implies prohibition. In other words, players are forbidden from carrying out an action unless there exists a clause permitting them to do so.

Obligation and permission

One design choice that arises in designing a contract grammar is the relationship between permission and obligation. Should it be possible to be obliged to do something but not permitted to do it? In this project we avoided this tricky situation altogether, by simply stating that wherever an obligation to carry out an action exists, permission to do that action is inherently implied.

Prohibition is king

Despite the fact that prohibition may be defined as absence of permission (or obligation), the *forbidden* operator $\mathbb{F}$ was still included to allow the possibility of overriding previous permissions. Thus the deontic operators are given the following order of precedence: $\mathbb{F} \geq \mathbb{O} \geq \mathbb{P}$. Note that this still allows for a similar situation as that described above, where a user may be obliged to perform an action but yet forbidden to do so.

5.3.2 Choice

Certain clauses in the grammar involve an element of choice. Specifically, these are the *Choice* operators over contracts and activities, and the *sometimes* contract operator. Various strategies for implementing choice are available (see § 2.3.3), but for the sake of simplicity—both semantically and in the implementation—all choices in *BanaNomic* shall be implemented as external/angelic. Note that the disjunction operator between queries does not constitute choice in the same way, and has no related concepts of internal/external choice.

5.3.3 Permanent contracts

Any contract which is not enclosed within a *sometimes* or *always* clause is called a permanent contract. For permissions and prohibitions, permanent contracts are effective at every time interval without exception. Permanent obligations behave somewhat differently; while they are instantly effective and must be satisfied immediately, once they are fulfilled they will never again be re-enforced. In other words, they must be satisfied at once, and only once.

5.3.4 Timely contracts

Contracts enclosed within *sometimes* or *always* clauses are effective for only certain time frames. Both clauses may have specific start/end times or may be open ended, as summarised in tables 5.1 and 5.2.

Sometimes

Any contract enclosed in a *sometimes* clause means that it will only be active or satisfiable once within the given time frame (see table 5.1). For permissions and obligations, this means that the action in question may/must be carried out only once. This choice is made by the player in question, i.e. external/angelic choice. A prohibitive contract inside a *sometimes* clause has little purpose.

Expression	Description
$\Diamond\, [Int, Int]$	Clause is valid once between t_1 and t_2.
$\Diamond\, [Int, \varepsilon]$	No end; clause is valid once from t_1 onwards.
$\Diamond\, [\varepsilon, Int]$	No beginning; clause is valid once before t_2.
$\Diamond\, [\varepsilon, \varepsilon]$	No beginning and no end; clause is valid once.

Table 5.1: Time limits for *sometimes* clauses.

Always

Contrary to *sometimes* clauses, contracts within an *always* clause are effective at every point within the given time frame (see table 5.2). Of particular interest, *always* obligations must be satisfied at every time interval until the contract is no longer active.

Expression	Description
$\Box\,[Int, Int]$	Clause is valid at every point between t_1 and t_2.
$\Box\,[Int, \varepsilon]$	No end; clause is valid at every point from t_1 onwards.
$\Box\,[\varepsilon, Int]$	No beginning; clause is valid at every point before t_2.
$\Box\,[\varepsilon, \varepsilon]$	No beginning and no end; clause is valid at every point.

Table 5.2: Time limits for *always* clauses.

5.3.5 The generic player

Individual players in the game are identified by a unique player name, but the generic player ϕ is a special case for referring to all or any players with a single statement. The any-all distinction is determined entirely by the context; the generic player as the subject of the clause is taken to mean "all players", while as the predicate it is interpreted as "any player". Thus a clause like

$$\mathbb{O}\ (\phi : ThrowBanana(\phi))$$

would be interpreted as *"all players* are obliged to throw a banana at *any player*".

5.3.6 Rule manipulation

Being modelled on government systems, rule manipulation in Nomic is typically democratic; that is, proposals for rule additions or amendments are made, they are voted upon, and the motion is passed or rejected. However in our version of the game, we opted for a much simpler direct-manipulation model, where any player may enact or abolish rules without going through a voting process first (provided however that they are permitted to do so).

The reasons for this simplification of the game are twofold; firstly, knowing that the evaluation period available for the project would not be more than two weeks, having a cumbersome proposal-voting system would likely mean that many rules never get to see the light of day, and very few changes to the game's active rule set would actually be observed. Secondly, the implementation of such a system would be non-trivial, requiring specific handling of *propose* and *vote* actions, keeping track of proposals over time, and introducing the issues of permissions and obligations for players to vote. While this would have certainly been interesting to implement and observe, time restrictions forced us to adopt a much simpler rule manipulation system.

Effect on gamestate

Providing that they are permitted, *enact* and *abolish* actions have an immediate effect on the gamestate. In the case of rule enactment, the new rule contract is appending to the game's rule set, and any obligations arising out of this new rule are added to the game's obligations list.

For rule abolishment, the respective rule is removed from the game's list of rules. A special case arises when a rule consisting of a one-time obligation is removed. Consider table 5.3; initially two *sometimes* obligations are enacted, and at 5 b'clock Heather fulfils her obligation, meaning that only Paul's remains pending. But if the rule itself is abolished at 10 b'clock, should Paul's obligation also be removed? In *BanaNomic* we indeed opt to remove pending obligations, although as in shown in this case the issue of 'fairness' towards other players arises.

Time	Actions	Pending Obligations
1	Enact the following rules:	
	$\Diamond\, [\varepsilon, \varepsilon]\, \mathbb{O}\, (\text{Heather} : \mathit{PickBanana})$	Heather : *PickBanana*
	$\Diamond\, [\varepsilon, \varepsilon]\, \mathbb{O}\, (\text{Paul} : \mathit{PickBanana})$	Paul : *PickBanana*
⋮	⋮	⋮
5	Heather : *PickBanana*	Paul : *PickBanana*
⋮	⋮	⋮
10	Abolish both rules from (1).	?

Table 5.3: Example showing the abolition of a clause containing a one-time obligation.

5.4 Comparison with other works

While being based on the OPP-logic from (Pace and Schneider, 2009), the *BanaNomic* contract logic contains some significant omissions from the original for reasons of simplification. To begin with, the *Contract* category contains no operators for negation, repetition or sequencing as these were seen to be do little to improve the expressiveness of the logic at this stage. The checking for the presence of an action presented itself as a feature of dubious use, however in its place the query-conditional operator $\lhd\!\lhd \;\; \rhd\!\rhd$ was added as this was seen to be of more relevance to the game.

In addition to this, a deontic-conditional $\lhd \;\; \rhd$ operator was also provided, which evaluates one clause or another depending on whether the specified deontic expression is present in the contract. This allows for clauses of the type:

> *If Maureen is permitted to pick a banana, then . . .*

Unfortunately CTD/CTP operators are not directly supported in the logic, mainly due to the obstacles encountered in implementation. Rather than using reparation clauses, we rely on *explicit enforcement* of obligations and prohibitions, effectively making it impossible for users to neglect their obligations or do something which they are not permitted to do. This is achieved through the implementation itself (see § 7.5.1).

The *Activity* category serves the same purpose as the *CompAction* class in Pace and Schneider (2009), that is to provide expressions over actions. In our case, *all* operators are "pushed up" to *Activity*, and the *Action* category consists purely of the game's atomic actions.

With regards the *Activity* class, the initial intention was to also include the sometimes ($\Diamond$) and always ($\Box$) operators, which would allow for further investigation into the internal/external choice issue (see § 2.3.3). However, despite this, their addition would have also introduced a host of other semantic problems which were beyond the scope of this project, and so they were ultimately omitted from the logic too. Operators for negation, repetition and sequencing over activities were also not included for the same reasons as described in the case of contracts, above.

5.5 Chapter summary

In going through and describing the contract logic for *BanaNomic* and its semantic treatment, we noted and explained the deviations from the original logic on which ours is based. In addition, limitations made on the game of Nomic itself are discussed, with a particular look at our simplified version of the rule proposal-voting system.

Chapter 6

The language of BanaNomic

This chapter covers the design of the CNL interface for BanaNomic*, based on the contract logic defined in the previous chapter. We first describe the design approach taken, mentioning a number of common problem areas in NLP systems and how they were handled. Following this, we then go into the design of the grammar in GF, with particular attention paid to the abstract and concrete syntaxes defined and their relation to the* BanaNomic *contract logic.*

6.1 Designing the CNL

Much like the paradigm of GF itself, the linguistic interface for *BanaNomic* was designed with a focus on linearising into English the types of clauses already defined in the contract logic (chapter 5) for the game. In other words, the CNL built was not designed in terms of its linguistic boundaries, but rather as a collection of phrase structures which map directly onto the various constructs of the formal contract logic.

6.1.1 Template-based linearisations

In the simplest cases, the CNL for *BanaNomic* takes a straightforward template approach, combining together a number of canned phrases to form a complete sentence. While very basic, this approach works well for a number of uncomplicated situations. Table 6.1 shows some examples of formal constructs which follow this straightforward linearisation.

The degree to which this template approach is suitable depends on the natural language in question, and the level of linguistic correctness to which one wishes to adhere. For example, by taking this approach our CNL produces the following phrase:

Expression	$\mathbb{F}\ (\text{“}George\text{''} : ClimbDown)$
Linearisation	player “George” is forbidden to climb down the tree

Expression	$\mathit{Action} :: \mathit{ThrowBanana}\,(\mathit{PlayerName})$
Linearisation	throw a banana at ___
Expression	$\mathit{DeonticExp} :: \mathbb{O}\ (\mathit{PlayerName} : \mathit{Activity})$
Linearisation	___ is obliged to ___
Expression	$\mathit{Contract} :: \Box\,[\mathit{Time}, \mathit{Time}]\,\mathit{Contract}$
Linearisation	At all times between ___ and ___ ___
Expression	$\mathit{Contract} :: \mathit{Contract} \lhd\!\lhd\ \mathit{Query} \rhd\!\rhd\ \mathit{Contract}$
Linearisation	If ___ then ___ otherwise ___

Table 6.1: Examples of straightforward canned text linearisations.

While a more natural way of saying this would be to use "forbidden *from climbing* down", it cannot be disputed that the former is nevertheless completely understandable and within the limits of acceptability for a controlled language.

This same limitation also applies to inflections for noun number (i.e. singular/plural), for example:

Expression	$\mathit{Contract} \lhd\!\lhd\ (\mathit{Points}\,(\text{“}\mathit{John}\text{”}) > 1)\ \rhd\!\rhd\ \varepsilon$
Linearisation	if player "John" has more than 1 bananas then ___

Correctly handling such cases requires varying levels of complexity. The English plural example above is a near-trivial case, but more involved linguistic processes such as verb conjugation for aspect and tense would of course require a more in-depth treatment.

6.1.2 Semantic disambiguation

A slightly more complex situation which is to be handled by the *BanaNomic* CNL is that of context-sensitive linearisation. As mentioned in § 5.3.5, the *GenericPlayer* category may refer to either *every player* or *any player*, depending on the context where it appears in the contract clause. This is demonstrated in the statement:

Expression	$\mathbb{P}\ (\mathit{GenericPlayer} : \mathit{ThrowBanana}\,(\mathit{GenericPlayer}))$
Linearisation	*every* player is permitted to throw a banana at *any* player

Note how the former and latter occurrences of *GenericPlayer* are linearised differently, as they occur in different contexts in the clause. This example also demonstrates the limited existential and universal quantification capabilities available in the CNL (see § 6.1.4). For more detail on how this is achieved using GF, please refer to § 7.5.2.

6.1.3 Punctuation

Punctuation is one language aspect which shall not be covered by the CNL designed for this project, and is in fact completely omitted from the language grammar. The main reason for this is to simplify the text input process for the user. The types of phrases generated in *BanaNomic* are generally not too long and can get away with having no punctuation; however for larger projects this would be an important addition to the CNL layer.

6.1.4 Problem areas

Certain kinds of linguistic phenomena often create problem areas for the designers of NLP systems, including pronouns, quantifiers and the handling temporal notions. These are briefly discussed below, along with how they were tackled (or avoided) in the *BanaNomic* CNL.

Pronouns

Pronouns are words which are used in place of nouns to refer to some previously mentioned entity (known as the antecedent). For example, in the sentence[1]

> *The monkeys ate the bananas because they were hungry.*

the pronoun *they* is used in place of "the monkeys". The problem with pronouns is that the person or object to which they refer is usually implied from some higher-level knowledge about the situation, which often leads to ambiguous situations. Consider the sentences:

> *The monkeys ate the bananas because they were ripe.*
> *The monkeys ate the bananas because they were in the tree.*

In the first case, we would generally infer that *they* is now referring to "the bananas"; but how did we make that deduction? The fact is that determining which antecedent a pronoun refers to (or *anaphor resolution*) often requires a high degree of outside knowledge, which is notoriously hard to capture in a formal system. The second sentence is an even worse case, because now it is no longer clear—even to us—who or what *they* refers to.

Pronouns are a prime source of problems in NLP systems, however some CNLs do provide limited support for them (Fuchs et al., 2008; Schwitter, 2002). In our case, pronouns are simply not handled by the grammar and all references to people or object nouns are explicit.

Quantification

Quantifiers are a class of determiners which indicate quantity, such as "some", "most", "every" and so on. The use of quantification in natural languages is another problem area for NLP, again due to the ambiguities it often causes. For example, in the sentence[2]

[1] Example taken from `http://en.wikipedia.org/wiki/Anaphora_(linguistics)#Examples`
[2] Example taken from `http://en.wikipedia.org/wiki/Quantification#Natural_language`

Someone gets mugged in New York every 10 minutes.

it is not technically clear whether or not it is the *same* person who is getting mugged. As with pronouns, we generally solve such ambiguities with the help of external information or "general knowledge"; but there is no direct way of doing this formally.

Despite this potential for ambiguity, restricted quantification has been successfully implemented in some controlled languages (Fuchs et al., 2008; Schwitter, 2002). Our own *BanaNomic* CNL also includes the existential and universal quantification operators, which are linearised as *any player* and *every player* respectively (see § 6.1.2 above).

Temporal notions

One final problem area which we shall look at is the various notions of temporality used in natural language. These include:

Absolute time *My birthday is on the 14th of July.*

Relative time *Your birthday is three days after mine.*

Duration *The party will be two hours long.*

Sequence *Make a wish and then blow out the candles.*

Concurrency *Blow out the candles while we sing.*

Dependency *You can open your presents after cutting the cake.*

Each of these examples represents a specific way of reasoning about time and the relationship between events. The CNL designed for this project specifically covers the temporal notions defined in the underlying contract logic—i.e. *before*, *between*, *after*, *at some point*, *at all times* and *concurrently* (refer to the logic as defined in chapter 5). These notions are included in the CNL in a rigid manner, in that they can only be used in the precise ways defined by the logic. For example, the sentence

Player "John" is obliged to concurrently enact a new rule and climb up the tree.

is a valid *BanaNomic* statement, but

Player "John" is obliged to enact a new rule while climbing up the tree.

is not considered valid, despite having the same meaning.

6.2 Grammar design in GF

In this section we explain the grammar design using GF's abstract/concrete syntax system (see § 3.4.1). More complete examples of the grammar source code can be found in appendix D.2.

6.2.1 Abstract syntax

Given the declarative nature of GF grammars, the abstract syntax of *BanaNomic* could very easily be implemented from its formal logic. For example, the abstract GF equivalent for the definition of the *Contract* category (see § 5.1) would be as follows:

```
cat
  Contract ; [Contract]{2} ;
fun
  C_Empty       : Contract ;
  C_Deontic     : DeonticExp -> Contract ;
  C_Choice      : [Contract] -> Contract ;
  C_Always      : Time -> Time -> Contract -> Contract ;
  C_Sometimes   : Time -> Time -> Contract -> Contract ;
  C_Conditional : DeonticExp -> Contract -> Contract -> Contract ;
  C_Query       : Query -> Contract -> Contract -> Contract ;
```

Note how the `Choice` constructor is not defined between just two contracts, but over an indeterminate number of contracts represented as a list, i.e. `[Contract]`. The same applies for the choice and concurrency operators for the *Activity* category. The rest of the abstract grammar is implemented in a similar way, closely matching the contract logic defined in chapter 5.

One point of interest is that while the *Enact* and *Abolish* actions both take arguments in the contract logic, in the linguistic case we do not require them to, and they are always treated as generic actions:

```
cat
  Action ;
fun
  An_ClimbUp  : Action ;
  ...
  An_Enact    : Action ;
  An_Abolish  : Action ;
```

Abstract syntax examples

To show the direct conversion from formal contract logic to GF abstract syntax, table 6.2 re-lists the logic examples from § 5.2 along with their equivalents in GF syntax tree representation.

6.2.2 Concrete English syntax

For the design of the concrete grammar, each of the functions from the abstract syntax was first given a template-like linearisation as described in § 6.1.1. While suitable for many cases, certain constructs required a more involved approach in order to produce natural linearisations. A good example of this is the timely contract clauses, `C_Always`

Formal	$\mathbb{O}\,(\text{“Paul”} : \mathit{ThrowBanana}\,(\text{“Ringo”}))$

Abstract GF	`C_Deontic`
	`(DE_Obliged "Paul" (A_Action (An_ThrowBanana "Ringo")))`

Formal	$\mathbb{P}\,(\text{“John”} : \mathit{PickBanana}) \lhd\lhd (\mathit{Points}\,(\text{“John”}) < 5) \rhd\rhd \varepsilon$

Abstract GF	`C_Query (Q_PointsLt "John" 5)`
	`(C_Deontic (DE_Permitted "John" (A_Action An_PickBanana)))`
	`C_Empty`

Formal	$\varepsilon \lhd \mathbb{P}\,(\text{“George”} : \mathit{Enact}) \rhd \Diamond\,[\varepsilon, 9]\,\mathbb{O}\,(\text{“George”} : \mathit{Abolish})$

Abstract GF	`C_Conditional (DE_Permitted "George" (A_Action An_Enact))`
	`C_Empty`
	`(C_Sometimes T_None (T_Time 9)`
	`(DE_Obliged "George" (A_Action An_Abolish)))`

Formal	$\Box\,[3, \varepsilon]\,\mathbb{F}\,(\phi : \mathit{ClimbUp}\ \&\ \mathit{PickBanana})$

Abstract GF	`C_Always (T_Time 3) T_None`
	`(DE_Forbidden "GENERIC" (A_Concurrent (BaseActivity`
	`(A_Action An_ClimbUp) (A_Action An_PickBanana))))`

Table 6.2: Contract logic examples from § 5.2 and their equivalents in GF tree representation.

and `C_Sometimes`. In both of these clauses, either of the two *Time* parameters may or may not be set. Each of these cases carries a different semantic meaning, and therefore requires a particular linguistic linearisation. This behaviour—which is determined at runtime—is summarised in table 6.3 for the `C_Always` case.

Abstract Clause	Linearisation
`C_Always (T_None) (T_None) ...`	*at all times …*
`C_Always (T_Time 1) (T_None) ...`	*at all times from 1 …*
`C_Always (T_None) (T_Time 9) ...`	*at all times before 9 …*
`C_Always (T_Time 1) (T_Time 9) ...`	*at all times between 1 and 9 …*

Table 6.3: Example of different possible linearisations of a single lexical function.

6.3 Chapter summary

By starting out with a simple template-based approach to CNL design, we discussed the common issues involved in designing NLP systems and explained how the basic template approach could be supplemented to accommodate these requirements and achieve an acceptable degree of 'naturalness'. We then went into the design of the grammar in GF, demonstrating the similarities between the abstract syntax and the *BanaNomic* contract logic, and explaining the English linearisations of the grammar through the concrete GF syntax.

Chapter 7

Implementation

Starting with an overview of the completed system, we look at each major module in detail and discuss the implementation choices made, the technologies used and any particular points of interest. Finally, the unforeseen issues which arose during the implementation phase are discussed, along with how they were dealt with.

7.1 System overview

The the design of a working implementation of *BanaNomic*, the following system components were identified:

1. A contract logic evaluator which would process a game's rules and determine the validity of player actions.
2. A CNL grammar and library which would handle the parsing and generation between the formal contract logic and English.
3. A game application which would keep track of game states and provide an interface for users.

Since *BanaNomic* is a game which needed to be accessed by various users simultaneously for evaluation purposes, the obvious choice was to implement the game as a web application, hosted on the internet for all to access. This imposed a client-server architecture on the system, which is outlined as a block diagram in figure 7.1. Each major system module is described individually in the following sections.

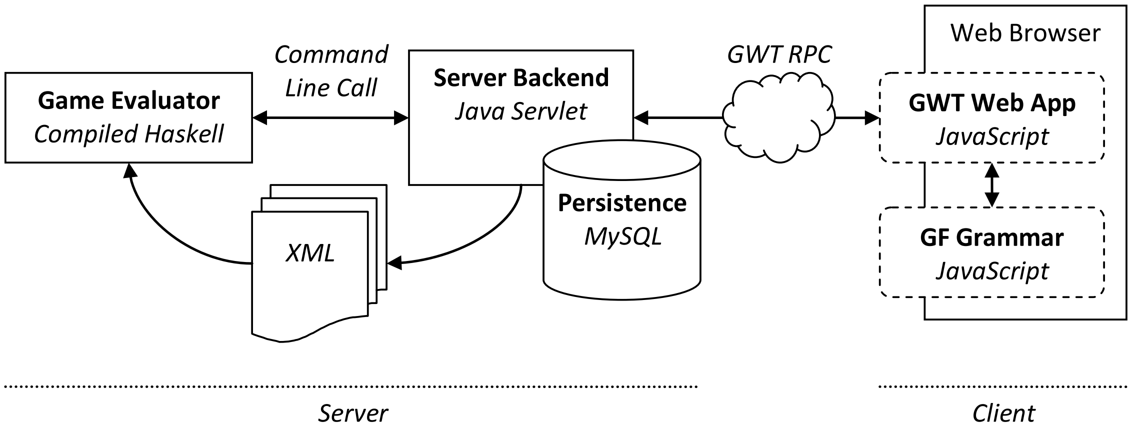


Figure 7.1: System block diagram.

7.2 Client-side web application

7.2.1 Google Web Toolkit

The Google Web Toolkit (GWT)[1] is a set of development tools for building modern, browser-based, AJAX-intensive web applications. It allows developers to write their application using a subset of Java, which is then converted by the GWT into cross-browser compatible JavaScript which is run entirely on the server. It also provides the ability to make asynchronous server calls (known as GWT RPC) for an AJAX-rich experience. GWT is used by Google itself in applications such as Google Wave[2], as well as by the GF developers in their sample applications distributed with the framework (see § 3.4.3).

While it is possible to use GWT to build only the client side of a web application, it was also used for developing the server backend for this project. While server calls still need to be made asynchronously through GWT RPC; the advantage of this setup is that plain old Java objects (POJOs) can be passed directly from client to server by making use of Java serialization. In this case the server-side code is *not* restricted to GWT's subset of Java.

The *BanaNomic* application was developed using the most recent version of the toolkit—GWT 2.0—and the Google Plugin[3] for Eclipse[4].

7.2.2 Grammatical Framework

GF is used in *BanaNomic* for providing a natural language (NL) interface to the formal contract logic (CL). This involves text generation/linearisation ($CL \rightarrow NL$) and parsing

[1] `http://code.google.com/webtoolkit/`

[2] `http://wave.google.com/`

[3] `http://code.google.com/eclipse/`

[4] Eclipse 3.5. `http://www.eclipse.org/`

($NL \rightarrow CL$), both of which are concerned primarily with the user. In other words, all the NL generation and parsing is done for the user interface. Because there is no internal processing which is done in terms of the natural language, all code associated with it can in fact live on the client side of the application.

This use of GF solely on the side of the client is made possible through the framework's inbuilt capability for compiling GF grammars directly into JavaScript. By including the GF JavaScript library which is distributed with the framework, all the language tools required—namely parsing and generation—became available to the rest of client application through GWT's JavaScript Native Interface (JSNI).

Input methods

The guided input methods developed for *BanaNomic* are closely based on those described in § 3.4.3, renamed slightly to the "suggest panel" and "fridge magnets" methods. While the features were re-coded from scratch for this project, the source code distributed with the GF framework was nevertheless consulted. One major difference in the implementations is that while the GF versions use a separate web service for generating the completion suggestions[5], in *BanaNomic* everything was kept at the client side. This would allow for a simpler server setup, while also avoiding making numerous, slow RPC calls each time the input methods were used.

It turned out that the GF JavaScript library did not include the `complete()` function required, and this ended up being implemented by the authors of this project. The JavaScript implementation of this function has since made it into the GF code repository and subsequent distributions of the framework (for source code refer to appendix D.2.2).

Grammar conversion

Both the game evaluator (written in Haskell, see § 7.3.2) and the CNL grammar (written in GF) use the same formal contract logic, as defined in § 5.1. Since the GF syntax is closely based on that of Haskell, the possibility of directly using the same logic code or implementing some automatic conversion from Haskell to GF notation was investigated. However, despite the similarities in notation, it was concluded that attempting this would be outside the scope of the project, especially for a relatively small grammar as ours. Thus, the conversion from the contract logic in Haskell to GF abstract syntax was carried out manually (the source code of both may be found in appendix D).

[5] This is done by compiling into PGF (portable grammar format), and then running this as a web service using FastCGI. Details at `http://www.grammaticalframework.org/doc/gf-tutorial.html#toc159`

Haskell API

The GF package comes distributed with a Haskell API, for accessing GF functionality directly from a Haskell application. Since the contract logic in *BanaNomic* is also implemented in Haskell (see § 7.3.2), using this API was a potential option. However in our case the CNL generation and parsing only needed to happen on the client side—in particular to have access to the guided input systems—and thus our choice was ultimately to use GF grammar compiled as JavaScript.

Multilingual features

The multilingual capabilities of GF were not in fact exploited in this project, since the CNL for *BanaNomic* was only developed in English. However the use of GF would mean that additional languages could very easily be added if the need were to arise in future developments.

7.3 Server-side backend & game evaluator

7.3.1 Java backend

The server backend is the part of the system which ties all other modules together. By taking care of all the application's persistence needs (see below), this module stores and manages all user and game data in the system. On the one hand, the backend handles all RPC calls from client web browsers—which would typically include authentication (logging in/out), retrieving current game data, submitting players' turns, and recording user feedback. On the other hand, the backend also invokes the game evaluator (see § 7.3.2) as needed by calling the executable through the command line.

Format conversions

In order to interface with the evaluator—which communicates gamestate information encoded as XML—the backend also handles the conversions between POJOs, GF notation, and the specified XML scheme (for an example refer to appendix D.1.2). Specifically, these operations are handled by two separate program modules, for converting XML ↔ GF and XML ↔ POJO respectively. It should be pointed out that these conversions are not necessarily total, and the POJOs created often contain class members which contain the corresponding XML as a string. Similarly, GF representations are only generated for certain classes, where necessary; it does not make sense to convert an entire gamestate into GF notation as this is never required.

Persistence

The application's storage needs were met using the EclipseLink[6] Java Persistence API library. This approach allows POJOs to be persisted directly to a database, without the developer having to get into the conversion from the object-oriented (OO) model to the relational one. The database server used was MySQL Community Server[7].

7.3.2 Haskell game evaluator

The game evaluator is arguably the heart of *BanaNomic*, as it is the part of the system which evaluates the game's rules and determines the validity of player actions. The evaluator is called in an *ad hoc* manner and has no persistence component or storage of its own. For any given invocation, the evaluator is passed an entire gamestate (in XML format; refer to appendix D.1.2) which contains one or more "scheduled" events, which would typically be a player's attempted turn. Each of the events in the schedule is processed in sequence. When the event is accepted its effects on the gamestate are affected (such as changes in player points), and the next scheduled item is processed until the schedule is empty, at which point the updated gamestate is returned to the caller (again, as XML). If *any* of the scheduled events happens to violate the gamestate (e.g. attempting to carry out a forbidden action), then the entire transaction will fail and the evaluator will return an error message to the caller.

Choice of Haskell

The game evaluator was written in Haskell[8], defining the contract logic for the game as an embedded language within the program. Haskell has been shown to be highly suitable for this purpose (Pace and Rosner, 2010), by conveniently allowing the embedded language to be defined declaratively, like so:

Formal Logic

$$\begin{aligned} \mathit{DeonticExp} = {} & \mathbb{O}\ (\mathit{PlayerName} : \mathit{Activity}) \\ \mid {} & \mathbb{P}\ (\mathit{PlayerName} : \mathit{Activity}) \\ \mid {} & \mathbb{F}\ (\mathit{PlayerName} : \mathit{Activity}) \end{aligned}$$

Haskell Embedded Grammar

```
data DeonticExp =
   DE_Obliged   Player Activity
 | DE_Permitted Player Activity
 | DE_Forbidden Player Activity
```

This approach benefits the developer by "enabl[ing] the use of abstraction and modularization techniques from the host language in the embedded language" (Pace and Rosner, 2010). While all classes and operators defined as part of the embedded language can be treated purely at a domain-specific level, they also enjoy the benefits of

[6] EclipseLink 2.0.0. http://www.eclipse.org/eclipselink/

[7] MySQL Community Server 5.1.41. http://www.mysql.com/downloads/mysql/

[8] Haskell Platform 2009.2.0.2. http://hackage.haskell.org/platform/

being implemented as Haskell objects. Thus, features like Haskell's powerful pattern matching and dynamic typing can be used directly with classes defined in the embedded language. These features and the functional paradigm of Haskell make it an ideal choice for the implementation of an embedded grammar along with the tools needed for its execution and analysis.

XML interface

As mentioned previously, input and output from the game evaluator is encoded as XML according to custom schema (see appendix D.1.2). For input, this XML is written to a temporary file and the evaluator is passed the corresponding file path. The resulting XML output is written straight to the program's standard output stream. Internal conversion between XML and native Haskell data structures was handled by the hexpat-pickle[9] Haskell library, which uses the Expat[10] XML parser library.

7.4 Implementation notes

7.4.1 Hosting setup

For its evaluation period (see chapter 8), *BanaNomic* was hosted on the internet from a personal machine running Windows XP SP3. The web server was run using Apache Web Server[11] with Tomcat[12], as distributed together with MySQL Community Server[13] in the XAMPP[14] package. The game evaluator written in Haskell was compiled into an executable file and called via the command line from the server backend. The GF grammar was compiled into JavaScript and served to clients to be run locally. A custom domain name was purchased, and the project hosted at `http://bananomic.net`.

7.4.2 Web applications & user interface

The major reason for deciding to develop *BanaNomic* as a web application was ease of access and distribution. Being a multi-player game, the requirement for a central server to administer player activity is unavoidable. The application *could* have been developed as a desktop one which users would download and run locally, however the added development effort for doing this would not have been justified. That being said, the development of the web application and the user interface (UI) design did themselves take up a considerable portion of the overall development time.

[9] hexpat-pickle 0.4. `http://hackage.haskell.org/package/hexpat-pickle`

[10] Expat 2.0.1. `http://expat.sourceforge.net/`

[11] Apache Web Server 2.2.14. `http://httpd.apache.org/`

[12] Apache Tomcat 6.0.20. `http://tomcat.apache.org/`

[13] MySQL Community Server 5.1.41. `http://www.mysql.com/downloads/mysql/`

[14] XAMPP 1.7.3. `http://www.apachefriends.org/en/xampp-windows.html`

As the game was to be playable by a diverse audience, user interface issues could not be ignored. The advantage of implementing a web-based solution is that users would already be familiar with a number of UI concepts which are commonplace on the web. These include tabbed layouts, tool-tips, check boxes, select boxes, auto-completion and user logins. While maximum use was made of such concepts, other common website functions were omitted for lack of time and irrelevance, such as user profile management and inter-user communication.

7.5 Development issues & unforeseen problems

7.5.1 Logic

Time waits for no one

The implementation of time may be handled in a number of different ways. One option could associate a time value for each action, indicating how long it takes to complete. This approach is relatively realistic, however introduces of host of problematic situations which would require correct handling. For example, what happens if an obligation exists to perform an action every day, but the action itself takes longer than a day to complete?

In our design, we opted for a simpler approach where performing actions does not take any actual time; i.e. actions are considered completed the instant they are carried out. As a result, actions can be performed together without needing to consider their durations. In *BanaNomic*, the passage of time is governed by a global clock which is synchronised to increment once every 24 hours, and there is no limit on the number of actions that can be carried out in any given time frame.

Concurrent actions

As part of a normal turn, players in *BanaNomic* may perform multiple actions concurrently; for example, climbing up the tree and picking a banana in one go. Unfortunately in the implementation of the game, no effective way of processing actions in a truly concurrent fashion could be found. As a result, concurrent player actions are still internally processed in sequence. This specific sequence is defined by the program code, and is made known to users as the order in which the turn options appear in the interface.

Conditional clauses

A conditional clause is one in which the resulting contract is dependent on the existence of a given deontic expression. For example:

> *if player "Paul" is permitted to climb up the tree then*
> *player "John" is obliged to climb down the tree*

In such cases, the former deontic expression must be evaluated to determine if the latter should be enforced. Unfortunately, the way that this evaluation is performed in *BanaNomic* is somewhat limited, as in order to avoid arbitrary recursion all other conditionals encountered in the evaluation are omitted. Thus, a clause such as the following would *not* be evaluated as intended:

> *if player "Paul" is permitted to climb up the tree then*
> *if player "George" is forbidden to pick a banana then*
> *player "John" is obliged to climb down the tree*

It may be argued that these types of phrases would not be that commonplace or even desirable, since such recursive clauses generally do not sound right when expressed in natural language. However it is a limitation of the system that—at least from a formal point of view—they are not correctly processed internally.

Enforcement of obligation and prohibition

During the implementation stage, various difficulties were met when working on the enforcement of player obligations. This proved to be a rather tricky task, in particular when dealing with obligations with sometimes/always clauses (see § 5.3.4 for the intended behaviour). The initial solution turned out to be rather flawed; player obligations could be viewed but not enforced, and some bugs in the code caused obligations which were already satisfied to re-appear when they shouldn't have.

These issues were initially unknown to the authors and were in fact present in the application when it was first released for evaluation. However after their discovery mid-way through the evaluation period, the necessary fixes were made and pushed to the live version of the game. These fixes produced the correct behaviour in terms of enforcement of obligations, except in the case of those within *sometimes* clauses. The issue in this case was that as it is left up to the user when to fulfill said obligations, they could in fact be left unsatisfied forever. For more information about this see § 8.4.1.

Finally, it should be noted that player obligations in the game are handled strictly by *enforcement*, meaning that if an obligation needs to be satisfied then the player involved will simply not be able to play without satisfying it. This also applies to prohibition, making it impossible for users to perform actions which they are not allowed to do. This method of enforcement excludes the possibility of more versatile violation handling, where arbitrary repercussions could come into effect for not fulfilling one's obligations or carrying out prohibited actions.

Command-line string length limitation

As described in § 7.3.2, when the game evaluator is invoked it is passed an XML representation of the current gamestate. In the initial version of *BanaNomic*, this XML data was passed directly as a command-line argument through the standard input stream of the process. However as the evaluation period went on and the games progressed, their corresponding gamestate representations also grew until eventually the 8191-character limit[15] on the length of command-line strings was reached. The input XML data was being truncated, which caused the evaluator process to wait endlessly for the rest of the input and meant that the application would hang when players attempted to play their turns.

Once this issue was noted and its source identified, changes to the code were quickly made such that the XML data was instead written to a temporary file. Now, the game evaluator process would only need to be passed the *path* to that file, rather than the entire XML string. The changes were then pushed to the live version of *BanaNomic* without interrupting the games in progress.

Haskell & the functional paradigm

In developing the game evaluator, the Haskell language was chosen for its suitability for defining declarative grammars and working with them. However, the functional programming paradigm is not without its trappings. For one thing, certain tasks which are traditionally imperative require a total re-thinking when using a functional language, in particular file I/O and program flow. Additionally, source code in Haskell tends to be very cryptic to read and therefore much greater care must be taken to ensure that it remains understandable when re-visited. Finally, while debugging tools for Haskell do exist[16], they are quite complicated to use and in reality are no comparison for the powerful debugging tools available for modern OO languages.

7.5.2 Language

Parametric categories

As noted in § 5.3.5, the effective meaning of the so-called generic player varies with the context in which it appears. To be able to handle this in the CNL implementation, GF's *parametric categories* were used. Consider the following extract from the **abstract** grammar:

```
cat
```

[15] As described in Microsoft Knowledge Base article 830473, at http://support.microsoft.com/kb/830473

[16] In this project the Glasgow Haskell Compiler (GHC)'s interactive debugger was used (version 6.10.4). More information at http://www.haskell.org/ghc/

```
    Player ;

fun
    P_Generic   : Player ;
    P_Player    : String -> Player ;
```

This simply declares the `Player` category and two constructor functions for generic and specific players. Then, as shown in the following excerpt from the **concrete** grammar, we declare that the linearisation of a `Player` varies according to the `PlayerParam` passed to it:

```
param
    PlayerParam = PP_Any | PP_Every ;

lincat
    Player = { s : PlayerParam => Str } ;

lin
    P_Generic = { s = table {
        PP_Any => "any player" ;
        PP_Every => "every player"
    } } ;

    P_Player name = { s = table {
        _ => "player" ++ name.s
    } } ;
```

Note how in the case of the specific player `P_Player` the linearisation is always the same, regardless of the `PlayerParam` used.

Null productions

As can be seen in § 5.1 the *Time*, *Contract* and *Activity* categories all have a null-production ε, which symbolises the empty instance of that category. In both the logic and language grammars these are represented by the constructors `T_None`, `C_Empty` and `A_Empty` respectively. These null cases are useful during logical analysis, however in the language case they present a bit of a problem. Specifically, trying to linearise them as the empty string “” will cause parsing problems because in certain cases the parser would not be able to determine whether an empty string in the input should be parsed as a null production or the end of a sentence[17].

To avoid this, the null productions for the *Contract* and *Activity* categories were both linearised to the string “(nothing)”. In the case of `T_None`, this was not necessary as *Time* items can never appear at the end of a phrase.

[17] The lack of punctuation in the CNL grammar also contributes to this problem.

7.5.3 User interface

Browser issues

The *BanaNomic* application was developed using Google Web Toolkit which allows the building of AJAX-rich web applications while writing code purely in Java. During development, the site was tested under the Firefox[18] and Chrome[19] browsers on Windows 7. While GWT may aim to provide compatibility across all modern browsers out of the box, in the evaluation of the application a number of users experienced difficulties in accessing and/or playing *BanaNomic*.

Initially some of these issues were found to be bugs in the code. However even after these were fixed, some user's problems still persisted. To confound the issue, the problems were not simply browser incompatibilities, and the site would not work correctly on certain machines irrespective of which browser was used. These issues are suspected to be related to a user's security settings, firewall, and anti-virus software.

However, achieving perfect cross-browser and cross-machine compatibility was not a priority in this project, and since the root of these problems could not be determined, some players were unfortunately forced to used different browsers and/or machines for accessing the site.

7.6 Chapter summary

This chapter covers the implementation of the system built for this project, by first looking at the overall architecture and then explaining in detail each of its major modules. We discuss the technology choices made during the process, in particular the use of GF for the language interface and Haskell for implementing the contract grammar as an embedded language. We then conclude with a number of development issues which arose, and how they were handled.

[18] Mozilla Firefox 3.6. `http://www.mozilla-europe.org/en/firefox/`

[19] Google Chrome 4.1. `http://www.google.com/chrome`

Chapter 8

Evaluation

This chapter covers the evaluation of the BanaNomic *system built. Based on the project aims in chapter 1, we first consider what it is we want to evaluate, and define the evaluation methodology to be used. We then describe the details of how the evaluation was carried out, and present our summarised results. Finally we conclude with an in-depth discussion of these results, the overall reaction to* BanaNomic, *and some possible limitations in the methodology used.*

8.1 Methodology

8.1.1 What answers are we seeking?

In accordance with the aims and objectives in 1.3, the evaluation stage of this FYP is intended to investigate the effectiveness of the methods used in interacting with a contract logic using a CNL representation. Specifically, measures of the effectiveness of the following aspects of the project are desired:

1. The expressivity of the grammar and its suitability for the given application.
2. The naturalness of the generated phrases.
3. The ease-of-use of the guided natural language input methods.

8.1.2 The questions we asked and how

One of the main reasons for implementing this project as a game was to be able to provide an accessible platform through which test users could interact with the system and provide feedback while being entertained. In order to obtain the answers we were looking for, user responses were obtained through *in-game feedback* and a *post-game questionnaire*.

In-game feedback

Firstly, a group of evaluators played the game for a stipulated time, filling in a short feedback form with each turn they played. The questions asked were:

- How understandable are the rules of the game at this point?
- In this turn you added a new rule. How easy was it to express what you wanted to?
- For this turn, how well could you understand what was going on?

For each question, players could give a quality rating from 0–4, as shown in table 8.1. A comments box was also made available for any extra comments that players may have. In addition to the feedback provided by the player, the enactment of rules and the input method used were also recorded automatically. For a full list of the feedback received, please refer to appendix C.2.

Rating	Description
0	Poor
1	Weak
2	Average
3	Good
4	Great

Table 8.1: Available rating levels.

Post-game questionnaire

After the evaluation period was over, each player was sent a slightly longer questionnaire about their experience with *BanaNomic* on the whole, with an emphasis on the natural language aspect of the game. This consisted of a series of questions which asked for ratings from 1–5, as well as a couple of multiple choice questions. The full questionnaire, along with players' responses, can be found in appendix C.3.

8.1.3 Evaluation period

Recruitment

A total of 14 individuals were recruited to help evaluate the game (all were personal acquaintances of the author). Since the scope of the project was to create a natural language interface which is usable by the non-technical, care was taken not to choose too many evaluators from an IT background. In fact, only one of the 14 recruited was actually qualified in IT. It is, however, pertinent to note that all the evaluators have had some level of University education, with two even studying for postgraduate degrees.

Prior to evaluation each user filled in a short consent form, which obliged them to list their field of expertise and level of education. Users were given the option of remaining anonymous; accordingly some of the user names as listed in table C.1 are aliases, named after random celebrities.

Setup and duration of games

The 14 evaluators were split between two concurrent games of *BanaNomic*, playfully named *Banana Bonanza* and *Potassium Paradise*. Each game was run for nine consecutive days, where each player was able to play up to one turn per day (or pass if unable to play). At the end of each turn, players would be presented with a short feedback form to fill in (see appendix C.2). Not all players were able to play every day due to personal reasons, and the total number of feedback entries obtained was 79.

8.2 In-game feedback

8.2.1 Average overall ratings

The graphs in figure 8.1 show the overall ratings given by users during the game for the following criteria:

1. Quality of generated text
2. Ease of expressibility using the CNL input method
3. Overall comprehensibility of the game

The first row of graphs show the totals of all feedback items together. However since some players would have submitted more feedback items than others, it is possible that these totals may be somewhat skewed. Thus the second row of graphs in figure 8.1 show the totals for the average ratings *per player*.

8.2.2 User ratings over time

To investigate how users' ratings varied as the game progressed, the average rating value for each criterion was found for each day, and plotted in figure 8.2. These were *not* normalized per player. As the graphs show, there was no noticeable improvement or worsening of the ratings over time.

8.2.3 Enactment of new rules

As shown in figure 8.3, two thirds of all turns played during the evaluation period involved the enactment of a new rule. In all those turns, the suggest panel input method

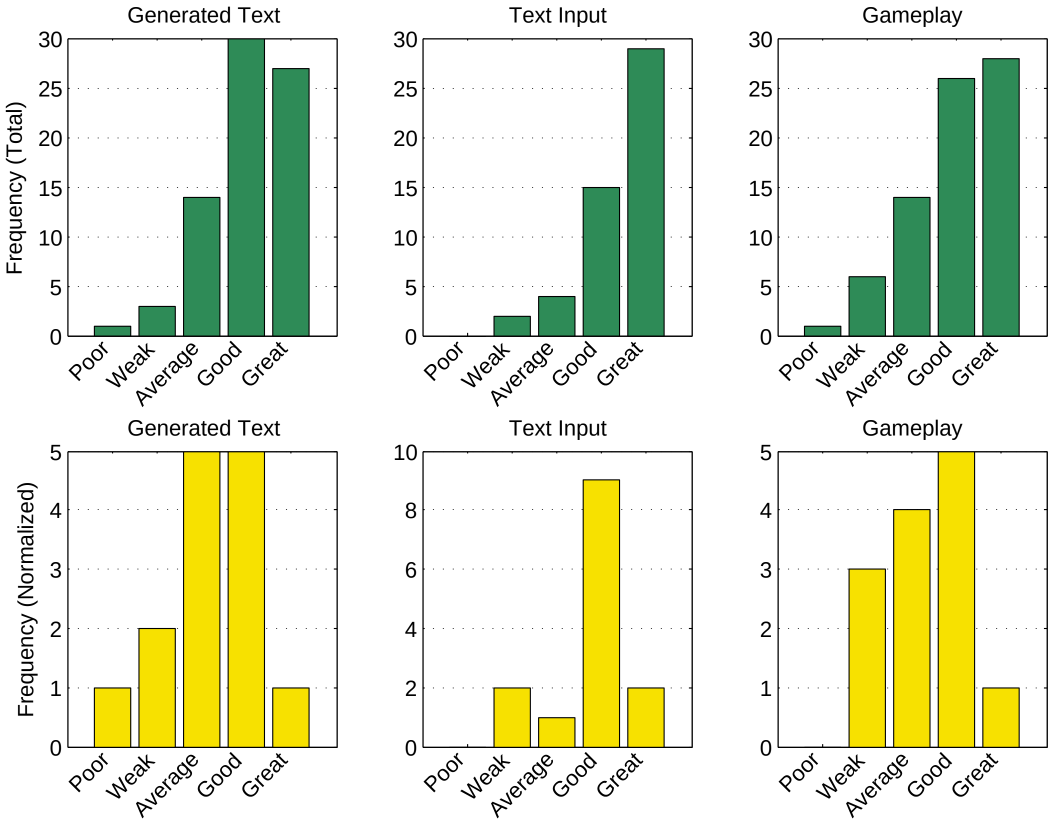


Figure 8.1: Overall in-game feedback ratings — Total (top row) and normalized per player (bottom row).

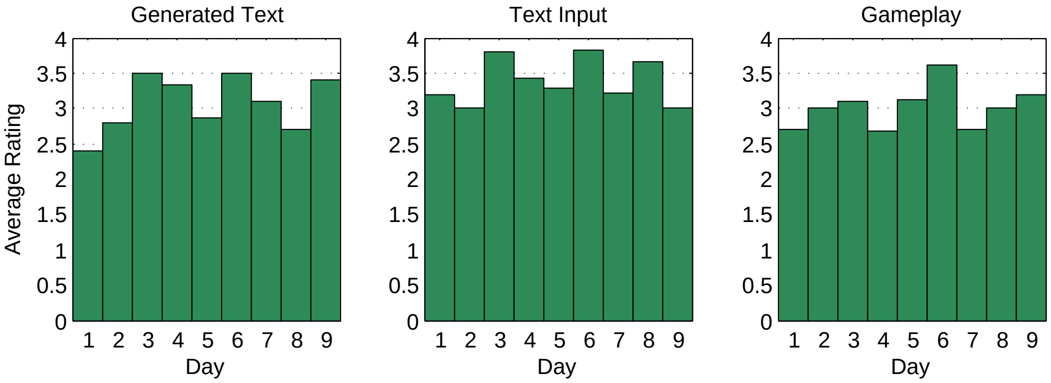


Figure 8.2: Average user ratings per day.

was the most commonly used text input method. However, it should be noted that this was the default input method shown to users when playing a turn, and using the fridge magnets method would require deliberately choosing it. A more thorough approach would have been to randomly choose the input method displayed to the user, and then track whether users bothered to change to their preferred input method or simply used whichever method was displayed first to them.

See also the first chart in figure 8.6, which shows the popularity of each input method based on what users named as their preferred choice in the post-game questionnaire (regardless of what they actually used during the game).

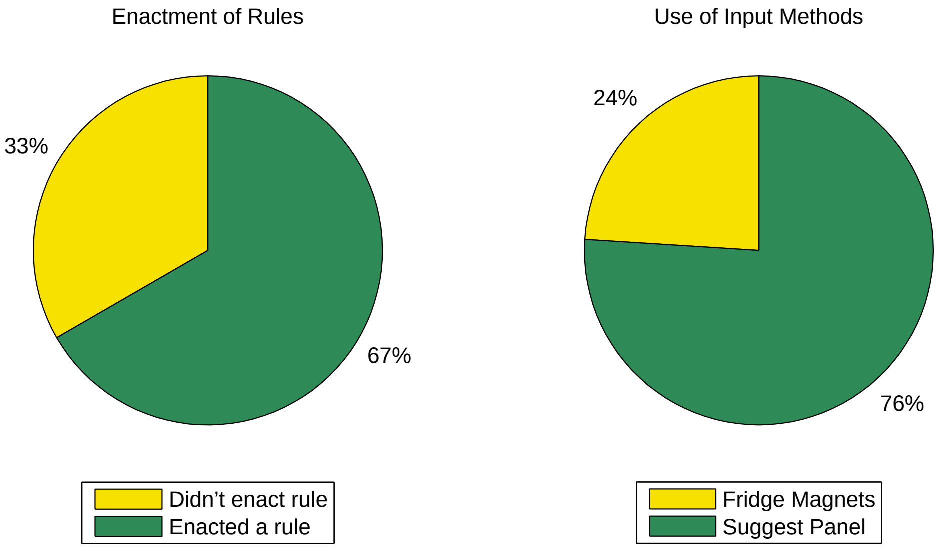


Figure 8.3: Enactment of new rules (as a percentage of total turns played) and input methods used.

8.2.4 Complete turns and passes

Given the self-altering nature of *BanaNomic*, one of the potential outcomes of a game is that users are rendered unable to play because of conflicting rules. Specifically for such situations, users were also given an option to pass their turn. As shown in figure 8.4, the number of players having to resort to passing was actually quite low at just 5 per cent.

8.3 Post-game questionnaire

Figures 8.5 and 8.6 show summaries of the data collected in the post-game questionnaire sent out to players. In all charts in figure 8.5 a higher x-value is desirable, except for the *Rule-Action Balance*, where a median value is ideal (implying the game was well balanced).

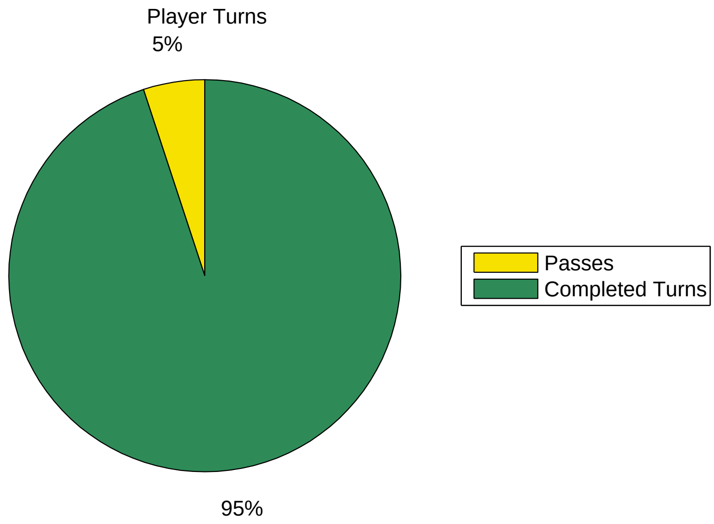


Figure 8.4: Complete turns and passes played.

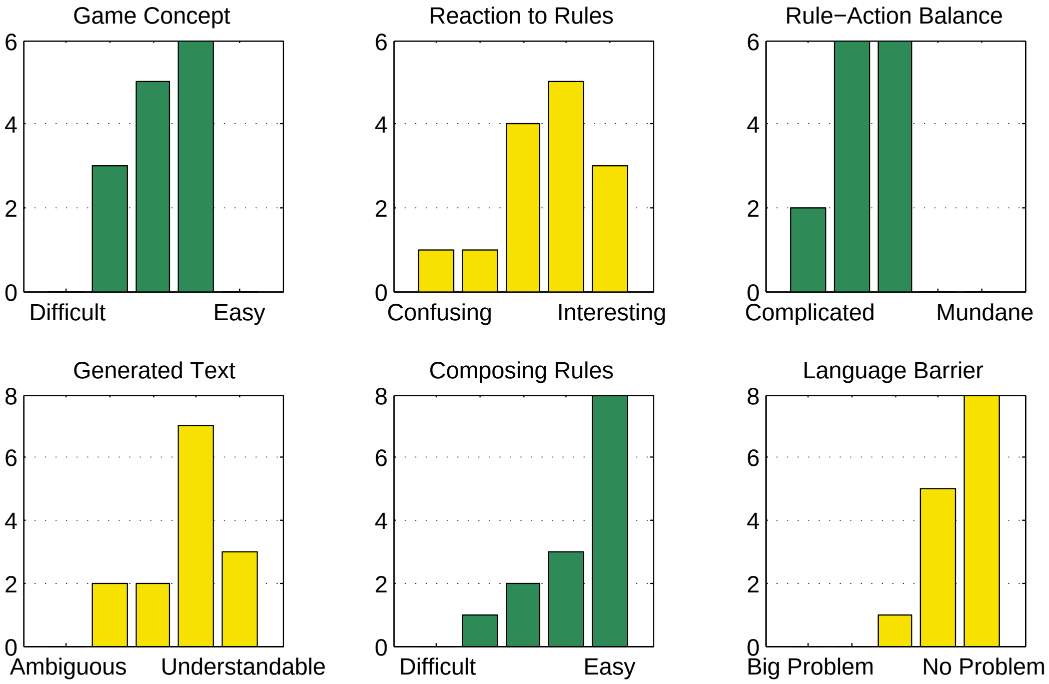


Figure 8.5: Summary of post-game questionnaire responses (see also figure 8.6).

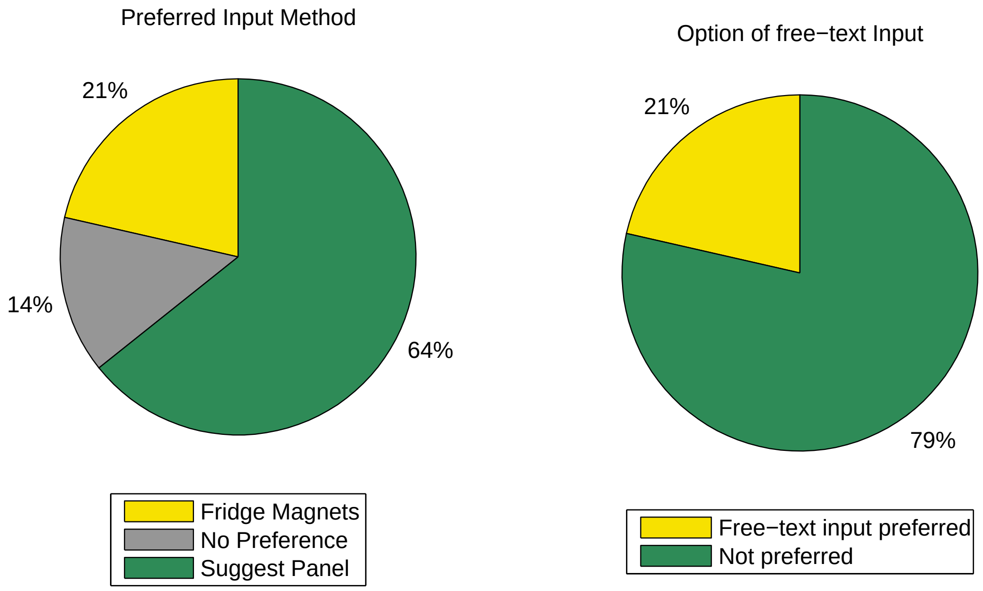


Figure 8.6: Post-game questionnaire responses about input methods.

8.4 Discussion

8.4.1 The contract logic and the self-amending game

As became apparent during the development of the project, trying to get feedback from users about the design of the contract logic is not an easy task. Since the scope of the project was to build a system which is accessible to non-technical people through the use of natural language, evaluators could not be asked directly about their views on the design of the contract logic. As a result, the feedback obtained was more about the game on the whole; however by analysing the players' comments we may identify areas in which the contract logic could be improved. The following observations were made.

Limited rule possibilities

Some users commented that once they were familiar with the idea of the game, the rule possibilities quickly started to seem limited. While this did not prevent them from manipulating the rules to their advantage, it is pertinent to note that these restrictions were quickly noticed. Thus were the contract grammar to be used for any sort of long-term project, it would need to be enriched in order to make it more expressive. Interestingly, certain language features were not used at all throughout the entire evaluation period, in particular the so-called deontic-conditional clause:

If Maureen is permitted to pick a banana, then . . .

Variety of actions

None of the user comments directly implied that there were not enough simple actions available. However since the actions available have a significant role in the general expressivity of the grammar, having a wider choice of actions would help contribute towards a more versatile and expressive grammar, as mentioned above.

Ease of rule manipulation

Another notable comment which users made was that the ability to instantly enact and abolish rules somewhat undermined the whole idea of having rules in the first place. As noted in the players' turn patterns, rules were regularly enacted to be used by the player themselves within that very turn. Similarly, players who were forbidden from carrying out a certain action would often simply abolish the rule prohibiting them to do so, and carry out that action within the very same turn

The initial intention was for rule enactment and abolition to happen democratically through a voting system, in true Nomic style. However this feature was eventually dropped in favour of a simpler direct system (see § 5.3.6), and to a certain degree these repercussions were anticipated.

In addition, the manipulation of rules is considered as a simple action, such that there is no way in which users can be prohibited from enacting a certain *type* of rule. For example, it is impossible in the grammar to express the following rule:

> *Ringo is prohibited from enacting a rule which forbids George to climb up the tree.*

Instead, only a generic rule such as the following may be enforced:

> *Ringo is prohibited from enacting a rule.*

Priority of prohibition

One user mentioned that the game had an overall sense of negativity to it, in that the best way of succeeding was ultimately to prohibit other users from performing whatever actions. This is a direct result of the precedence of prohibition over permission and obligation, which ultimately comes down to a design choice (see chapter 5).

Fallibility of obligation

In the first few days of the evaluation period, a number of bugs existed with the implementation of obligations and their satisfaction. This essentially meant that user's obligations were not enforced, and could in fact be completely ignored. Finally these issues were resolved and players were unable to complete their turns without fulfilling their pending obligations. However the implementation of obligations within *sometimes* clauses remained problematic; while the choice of *when* to fulfil the obligation was left up to the individual users (external/angelic choice), there was ultimately no final check to see if it had in fact been satisfied. In other words, *sometimes* obligations could remain pending forever, and thus never fulfilled.

8.4.2 Language aspect

While most of the feedback collected during the evaluation stage pertained to the natural language aspect of the game, it would seem that this was the area with which users had the least problems. Data from the in-game feedback and the post-game questionnaire show that both the generated text and CNL input methods were quite well-received, and did not act as a hindrance to understanding and participating in *BanaNomic*.

CNL input methods

Of the two input methods provided—the suggest panel and fridge magnets—the former proved to be the most popular with users as shown in figures 8.3 and 8.6. While they both provided access to exactly the same linguistic constructs, users found the suggest panel method "quicker" to use and "more streamlined" (taken from user comments; see table C.3). In addition, it is suspected that the mouse-only approach of the fridge magnets method proved to be a slight deterrent, in that all computer users are conditioned to enter natural language phrases through typing. However this explanation is mostly speculative.

Free-text input

As a point of interest, users were asked in the post-game questionnaire whether they would prefer a free-text input method rather than the ones provided. Interestingly, only three of the 14 respondents (21 per cent) answered that a free-text input would be preferable (see figure 8.6). Though one cannot generalise from just three responses, the feeling seemed to be that while the guided input was helpful *at first*, once the general structure of the language was learnt a free-text input would have been quicker and more natural. However the remaining majority said that they would *not* prefer such an unrestricted form of input, saying that:

> "the possibilities would have been endless and the rules more difficult to follow"

and

> "[when using the suggest panel] there was no need to worry about how to word the rules"

(Quotes taken from user comments; see table C.3).

8.4.3 Overall reflections on the game

Information overload

While on the whole the game showed some success within just 9 days, initially there was quite a lot of confusion over how things worked, in particular the rule amendment system. Users asked many questions to the author through informal channels about the how's and why's of *BanaNomic*, and although a full user guide was provided there was a general sense of information overload. Users expressed this repeatedly through their comments (see table C.3), saying:

> "I'm still not quite sure what I'm supposed to be doing!"
> "There is a huge amount of information and rules that needs to be taken into account ... which tends to make strategies for rule creation and use a bit obscure."
> "Overall, the game was quite hard to understand at first—mainly because I wasn't sure what I was allowed/not allowed to do"

One point to note is that even with the limited contract grammar used in *BanaNomic*, there was still a considerable amount of uncertainty and confusion. As such, the decisions to strip out the more complex aspects of the grammar was justified by users' difficulty in understanding the game.

Unhelpful error messages

A factor which also contributed to users' confusion was the generality of the error messages provided by the system. When playing their turn, players would often be told that their turn could not be accepted because of rule constraints. However the way the system was implemented meant that users could rarely be told exactly *why* their turn was not accepted; simply that it was not valid. This resulted in players having to make many attempts at their turn before successfully playing, and frequently playing whatever would be accepted—as expressed directly in comments such as the following (see table C.3).

> "For some reason I couldn't enact a new rule"
> "I had some obligations to fill and my turn wasn't accepted until I did them, the problem was that the error message did not tell me why my turn was not accepted, the error message was too generic."
> "I tried to enact the rule ... but it didn't let me, although I can't quite figure out why"
> "... after several attempts I managed it though not what I really wanted to do"

> "There are too many rules and counter rules. I ended up playing what it let me not what I wanted."

Too many rules

As the games in the evaluation period progressed, eventually so many rules existed that keeping track of them was close to impossible—at least for a human. Of course the whole point of *BanaNomic* was to create a system where the rules are automatically checked for the users, but given the unhelpful error messages as mentioned above this proliferation of rules also contributed to somewhat of an information overload for users. This problem was undoubtedly a result of the ease with which rules could in fact be created. A democratic rule amendment system—as initially intended—would have greatly diminished this problem of rule proliferation. However this was not technically achievable within the time available, and the relatively short evaluation period of nine days would likely not have been enough time to properly evaluate such a system.

Turn processing and player tactics

One of the implementational limitations of the project is that when a user's turn consists of multiple actions, these actions are actually processed *in sequence* rather than as a concurrent set of actions (see § 7.5.1). This order was pre-determined and in fact made known to users, who subsequently used it directly in their gameplay tactics. For example, the abolition of rules was processed before any other basic action, such as climbing up the tree. Thus, if a rule existed stating

> *Paul is prohibited to climb up the tree.*

then Paul would still be able to climb up the tree, as long as he abolished this rule within the same turn. Unfortunately, once this tactic was discovered by a player it meant that the circumvention of prohibitive rules was easier than it should have been. Some users even directly commented about this in their feedback, saying

> "[it was] easy to get around existing rules by abolishing an old one"

(Quote taken from user comments; see table C.3).

8.4.4 Limitations with the methodology

Embedded generated text

One of the primary purposes of the *BanaNomic* evaluation period was for users to provide qualitative feedback on the naturalness of the generated text elements of the game—i.e. the wording of the rules. Within the game, all instances of generated text—rules and obligations—always appeared in yellow boxes to distinguish them from other parts of the site whose wording was *not* automatically generated. While the both the in-game and post-game feedback forms specifically asked the users to evaluate the linguistic qualities of the game's rules, one cannot guarantee that they did not provide ratings based on the language of the entire *BanaNomic* site—including, for example, the user guide.

Limited evaluator group

With the limited time available for this project, the game evaluation could only be carried out by a single group of 14 individuals. These evaluators came from different professional backgrounds, yet all had some form of University education. For a more complete evaluation of any project, a larger sample size with a more varied spread of education levels would be needed in order to have more statistically significant results.

Evaluator bias and incentive

As with any statistical study, the possibility of bias is always present. In the case of this project, one notable source of potential evaluator bias is that all users involved in the evaluation were personal acquaintances of the author, either as relatives or as colleagues. Another important point which may have also skewed the results obtained is the relatively high education level of all the evaluators.

Finally, it should also be pointed out that the users evaluating this project had no particular incentive for doing so. The idea of developing a game did definitely help to keep users interested, however at the end of the day there was no real benefit for users. Thus there is always the possibility the feedback received was not totally genuine, and simply filled in 'to get it over and done with'.

8.5 Chapter summary

In this chapter we described the goals, method, and results of evaluation for this project. The qualitative results received during the evaluation period were provided through in-game feedback and a post-game questionnaire, and are summarised in the various charts provided along with explanations of interesting findings. We conclude the chapter with an in-depth discussion of these results and the comments from users about various aspects of the game. Possible limitations in the evaluation methodology used are also highlighted.

Chapter 9

Conclusions

In this final chapter, we look back at the progression of the project from inception to evaluation. The original project aims are re-visited and the degree to which they were met discussed. A number of shortcomings with the implementation are mentioned, and finally ideas and scope for future work are put forward.

9.1 Reflections on the project

9.1.1 Project overview

Motivation and aims

This project was borne out of the current interest in electronic contracts and the possibility of combining them with CNL techniques for real-world systems. The primary aim was to investigate the use of CNLs for contract logics and come up with some measure of the effectiveness of the approach. As a case study, the game of Nomic was chosen for its contract-like rule system and its interesting self-amendment features. These ideas were to be developed into a working version of the game where users could play against each other. Evaluation of the system would consist of qualitative feedback provided by players on the language interface and the underlying contract logic (chapter 1).

Design

Our own version of Nomic called *BanaNomic* was defined, set in a rainforest where each player is a monkey in a tree, competing to pick bananas and able to manipulate the rules of the forest to their advantage (chapter 4). A suitable contract logic was designed for the game (chapter 5), based on the *OPP-logic* as defined in Pace and Schneider (2009). The possible types of expressions in this logic were then given linearisations in English, which constituted a CNL interface for the logic (chapter 6).

Implementation

The system built for this project was implemented as a web application, using Google Web Toolkit (GWT) for building the user interface and handling AJAX requests on the server. Haskell was used for the game evaluator (which handles all the contract logic evaluation), while the CNL interface for the application was built using GF (Ranta, 2004). Some deviations from the original designs had to be made because of unforeseen difficulties in implementation. A number of system shortcomings were also identified at this stage (chapter 7).

Evaluation

Two games of *BanaNomic* were played by 14 players over nine days, where qualitative assessments were obtained through in-game user feedback and a post-game questionnaire. Once collected, these results were analysed and a number of observations on the success of the approach were determined (chapter 8).

9.1.2 Summary of results

From the results analysis in chapter 8, the following observations were made about the various aspects of the project.

The choice of a Nomic-like game as a case study for CNLs and contract logics proved a good choice and was well suited to the area of electronic contracts, yet still entertaining and interesting to users. While developing a Nomic variant was not one of the primary aims of this FYP, the use of a game for evaluation purposes was definitely beneficial; the excitement and competitiveness involved in playing a game helped to prevent users thinking of evaluating the system as a boring duty. *BanaNomic*'s self-amending nature was met with interest by the evaluators, although many did express initial difficulties with understanding the gameplay.

The contract logic designed for *BanaNomic* worked well on the whole and provided for a playable, well-balanced game. However its limited expressivity was also quickly noted by users, so attempting to apply the logic for a more complete version of Nomic would require considerably more work. The implementation of the game evaluator (§ 7.3.2) was also not without its share of limitations, and the code itself was not written with extensibility in mind such that adding new features to the evaluator would not be a trivial task.

The CNL interface used in *BanaNomic* seemed to be the area with which users experienced the least problems. None of the users involved expressed any notable difficulties with understanding the generated phrases, and all seemed to find the guided input methods both helpful and easy to use. The use of GF worked as desired, and proved to be an excellent way for building a natural language interface to a formally defined logic grammar.

Fulfilment of aims

All in all, the project goals as set out in § 1.3 were successfully completed. A workable system which incorporated all the desired features was built, and successfully tested by a group of external evaluators who assessed the quality of the CNL used and its effect on the use of the application. In the process of designing, building and evaluating the system, a number of limitations with the project methodology were established; these are discussed in the following section.

9.2 Points of limitation

9.2.1 Natural language

Multiple linearisations

Firstly, the CNL used has no real concept of *multiple linearisations* for any given statement. While in natural languages the same thing may be said in a variety of ways, there is no such flexibility here. It would be possible to handle these by having additional ”style” parameters in GF, however there is no automatic solution and every variation in style needs to be explicitly defined in the grammar. One might argue that this is the price that is paid for opting for a CNL approach.

CNL simplicity

While probably the most successful aspect of this project, in reality the design of the CNL in GF probably took the least effort of all system modules, in terms of implementation. While this may be a testament to the suitability of GF for the task, the restricted contract logic designed for the project meant that there was not that much scope to explore the deeper aspects of controlled languages.

Ultimately, the CNL interface used came quite close to a fully template-like approach, but as shown in the evaluation results it proved to be adequate for its purpose. The weaknesses of this approach tend to become apparent when long, nested clauses are used:

> *If every player has less than 5 bananas then at some point if any player has more than 2 bananas then every player is permitted to climb up the tree otherwise every player is forbidden to climb down the tree otherwise every player is permitted to pick one banana.*

While well-formed according to the CNL grammar, sentences such as these are completely impossible for human readers to follow. Cases such as these require proper handling if the CNL is to retain its 'naturalness', however it must be said that—at least within the evaluation period—users never created rules of this depth. Whether this is in spite of the CNL limitations or because of them cannot really be determined.

Use of English

In relation to the previous point, it should also be noted that the CNL interface designed for this project was in English only. As with other CNLs that are designed for only a single natural language (Fuchs et al., 2008; Pulman, 1996), regardless of our own experience we simply cannot comment on the ease or difficulty involved in extending our CNL to other languages. The huge variations that exist between language families (e.g. Germanic, Romance, Slavic, Semitic etc.) mean that the design and implementation of any CNL interface are largely dictated by the 'base' natural language used. That being said, Ranta and Angelov (2009) did successfully manage to port ACE (Fuchs et al., 2008) to French, German and Swedish by using the GF Resource Grammar Library.

9.2.2 Contract logic

Restrictions on OPP-logic

The contract logic used in this project is based on the *OPP-logic* from (Pace and Schneider, 2009), however a number of features were cut back or even removed for the benefit of simplicity:

i. Negation, repetition and sequencing of *Contracts*, as well as negation and timely clauses (sometimes/always) for *Activities* were omitted (see § 5.1).

ii. CTD, CTP and *unless* contracts were not implemented and there is no concept of 'repercussions' for unsatisfied obligations; instead all obligations are simply enforced. This is a big simplification over real-life contracts.

iii. The "if contract exists" conditional clause had to be left out as it proved overly complex to implement.

Concurrent actions

As explained in § 7.5.1 the handling of concurrent events could not be implemented as originally intended, which had major implications on the way the game was played (see also § 8.4.3).

Consistency checking

The game evaluator, as implemented, cannot really check for inconsistencies within a given contract; rather, it can only recognise contract-violating events as they happen. This was adequate enough for the requirements of *BanaNomic*, however it is a long way from being given a complete contract and determining whether it is self-contradictory.

Choice

Because of the limitations in the contract logic as described above, there was no opportunity to properly tackle the issues of internal and external choice. Instead, in *BanaNomic* all choice is always treated as external (angelic), i.e. decisions are made by the user performing the action in question.

9.2.3 Nomic

General limitations

Usually in games of Nomic players are free to think up any type of rule imaginable, however no such luxury could be afforded in our restricted version of the game. Since *BanaNomic* is a computerised game, all actions and rules must have a formal representation, and thus must be hardcoded in some way into the system. This of course results in a much more limited game than a purely linguistic version of Nomic.

Lack of voting system

A major departure in our project from the traditional Nomic is that rules can be enacted and abolished immediately, without going through a system of voting. This design choice was intentional, for it was feared that having a democratic voting system would introduce a whole lot of bureaucracy and result in not many rule amendments actually seeing the light of day (see § 5.3.6). This simplification *did* have the desired affect, although a related problem is that the ability to instantly abolish rules makes them somewhat futile in the first place (see § 8.4.1). This was quickly noted by the evaluators, however in general did *not* seem to detract from the enjoyability of the game.

9.2.4 Evaluation

Every method of evaluation has its boundaries and limitations. As discussed in § 8.4.4, the potential issues with evaluation in this project include:

- A possible over-generalisation by users, rating the quality of the natural language across the entire site rather than just the generated text phrases.

- The limited size and variety of the evaluator group.
- A potential evaluator bias because of personal relationships with the authors.
- The possibility that the feedback submitted was not thought out much, and simply completed because it was mandatory.

9.2.5 Other observations

Self-amendment of the game

In the evaluation period some users did comment that the rule possibilities quickly began to seem restricted, and they were not necessarily able to create the rules that they would have liked to. This perception is definitely warranted, and is a direct result of restriction made on the contract logic itself.

However the game in general is limited in a much greater yet subtler way; indeed, the very code on which the system runs is the ultimate limiting factor in *BanaNomic*. While this did not seem to be an issue for users, almost everything about how the game works is hard-coded, meaning that most things could never be changed throughout the duration of a game. These could include:

i. what happens when a banana is thrown,

ii. who is eligible to play their turn and when,

iii. the process through which rules are updated, and

iv. how victory is defined.

The true essence of Nomic really is this concept of total self-amendment, but clearly in our electronic version of the game this ideal could nowhere near be reached. When your game is purely language-based this is never a consideration, but to become amenable to computation it becomes a major issue. Ultimately, the area of self-amending programs is a vast and complex one, but achieving a high degree of self-amendment was not the primary goal of this project.

User trust & understanding

Throughout the evaluation period, a considerable amount of time had to be devoted to answering user questions about how the game rules are processed, notwithstanding the full user guide that was provided to them (see appendix A). In addition, the error messages generated by the system were not as detailed as they could have been (see § 8.4.3), which also contributed to users' uncertainties about the system.

From the questions being asked, it became clear that fully understanding the way the system worked was of utmost importance to the users, and directly affected the way they played and the rules they enacted. This is entirely reasonable, and one can imagine that in a real-world situation the correct understanding and trusting of an electronic contract system would be ever the more important. Thus an unexpected lesson learnt from this project is that simply building such a system is only *one* step towards gaining true acceptance for its actual use.

9.3 Future work

9.3.1 Unrealised ideas

The first candidates for future work would of course be the implementational limitations as listed in § 9.2. These include making both the logic and language grammars more expressive, correctly handling concurrent events, exploring the areas of internal and external choice, and implementing obligation management using a 'repercussions' system.

Beyond these basic improvements, one major step forward from this work would be the use of a full contract consistency checker, such as the CLAN tool (Fenech et al., 2009b). Such a tool would allow not only the validation of events as they happen, but also be able to detect anomalous or contradictory contract clauses. In terms of the Nomic game, this would allow for more interesting possibilities and victory conditions. For example, points could be awarded for giving other players conflicting obligations, or placing them in some other unsatisfiable situation.

One final aspect of the project which also lends itself to further research is the self-amending nature of Nomic. In *BanaNomic*, the degree to which the game can change itself only goes as far as the players' permissions and obligations. However there is potentially no limit to what aspects of the game are open to self-amendment (see § 9.2.5), and there is quite some scope for investigating the area of self-amending computer programs and how they could be used to simulate the game of Nomic.

9.3.2 Ultimate goals

On a broader scale, the eventual intention for a project such as this is to reach a point where electronic contracts combined with natural language technologies can be used together in real-world applications where computers have not yet penetrated. Some ideas for such applications are listed below.

Personal contract manager

In our everyday lives we are surrounded by contracts, from mobile phone subscriptions to house mortgages and everything between. Each such contract generally binds us to carry out certain actions (usually paying!) within specific time constraints, which people usually remember through calender reminders or by receiving a bill each month. But with an established standard for electronic contracts, computer applications could easily be built to manage all of one's contracts together in one place; for example within a calendar application. Such a program could notify the user of upcoming payment deadlines, advise on what would happen if payments are delayed or missed, and even ensure that the service given by the provider is itself as specified in the contract. This sort of capability would empower consumers to ensure their providers are living up to their end of the contract without the need to bring in professional lawyers.

Contract authoring & optimisation

So far we have mostly talked about the checking of events *against* an existing contract, with mention of some tools which can check for conflicts within a contract. But why not take this further, to optimising contracts to be as short as possible, or to be favourable to one party over another (ignoring ethical considerations!). Better yet, one can imagine a contract authoring system which, given a declarative set of requirements, could compose a complete contract for the author.

Government simulator

The study of electronic contracts usually names service-oriented architectures (SOAs) as its primary focus, such as ISP service-level agreements (SLAs). However the concept of a contract can even be extended to government legislation, which ultimately deals with the same notions of obligations, permissions and prohibitions (along with the related concepts of duties and rights). From this, one can easily imagine a system for simulating governmental systems which could indicate public law violations of constitutional rights, or automatically assess how new international treaties would affect the policies of an individual state. The possibilities are vast, but whether there would be a demand for such systems or not is a different story.

9.4 Closing remarks

On the whole this project was successful in meeting all its initial aims. The system built included all the desired components, and was evaluated as planned by a group of external users who provided qualitative feedback on the contract logic and natural language used. The feedback received indicated an overall positive reception from the users, while still revealing the weaker areas of the implementation and providing scope for future work and improvements.

This project has demonstrated that the use of CNLs in conjunction with electronic contracts can produce a usable system with highly acceptable results. However in establishing the viability of the approach used, this project also uncovered some other interesting issues along the way.

The sheer amount of questions asked by players about the way the system processed their turns indicated that even with full information provided in writing, users were still in general unsure about how it would behave and apprehensive to accept and trust it. The unhelpful error messages produced by the system likely contributed to this, and the requirement of a user to know *why* their turn was not accepted turned out to be quite a basic need—yet it was not fully addressed by the system. While this project only involved the playing of a game, in a real-world electronic contract system such problems of trust would be far more serious.

Some of the main motivations for working with contracts electronically is to relieve people of the complicated task of figuring out contractual clauses and their implications, to reduce the general uncertainties that contracts often generate, and to bring a level of certainty and transparency to the evaluation of contracts. However it would be no understatement to say that in our case, the system built seemed to generate equally as much ambiguity and uncertainty over its behaviour as it solved. While the implementational issues encountered during this project could readily be solved with enough time, the human issue of gaining the user's trust in an automated system is not so easily addressable.

Appendix A

User guide

> *This is a copy of the user guide as provided on the* BanaNomic *site for the evaluators. It is included here in order to provide a full reference on how the game works and how it is played.*

A.1 The basics

A.1.1 What's this game all about?

BanaNomic is my take on the game of Nomic, a game where the rules of the game change with every turn. Usually games of Nomic are played with pen and paper or via email, but *BanaNomic* is completely computerised—which means that the program can actually understand the rules directly. The purpose of *BanaNomic* is to test these distinct features:

1. The management and self-amendment of formally defined game rules.
2. The automatic generation of English phrases.
3. Different methods for composing rules in simple English.

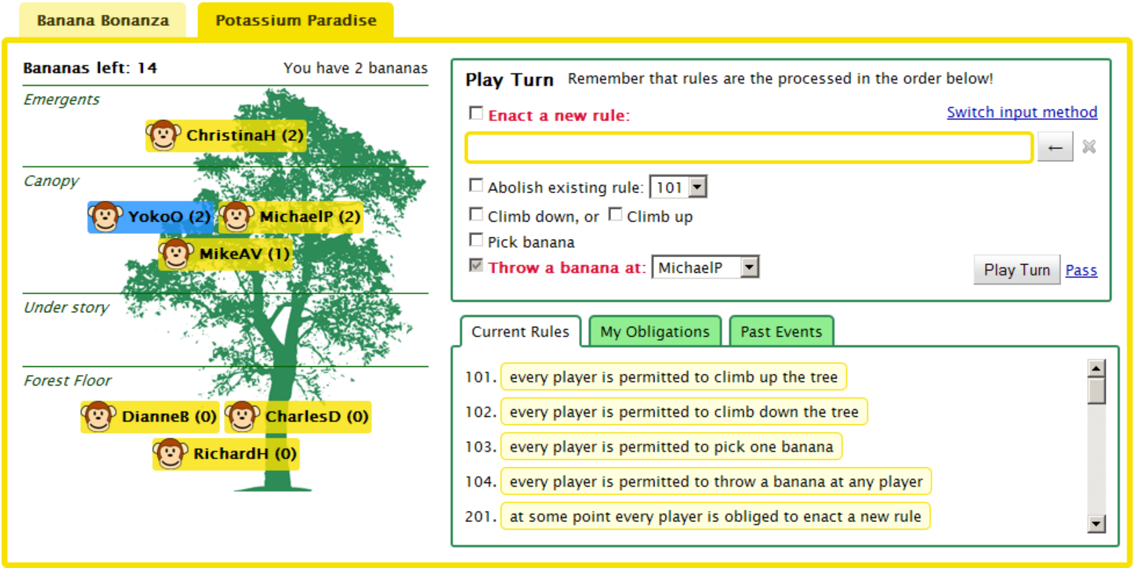
A.1.2 The setting

BanaNomic is a multiplayer game, where all players live together in a tree in the rainforest and compete against each other to collect the most bananas possible. While bananas can easily be picked from the tree, all players also must abide by a set a rules which govern rainforest life. These rules dictate what players are allowed, prohibited, and obliged to do. These rules cannot be broken; However, being a democratic rainforest, these rules can be changed by tree citizens. Thus, it is up to every player to bend the rules in their favour! (ok, so it's not very democratic).

During the evaluation phase two separate games—Banana Bonanza and Potassium Paradise—will be running concurrently, and every player will be assigned to one of the two games. Each player may play up to one turn per day, and both games will run for 7 days (or until a stalemate is reached).

A.1.3 Playing your turn

To view a game in progress, click on the Games tab to show a box similar to the one in figure A.1.



Play Turn Remember that rules are the processed in the order below!

□ Enact a new rule: Switch input method

□ Abolish existing rule: 101

□ Climb down, or □ Climb up

□ Pick banana

☑ Throw a banana at: MichaelP

Play Turn Pass

101. every player is permitted to climb up the tree
102. every player is permitted to climb down the tree
103. every player is permitted to pick one banana
104. every player is permitted to throw a banana at any player
201. at some point every player is obliged to enact a new rule

Figure A.1: Panel for playing one's turn.

The tree image on the left shows where each player is in the tree. The number in brackets next to each person's name shows how many bananas they currently have. Note the number of bananas left which can be picked.

The first tab at the bottom-right shows the Current Rules that are active in the game. These will change every time a player takes their turn, so always keep an eye on them. The Events tab will show a history of all the turns played so far, so you may go back in time and see how other players have played.

The Play Turn section at the top-left is where you play your turn for the day. Each turn may consist of one or more of the following:

- Enact a rule—compose a new rule which will become effective after your turn. This is the most important feature of the game, and your most powerful (and devious) tool for achieving victory!
- Abolish an existing rule—if a rule is working against you, you can abolish it altogether... as long as you are allowed to!
- Climb up or down the tree—See below for why you would want to do this.

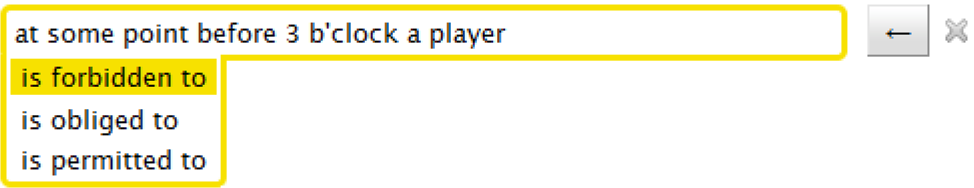
- Pick banana—pick a banana from the tree. Players cannot pick bananas from the Forest Floor, so you would need to climb up before doing so.
- Throw banana—throw a banana at another player who is on the same level as your are. You will lose one banana, but the other player will lose all their bananas and end up on the Forest Floor again (ouch!)

A.1.4 Composing rules

When composing a new rule, rather than just typing it out, you may use one of the following 2 input methods. Whichever you use is totally up to you! Just give both a try and see which you prefer. In both cases, a small icon on the right will indicate whether the phrase you've constructed is valid (✓) or not (✗). To make things more straightforward, neither of the input methods handles punctuation.

Suggest Panel

The Suggest Panel (figure A.2) is a normal text box with a drop-down selection of matching phrases, which most users will already be familiar with.



at some point before 3 b'clock a player

Figure A.2: Suggest panel input method.

Fridge Magnets

The so-called Fridge Magnets method does not involve typing; Instead one simply clicks the desired phrase at each point until an entire sentence is constructed.

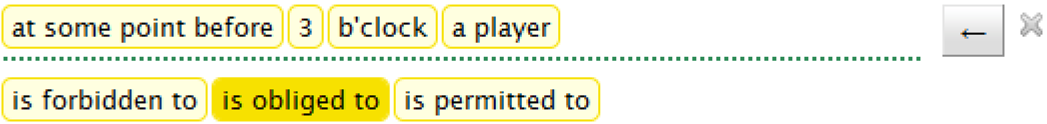
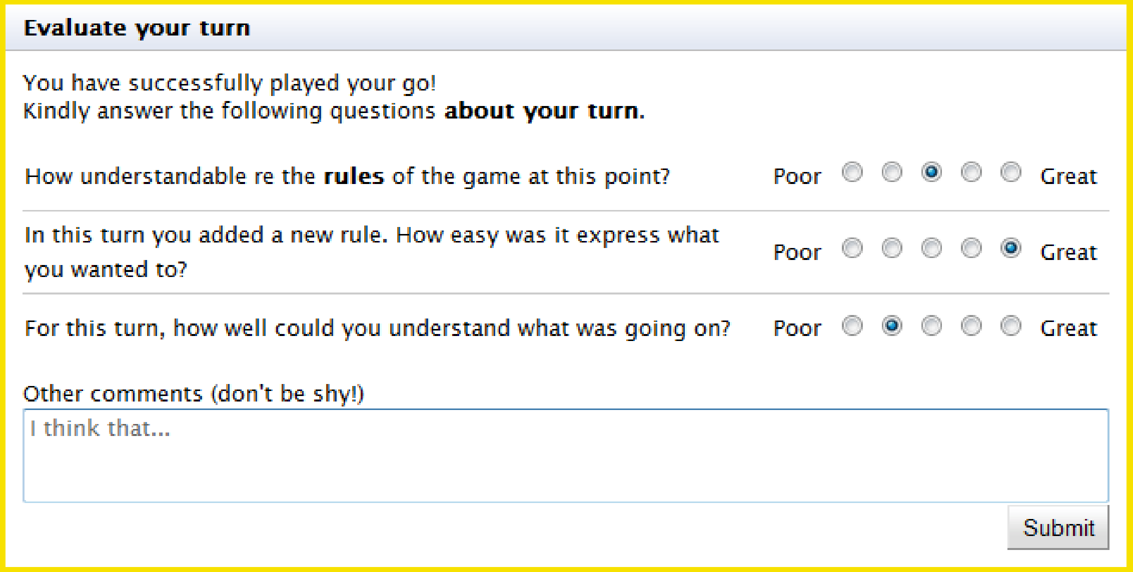


Figure A.3: Fridge magnets input method.

A.1.5 Feedback

After every turn, you will be shown a small feedback dialog which you will need to fill in in order to continue. As shown below, you will just be asked some basic questions about your turn and how easy or difficult you found it to play. This is one of the most important parts of the project for me, and I'd really appreciate it if you answered truthfully each time! Don't worry about giving negative feedback, as I will not be graded on the success of the project, but rather on the method.



Evaluate your turn

You have successfully played your go!
Kindly answer the following questions **about your turn.**

How understandable re the **rules** of the game at this point? Poor ○ ○ ● ○ ○ Great

In this turn you added a new rule. How easy was it express what you wanted to? Poor ○ ○ ○ ○ ● Great

For this turn, how well could you understand what was going on? Poor ○ ● ○ ○ ○ Great

Other comments (don't be shy!)

I think that...

Figure A.4: The in-game feedback dialog shown to players after each turn.

A.1.6 The initial rules

Each game starts off with the following initial rule set, which is just intended to get you going:

1. every player is permitted to climb up the tree
2. every player is permitted to climb down the tree
3. every player is permitted to pick one banana
4. every player is permitted to throw a banana at any player
5. at some point every player is obliged to enact a new rule
6. at all times from 3 b'clock every player is permitted to abolish an existing rule
7. if any player has more than 10 bananas then every player is forbidden to throw a banana at any player

A.1.7 Some gameplay tips

- Tool-tips are included all over the site, so to get more information about something just hover the mouse cursor over it.
- You are free to add all the rules you want. However, too many rules will tend to conflict with each other and make the game unplayable! Instead, consider abolish an existing rule.

A.2 Other issues

A.2.1 What's b'clock?

Banana time of course! One b'clock is equivalent to one human day, so 3 b'clock would mean three days since the commencement of the game.

A.2.2 How actions are processed

Although all the actions in a single turn are chosen together, internally they are in fact processed in sequence. This sequence is precisely the same as the way the actions are presented in the play panel, i.e.:

1. Enact rule
2. Abolish rule
3. Climb down / Climb up
4. Pick banana
5. Throw banana

This means that if in your turn you choose climb down and pick banana, then the banana will be picked after you have climbed down.

A.2.3 How rules are processed

Every action in the game must satisfy at least one of the game's active rules. For example, given these two rules:

1. Player 1 is permitted to either climb up the tree or pick a banana
2. Player 1 is obliged to abolish a rule

Then if Player 1 picks a banana, it will be accepted because it satisfies the first rule. Note that prohibition takes precedence over all other rules (see below).

A.2.4 Permission, obligation and prohibition

The game centres around these 3 basic concepts:

Permission A player is allowed to do something, but does not necessarily need to do it.

Obligation A player must do something, and there may be repercussions for failing to do so. Note that obligation implies permission.

Prohibition A player is forbidden from doing something, and there may be repercussions for disobeying. Prohibition takes precedence over permission and obligation.

Notes

- The absence of permission implies prohibition. So a player is only allowed to do something if a rule exists which explicitly allows them to do so.

A.2.5 Sometimes & always

Rules can include the sometimes/always clauses to make them effective only at certain times, or within a time range (e.g. between 1 b'clock and 5 b'clock). The sometimes clause means that an obligation will only need to be satisfied once within the given range, while an obligation within an always clause will mean it is effective at every turn.

A.2.6 Fixed laws

While the rules of the game are mutable (can be changed by players), the Laws below are fixed and dictate the basics of how the game is played:

1. The game will end after a fixed amount of time.
2. At the end of the game, the player with the highest number of bananas will be declared the winner.
3. Players can only play once per day.
4. Bananas cannot be picked from the Forest Floor.
5. Bananas can only be thrown at players who are at the same level as the thrower.
6. One cannot climb up and down at the same time.

A.3 Technical issues

A.3.1 Browser compatibilities

BanaNomic has been fully tested under Mozilla FireFox and Google Chrome, however it should work in all modern browsers. The site will look its best in FireFox, but that mostly just refers to aesthetics (rounded corners and such). This site has not been tested under mobile browsers (e.g. iPhone, BlackBerry) but you are welcome to try and let me know.

A.3.2 Bugs and glitches

As hard as I've tried to make sure everything works smoothly, it is entirely possible some bugs will crop up in the game. If you discover such a bug, be sure to let me know while trying to continue playing as best you can.

Appendix B

Game transcripts

This section contains the full transcripts from the two BanaNomic *games played during the project's evaluation period—*Banana Bonanza *and* Potassium Paradise. *Included are the initial rule set (common to both games), details of all turns played and rules enacted and summaries of the final rule sets and player standings for each game. Finally, we then take a closer look at some of the more interesting rules created during the course of the two games.*

B.1 Initial rules

Table B.1: Initial rules for both games.

Rule ID	Clause
101	*every player is permitted to climb up the tree*
102	*every player is permitted to climb down the tree*
103	*every player is permitted to pick one banana*
104	*every player is permitted to throw a banana at any player*
201	*at some point every player is obliged to enact a new rule*
202	*at all times from 3 b'clock every player is permitted to abolish an existing rule*
301	*if any player has more than 10 bananas then every player is forbidden to throw a banana at any player*
Total rules: 7	

B.2 Turn histories

The following tables list in order all the turns played in both games throughout the evaluation period, along with each new rule enacted and each existing rule abolished.

B.2.1 Banana Bonanza

Table B.2: Transcript for *Banana Bonanza* game.

Player	Action
1 b′clock (Start of game)	
SilvanaF	– Climbed up the tree
GarethF	– Climbed up the tree
AnnaC	– Climbed up the tree
JoeC	– Enacted rule 302: *at all times before 9 b′clock player "JoeC" is permitted to enact a new rule* – Climbed up the tree
StephenH	– Climbed up the tree
2 b′clock	
JoeC	– Enacted rule 303: *at some point every player is obliged to throw a banana at any player* – Picked a banana
SilvanaF	– Climbed up the tree – Picked a banana
AnnaC	– Enacted rule 304: *at some point every player is obliged to climb down the tree* – Threw a banana at JoeC
3 b′clock	
SilvanaF	– Climbed up the tree – Picked a banana
StephenH	– Abolished rule 302 – Climbed up the tree – Picked a banana
JoeC	– Enacted rule 305: *if player "AnnaC" has more than 1 bananas then every player is obliged to throw a banana at player "AnnaC"* – Abolished rule 202 – Climbed up the tree
AnnaC	– Picked a banana
4 b′clock	

SilvanaF	– Enacted rule 306: *if player "SilvanaF" has more than 3 bananas then every player is forbidden to climb up the tree* – Picked a banana
JoeC	– Enacted rule 307: *at all times between 4 b'clock and 8 b'clock player "SilvanaF" is obliged to climb down the tree* – Picked a banana
GarethF	– Enacted rule 308: *every player is forbidden to throw a banana at player "GarethF"* – Climbed down the tree – Picked a banana – Threw a banana at StephenH
AnnaC	– Enacted rule 309: *player "AnnaC" is permitted to climb up the tree* – Picked a banana
5 B'CLOCK	
JoeC	– Enacted rule 310: *player "JoeC" is permitted to abolish an existing rule* – Picked a banana – Threw a banana at AnnaC
SilvanaF	– Enacted rule 311: *at some point between 4 b'clock and 8 b'clock player "JoeC" is obliged to climb down the tree* – Climbed down the tree – Threw a banana at AnnaC
NicolaVH	– Enacted rule 312: *(nothing)* – Threw a banana at AnnaC
StephenH	– Enacted rule 313: *player "SilvanaF" is forbidden to pick one banana* – Picked a banana – Threw a banana at AnnaC
JeanLucP	– Threw a banana at AnnaC
6 B'CLOCK	
SilvanaF	– Enacted rule 314: *player "SilvanaF" is permitted to abolish an existing rule*
StephenH	– Picked a banana – Threw a banana at SilvanaF
AnnaC	– Enacted rule 315: *player "AnnaC" is permitted to abolish an existing rule* – Abolished rule 304
JoeC	– Enacted rule 316: *at all times from 6 b'clock player "StephenH" is obliged to climb down the tree* – Abolished rule 314 – Climbed up the tree – Picked a banana – Threw a banana at StephenH
7 B'CLOCK	

JoeC	– Enacted rule 317: *at all times every player is forbidden to throw a banana at player "JoeC"* – Abolished rule 101 – Climbed down the tree – Picked a banana
StephenH	– Enacted rule 318: *player "JoeC" is obliged to concurrently throw a banana at player "GarethF" and climb down the tree* – Climbed down the tree
AnnaC	– Enacted rule 319: *player "AnnaC" is permitted to pick one banana* – Abolished rule 305
SilvanaF	– Enacted rule 320: *at some point between 7 b'clock and 8 b'clock player "JoeC" is obliged to climb down the tree*
8 b'clock	
JoeC	– Enacted rule 321: *player "JoeC" is permitted to climb up the tree* – Abolished rule 308 – Climbed down the tree – Threw a banana at GarethF
StephenH	– Enacted rule 322: *player "StephenH" is obliged to pick one banana* – Climbed down the tree
SilvanaF	– Enacted rule 323: *at all times between 8 b'clock and 9 b'clock player "JoeC" is obliged to climb down the tree* – Climbed down the tree
AnnaC	– Climbed up the tree – Picked a banana
JeanLucP	– Enacted rule 324: *every player is forbidden to concurrently throw a banana at any player and pick one banana* – Picked a banana – Threw a banana at AnnaC
9 b'clock	
JoeC	– Abolished rule 309 – Climbed up the tree – Picked a banana – Threw a banana at AnnaC
StephenH	– Enacted rule 325: *player "JoeC" is obliged to throw a banana at player "NicolaVH"* – Climbed down the tree – Picked a banana
AnnaC	– Threw a banana at JeanLucP

B.2.2 Potassium Paradise

Table B.3: Transcript for *Potassium Paradise* game.

Player	Action
1 b′clock (Start of game)	
MikeAV	– Climbed up the tree
ChristinaH	– Enacted rule 302: *if any player has exactly 7 bananas then every player is forbidden to enact a new rule* – Climbed up the tree
MichaelP	– Enacted rule 303: *if any player has exactly 5 bananas then every player is obliged to climb down the tree* – Climbed up the tree
YokoO	– Enacted rule 304: *every player is forbidden to throw a banana at player "YokoO"* – Climbed up the tree
RichardH	– Enacted rule 305: *every player is forbidden to throw a banana at player "RichardH"* – Climbed up the tree
2 b′clock	
MikeAV	– Enacted rule 306: *every player is forbidden to throw a banana at player "MikeAV"* – Climbed down the tree – Picked a banana
ChristinaH	– Enacted rule 307: *every player is forbidden to throw a banana at player "ChristinaH"* – Climbed up the tree – Picked a banana
YokoO	– Enacted rule 308: *player "RichardH" is obliged to climb down the tree* – Climbed up the tree – Picked a banana
3 b′clock	
MikeAV	– Enacted rule 309: *player "ChristinaH" is obliged to climb down the tree* – Climbed up the tree
YokoO	– Abolished rule 305 – Climbed down the tree – Picked a banana – Threw a banana at RichardH
DianneB	– Enacted rule 310: *every player is forbidden to throw a banana at player "DianneB"*

MichaelP	– Enacted rule 311: *at all times every player is permitted to throw a banana at any player* – Climbed up the tree – Picked a banana
ChristinaH	– Abolished rule 309
4 b′clock	
MikeAV	– Enacted rule 312: *if player "ChristinaH" has more than 1 bananas then player "MichaelP" is obliged to throw a banana at player "ChristinaH"* – Abolished rule 307 – Climbed up the tree – Picked a banana
DianneB	– Climbed up the tree
ChristinaH	– Enacted rule 313: *player "MikeAV" is forbidden to climb up the tree* – Climbed up the tree – Picked a banana
YokoO	– Climbed up the tree – Picked a banana – Threw a banana at MichaelP
MichaelP	– Enacted rule 314: *every player is forbidden to throw a banana at player "MichaelP"* – Abolished rule 310 – Climbed up the tree
5 b′clock	
MikeAV	– Enacted rule 315: *player "ChristinaH" is obliged to climb down the tree* – Abolished rule 313 – Climbed up the tree – Picked a banana
ChristinaH	– Enacted rule 316: *player "YokoO" is forbidden to climb up the tree* – Picked a banana
YokoO	– Abolished rule 316 – Climbed up the tree – Picked a banana – Threw a banana at ChristinaH
6 b′clock	
YokoO	– Enacted rule 317: *every player is forbidden to climb up the tree* – Abolished rule 306 – Picked a banana – Threw a banana at MikeAV

MikeAV	– Enacted rule 318: *player "YokoO" is obliged to climb down the tree* – Abolished rule 317 – Climbed up the tree
ChristinaH	– Enacted rule 319: *if player "YokoO" is at the emergents level then player "YokoO" is obliged to throw a banana at player "MikeAV"* – Abolished rule 315
DianneB	– Climbed up the tree
7 b′clock	
MichaelP	– Enacted rule 320: *player "YokoO" is obliged to climb down the tree* – Picked a banana
MikeAV	– Enacted rule 321: *player "YokoO" is obliged to throw a banana at player "DianneB"* – Abolished rule 319 – Climbed up the tree – Picked a banana
ChristinaH	– Enacted rule 322: *if player "YokoO" has exactly 2 bananas then player "MikeAV" is obliged to throw a banana at player "YokoO"* – Abolished rule 312 – Climbed up the tree
DianneB	– Enacted rule 323: *player "YokoO" is obliged to throw a banana at player "MichaelP"* – Picked a banana
CharlesD	– Climbed up the tree
8 b′clock	
MichaelP	– Enacted rule 324: *every player is forbidden to throw a banana at player "MichaelP"* – Abolished rule 316 – Climbed up the tree – Picked a banana
CharlesD	– Climbed up the tree – Picked a banana
ChristinaH	– Enacted rule 325: *every player is forbidden to throw a banana at player "ChristinaH"* – Climbed up the tree – Picked a banana
DianneB	– Picked a banana
YokoO	– Climbed up the tree – Picked a banana – Threw a banana at DianneB

9 b′clock	
ChristinaH	– Abolished rule 304 – Climbed up the tree – Picked a banana
YokoO	– Picked a banana – Threw a banana at CharlesD

B.3 Final summaries

This section contains summaries of the final set of rules and player standings for each game.

B.3.1 Banana Bonanza

Table B.4: Final rules for *Banana Bonanza* game.

Rule ID	Clause
102	*every player is permitted to climb down the tree*
103	*every player is permitted to pick one banana*
104	*every player is permitted to throw a banana at any player*
201	*at some point every player is obliged to enact a new rule*
301	*if any player has more than 10 bananas then every player is forbidden to throw a banana at any player*
303	*at some point every player is obliged to throw a banana at any player*
306	*if player "SilvanaF" has more than 3 bananas then every player is forbidden to climb up the tree*
307	*at all times between 4 b′clock and 8 b′clock player "SilvanaF" is obliged to climb down the tree*
310	*player "JoeC" is permitted to abolish an existing rule*
311	*at some point between 4 b′clock and 8 b′clock player "JoeC" is obliged to climb down the tree*
312	*(nothing)*
313	*player "SilvanaF" is forbidden to pick one banana*
315	*player "AnnaC" is permitted to abolish an existing rule*
316	*at all times from 6 b′clock player "StephenH" is obliged to climb down the tree*
317	*at all times every player is forbidden to throw a banana at player "JoeC"*
318	*player "JoeC" is obliged to concurrently throw a banana at player "GarethF" and climb down the tree*
319	*player "AnnaC" is permitted to pick one banana*
320	*at some point between 7 b′clock and 8 b′clock player "JoeC" is obliged to climb down the tree*
321	*player "JoeC" is permitted to climb up the tree*

322	*player "StephenH" is obliged to pick one banana*
323	*at all times between 8 b'clock and 9 b'clock player "JoeC" is obliged to climb down the tree*
324	*every player is forbidden to concurrently throw a banana at any player and pick one banana*
325	*player "JoeC" is obliged to throw a banana at player "NicolaVH"*
Total rules: 23	

Table B.5: Final player standings for *Banana Bonanza* game.

Bananas left	20	
Emergents level	–	
Canopy level	–	
Under Story level	JoeC (1 banana)	
Forest Floor level	SilvanaF (0 bananas) NicolaVH (0 bananas) JeanLucP (0 bananas)	GarethF (0 bananas) AnnaC (0 bananas) StephenH (0 bananas)

B.3.2 Potassium Paradise

Table B.6: Final rules for *Potassium Paradise* game.

Rule ID	Clause
101	*every player is permitted to climb up the tree*
102	*every player is permitted to climb down the tree*
103	*every player is permitted to pick one banana*
104	*every player is permitted to throw a banana at any player*
201	*at some point every player is obliged to enact a new rule*
202	*at all times from 3 b'clock every player is permitted to abolish an existing rule*
301	*if any player has more than 10 bananas then every player is forbidden to throw a banana at any player*
302	*if any player has exactly 7 bananas then every player is forbidden to enact a new rule*
303	*if any player has exactly 5 bananas then every player is obliged to climb down the tree*
308	*player "RichardH" is obliged to climb down the tree*
311	*at all times every player is permitted to throw a banana at any player*
314	*every player is forbidden to throw a banana at player "MichaelP"*
318	*player "YokoO" is obliged to climb down the tree*
320	*player "YokoO" is obliged to climb down the tree*
321	*player "YokoO" is obliged to throw a banana at player "DianneB"*
322	*if player "YokoO" has exactly 2 bananas then player "MikeAV" is obliged to throw a banana at player "YokoO"*

323	*player "YokoO" is obliged to throw a banana at player "MichaelP"*
324	*every player is forbidden to throw a banana at player "MichaelP"*
325	*every player is forbidden to throw a banana at player "ChristinaH"*
Total rules: 19	

Table B.7: Final player standings for *Potassium Paradise* game.

Bananas left	14	
Emergents level	ChristinaH (2 bananas)	
Canopy level	YokoO (2 bananas) MikeAV (1 banana)	MichaelP (2 bananas)
Under Story level	–	
Forest Floor level	DianneB (0 bananas) CharlesD (0 bananas)	RichardH (0 bananas)

B.4 Interesting behaviour

This section contains some of the more interesting rules and tactics which emerged during the *BanaNomic* evaluation games.

Self-protective rules

One of the first trends to emerge was that of self-protecting rules, in particular:

> YokoO : *every player is forbidden to throw a banana at player "YokoO"*

As soon as the first player came up with this rule, practically all the other players immediately followed suit. Evidently, it was going to be a monkey-eat-monkey kind of game.

The prohibition loophole

As mentioned § 8.4, it didn't take players too long to notice that any prohibitive rules could easily be circumvented by simply abolishing them and carrying out whatever action they were meant to prohibit within the same turn. For example, at one point the following rule was enacted:

> ChristinaH : 316. *player "YokoO" is forbidden to climb up the tree*

However in the next turn, YokoO immediately got around this by choosing to *abolish rule 316* and *climb up the tree* in a single go.

Blind vengeance

One player in particular exhibited a notable ruthlessness in their approach; after JoeC was hit by a banana thrown by AnnaC, he proceeded to enact this rule:

> JoeC : *if player "AnnaC" has more than 1 bananas then every player is obliged to throw a banana at player "AnnaC"*

While this did result in everyone throwing a banana at AnnaC, all players actually had to throw their bananas at her *in the same turn*. This was really quite a waste of everyone's effort as after the first banana was thrown, AnnaC had nothing left to lose anyway.

Exploiting security holes

While some players tended to create protective rules such that no bananas could be thrown at them, others seemed to find creative ways of poking holes in these approaches. One player first enacted this rule to protect her position at the top of the tree:

> SilvanaF : *if player "SilvanaF" has more than 3 bananas then every player is forbidden to climb up the tree*

This may look fairly safe, but JoeC cleverly managed to circumvent this by bringing the mountain to him:

> JoeC : *at all times between 4 b'clock and 8 b'clock player "SilvanaF" is obliged to climb down the tree*

Thus SilvanaF was forced to leave her safety zone, exposing herself to further attacks.

Getting others to do your dirty work

Another clever tactic used by some players was getting their *opponents* to deal with their potential threats. One good example of this is:

> ChristinaH : *if player "YokoO" is at the emergents level then player "YokoO" is obliged to throw a banana at player "MikeAV"*

This allowed ChristinaH to throw a banana any other player she wished, safe in the knowledge that MikeAV was already being taken care of.

Down you go, and stay there!

In another brilliant exhibition of ruthlessness, JoeC enacted the following rule while simultaneously throwing a banana at poor StephenH:

> JoeC : *at all times from 6 b'clock player "StephenH" is obliged to climb down the tree*

This turn was clearly intended to send StephenH to the bottom of the tree, and totally prevent him from further advancing in the game by prohibiting him to climb up again. Quite an effective combination!

Two birds, one stone

StephenH was not without his own fair share of deviousness (see previous item), and quickly devised the following rule to get back at his aggressor and also take care of another opponent at the same time:

> StephenH : *player "JoeC" is obliged to concurrently throw a banana at player "GarethF" and climb down the tree*

Clutching at straws

Finally, desperation seemed to get the better of some players, who tried to enact permissions for themselves that already existed, in the hopes that they would somehow "work more". In one game, the following initial rules had still remained intact:

> 103. *every player is permitted to pick one banana*
> 201. *at some point every player is obliged to enact a new rule*

However this not stop these players from re-enacting the same thing, in the hopes that other prohibitions for these actions would somehow become cancelled out:

> AnnaC : *player "AnnaC" is permitted to pick one banana*
> StephenH : *player "StephenH" is obliged to pick one banana*
> SilvanaF : *player "SilvanaF" is permitted to abolish an existing rule*
> AnnaC : *player "AnnaC" is permitted to abolish an existing rule*

Appendix C

Evaluation forms & results

A collection of all evaluation materials and results. This appendix includes a list of the evaluation participants, the questions asked of them and their full responses for both the in-game feedback and the post-game questionnaire.

C.1 Evaluators

Table C.1 shows a list of the evaluators who played *BanaNomic* and provided in-game and post-game feedback. Note that some names have been randomly assigned for those users who wished to remain anonymous.

Table C.1: List of evaluators.

Username	Profession/Area of Study	Education Level
MichaelP	Molecular Genetics/Pharmacy	PhD
YokoO	Pharmacist	MSc
AnnaC	Secondary Teacher	BSc
JoeC	Engineering	BSc
DianneB	Pharmacist	BSc
JeanLucP	IT	BSc
CharlesD	English Teacher	BA
StephenH	Accountancy	BCom
MikeAV	Accountancy	BCom
ChristinaH	Events Project Manager	BA
SilvanaF	Learning Support Assistant	Diploma
GarethF	Dental Surgery (student)	BChD
NicolaVH	Statistician	BSc
RichardH	Property Surveying	LLB

C.2 In-game feedback

C.2.1 Feedback form

Generated Text How understandable are the rules of the game at this point?
(0 = Poor, 4 = Great)

Text Input In this turn you added a new rule. How easy was it express what you wanted to?
(0 = Poor, 4 = Great)

Gameplay For this turn, how well could you understand what was going on?
(0 = Poor, 4 = Great)

Other Comments
(Free-text box)

C.2.2 Feedback responses

Table C.2 is a raw list of feedback submitted by users in the evaluation period. The rating columns are abbreviated as follows:

1. *G* — Rating for generated text
2. *I* — Rating for input method
3. *P* — Rating for general game play
4. *M* — Input method used, where:
   - *SP* — Suggest Panel
   - *FM* — Fridge Magnets

The user comments have been corrected for typographical errors.

Table C.2: In-game feedback evaluation data.

Date	User	Comment	G^1	I^2	P^3	M^4
2010-04-15	MikeAV	Prosit John! Seems like fun, looking forward to playing my next turn.	4	-	4	-
2010-04-15	ChristinaH	I like the predictive text in the rules box - makes it easier.	2	4	2	*SP*
2010-04-15	MichaelP	Personally I wouldn't start the Monkey's on the forest floor as it seems to be a waste of a turn, as everyone will just climb up one (a bit predictable) having 2 turns just for the first day might help	3	3	4	*SP*

2010-04-15	SilvanaF	This first step was easy, but there are too many rules to remember. Maybe once I get used to the game it will get easier. Let's hope the game will not have ended by the time I get the hang of it! Bear with me!! Good luck anyway!	3	-	3	-
2010-04-15	GarethF	For some reason I didn't manage to play the game using my laptop but it worked perfectly on the other pc...	3	-	3	-
2010-04-15	AnnaC	It seems that there are a lot of rules to follow but hopefully playing the game will make them more obvious	2	-	2	-
2010-04-15	JoeC	It was easier to pick phrases from the list then use the dropdown menus	2	3	2	*FM*
2010-04-15	StephenH	-	2	-	2	-
2010-04-15	YokoO	Fridge magnet method of enacting rules very easy to use.	3	4	4	*FM*
2010-04-15	RichardH	-	0	2	1	*SP*
2010-04-16	JoeC	It didn't let me stop players from getting bananas!	3	3	3	*FM*
2010-04-16	MikeAV	-	4	4	4	*SP*
2010-04-16	ChristinaH	I think 'a player' was clearer than 'every player' but from your facebook message I might be in the minority	2	3	3	*SP*
2010-04-16	SilvanaF	I understood the rules ok but now I hope I will be able to abolish rule 101 before I quit the game!	3	-	3	-
2010-04-16	AnnaC	Although asked to enact a new rule, the choice was very restricted. However, the phrases came up in turn well.	2	2	2	*SP*
2010-04-17	YokoO	I was unable to play and was receiving errors, but after the bug was fixed I could play successfully!	4	4	3	*FM*
2010-04-17	SilvanaF	-	4	-	4	-
2010-04-17	MikeAV	-	4	4	4	*SP*
2010-04-17	StephenH	-	4	-	4	-
2010-04-17	YokoO	Easy to get around existing rules by abolishing an old one.	4	-	4	-
2010-04-17	DianneB	-	3	3	2	*SP*
2010-04-17	JoeC	-	4	4	3	*FM*
2010-04-17	AnnaC	-	3	-	3	-
2010-04-17	MichaelP	-	3	4	3	*SP*
2010-04-17	ChristinaH	For some reason I couldn't enact a new rule	2	-	1	-

2010-04-18	MikeAV	-	4	4	4	*SP*
2010-04-18	DianneB	-	3	-	2	-
2010-04-18	SilvanaF	How cruel I am but I enjoying winning!!!	3	3	3	*SP*
2010-04-18	ChristinaH	I wanted to add this rule but could not: every player is forbidden to have more bananas than "ChristinaH". Overall, the rules you can set are limited even if you understand the language with which you can set new rules	3	1	3	*SP*
2010-04-18	YokoO	Forgot to tick "enact" a new rule so my MEGA rule was not implemented.	4	-	2	-
2010-04-18	JoeC	As the game develops, the possibilities become more interesting	4	4	3	*FM*
2010-04-18	GarethF	I had some obligations to fill and my turn wasn't accepted until I did them, the problem was that the error message did not tell me why my turn was not accepted, the error message was too generic.	2	4	0	*FM*
2010-04-18	AnnaC	-	3	4	3	*SP*
2010-04-18	MichaelP	-	4	4	4	*SP*
2010-04-19	MichaelP	Rule says I have to throw a banana - but I have zero bananas, which means I cannot play my turn.	-	-	-	-
2010-04-19	JoeC	-	3	4	3	*FM*
2010-04-19	MikeAV	-	4	4	4	*SP*
2010-04-19	ChristinaH	I tried to enact the rule: if player ChristinaH is at the Emergents level then player YokoO is forbidden to climb up the tree, but it didn't let me, although I can't quite figure out why because I couldn't see a conflicting rule	4	3	4	*SP*
2010-04-19	SilvanaF	I'm getting the hang of it now. I can understand why AnnaC threw a banana at JoeC, I would gladly throw him down the tree!	3	4	4	*SP*
2010-04-19	NicolaVH	I had many obligations, but I couldn't act out all of them. I'm still not quite sure what I'm supposed to be doing!	1	1	1	*FM*
2010-04-19	StephenH	-	3	4	2	*SP*
2010-04-19	AnnaC	Some smart a-- made a rule that I have to throw a banana - and I do not have any to throw :(	-	-	-	-
2010-04-19	YokoO	It was quite easy to get round other people's rules	3	-	4	-
2010-04-19	DianneB	-	2	3	3	*SP*

2010-04-20	SilvanaF	I could not chose more than 1 action in my new rule I do not know if that was because of what player JoeC rules said about me.	3	3	4	*SP*
2010-04-20	YokoO	-	4	4	4	*SP*
2010-04-20	MikeAV	-	4	4	4	*FM*
2010-04-20	ChristinaH	-	4	4	4	*SP*
2010-04-20	StephenH	-	3	-	3	-
2010-04-20	DianneB	-	3	-	3	-
2010-04-20	AnnaC	-	3	4	3	*SP*
2010-04-20	JoeC	It's a great game!	4	4	4	*FM*
2010-04-21	MichaelP	-	4	4	4	*SP*
2010-04-21	JoeC	I changed my input method. At first I was using the 'fridge magnets' method, and got quite used to it, but now I tried the direct input method and found that it works well once you know roughly how to enact the rules	4	4	3	*SP*
2010-04-21	MikeAV	-	4	4	4	*SP*
2010-04-21	ChristinaH	How come I don't have the option to pick a banana?	4	4	4	*SP*
2010-04-21	StephenH	-	3	3	3	*SP*
2010-04-21	AnnaC	I have been spending more time counteracting other people's rules against me than going up the tree or picking bananas	3	2	2	*SP*
2010-04-21	DianneB	-	3	3	3	*SP*
2010-04-21	CharlesD	Not sure how I'm meant to get bananas at this point but just trying it out	2	-	1	-
2010-04-21	SilvanaF	I had 3 obligations and could only enact the rule because I was already on d forest floor and couldn't throw a banana at anyone since d box was grey. I didn't quite understand what I was meant to do. Anyway after several attempts I managed it though not what I really wanted to do...	2	3	1	*SP*
2010-04-21	YokoO	In this turn I was obliged to throw a banana at a number of players and climb down a level. The "throw a banana" box was already ticked and I could not in any way change the name of the player I wanted to throw a banana at. I ended up wasting 1 banana as the player was not even on my level.	2	2	2	*FM*
2010-04-22	MichaelP	-	3	4	4	*SP*

2010-04-22	JoeC	I have used this to my advantage, however the capability to remove rules should not be removable, as this tends to kill the game. Also, I think it would be better if monkeys could throw bananas at people below them rather than at the same level, and not lose all bananas and fall off. It is too harsh.	4	4	4	*SP*
2010-04-22	MikeAV	Couldn't play my turn - even though I was trying to climb the tree to carry out my obligation. Since the rules are processed in order I thought I would be able to. mav	-	-	-	-
2010-04-22	CharlesD	-	2	-	2	-
2010-04-22	StephenH	-	1	4	1	*SP*
2010-04-22	SilvanaF	I don't think anyone will win the game because we all have 0 bananas except HIM and no one can throw him a banana!!	3	3	3	*SP*
2010-04-22	ChristinaH	Easy peasy	4	4	4	*SP*
2010-04-22	AnnaC	-	3	-	3	-
2010-04-22	DianneB	-	3	-	3	-
2010-04-22	JeanLucP	There is a huge amount of information and rules that needs to be taken into account both on a constant basis (time constraints) as well as during one's turn, which tends to make strategies for rule creation and use a bit obscure.	1	3	2	*SP*
2010-04-22	YokoO	Previous problem seems to have been fixed!	3	-	4	-
2010-04-23	JoeC	-	4	-	4	-
2010-04-23	StephenH	-	3	3	3	*SP*
2010-04-23	ChristinaH	-	4	-	4	-
2010-04-23	SilvanaF	I had one obligation and no bananas to throw, although I do not know why I was not permitted to climb the tree and pick a banana since they were all in black!	-	-	-	-
2010-04-23	AnnaC	There are too many rules and counter rules. I ended up playing what it let me not what I wanted. But it's been a bit of fun.	2	-	2	-
2010-04-23	YokoO	-	4	-	3	-

C.3 Post-game questionnaire

C.3.1 Questionnaire form

About the game

Understanding the game concept When you first learned about the game, how easy was it understand the concept of the game?
(1 = Difficult, 5 = Easy)

The dynamic rule system Did having a system of changeable rules make the game more interesting (better) or too confusing (worse)?
(1 = Too confusing, 5 = Interesting)

Balance between rules and actions Do you think a good balance was achieved between the 'rules' aspect of the game and the simple actions (climb up/down, pick/throw banana)?
(1 = Too rule-focused (complicated), 5 = Too action-focused (mundane))

About the language used

Generated text When reading the rules set by other players, were they easy to understand or did they tend to be ambiguous?
(1 = Ambiguous, 5 = Very understandable)

Composing rules When creating new rules, how easy was it to express what you wanted to say?
(1 = Difficult, 5 = Easy)

Input method Which input method would you say you preferred to use when composing new rules?

- Suggest Panel
- Fridge Magnets
- No preference

Why? Reasons for your choice in the previous question (optional).
(Free-text box)

Would you have preferred a blank text box? Instead of the Suggest Panel or Fridge Magnets, would you have preferred a blank box where you could type anything you liked?

- Yes, a blank text box would have been better
- No, the methods provided were adequate

Language barrier Overall, how much would you say the language used in the game (i.e. the wording of the rules) was a barrier to your participation and enjoyment?
(1 = Language was a big problem, 5 = Language was not a problem at all)

Others

Comments Any other comments you may have about the game in general.
(Free-text box)

C.3.2 Questionnaire responses

1. *C* — Initial understanding of the game concept
2. *R* — Reaction to the self-amending rules system
3. *B* — Balance between rule-amendment and simple actions
4. *G* — Quality of generated text
5. *I* — Expressivity of input methods
6. *M* — Preferred input method, where:
   - *SP* — Suggest Panel
   - *FM* — Fridge Magnets
   - *N/P* — No preference
7. *F* — Preferability of a free-text input method (Yes/No)
8. *L* — Language used as a barrier to playability
9. Why/Comments — Combines the *Why?* and *Comments* fields together.

The user comments have been corrected for typographical errors.

Table C.3: Post-game questionnaire responses.

User	C^1	R^2	B^3	G^4	I^5	M^6	F^7	L^8	Why/Comments^9
AnnaC	3	3	1	2	4	*SP*	No	4	Why? It was the first one to come up, so did not bother to use the other since I found this method easy. The fridge-magnet method was just as easy to use. Comments If there had been a blank text box to make rules, the possibilities would have been endless and the rules more difficult to follow.
DianneB	3	4	3	4	4	*SP*	No	5	-
MichaelP	4	4	2	4	5	*SP*	No	5	Why? The Drop down suggestions gave a variety of options which you could easily choose from in order to get you point across.
GarethF	3	4	2	4	5	*FM*	No	4	-
NicolaVH	4	1	3	4	3	*FM*	No	4	Why? I only managed to play once and I used the fridge magnets option. I didn't get to try the drop-down menu.

JoeC	4	5	2	2	5	*SP*	Yes	5	Why? It seemed that there were more options available to me using this method. Comments In general, I found the game too negative. Although I won, this was mainly by stopping everyone else from getting anywhere. It would be better if the 'allowed' took precedence over the 'forbidden'.
YokoO	3	5	3	4	5	*FM*	No	5	Why? Fridge Magnets method was actually helpful in creating interesting rules.
MikeAV	4	4	3	5	5	*SP*	No	5	-
ChristinaH	4	5	3	5	5	*SP*	Yes	4	Comments With writing the rules - it was good to have the drop-down suggestions, especially for the first few goes, but once you get the hang of it it would have been better to have the option of having a blank box - maybe the option to choose if you want 'predictive rules' or not. Overall, the game was quite hard to understand at first - mainly because I wasn't sure what I was allowed/not allowed to do - but saying that I never read the help.
CharlesD	2	3	2	3	3	*N/P*	No	5	Why? Both were equally confusing to me but then I jumped in without reading the instruction book. Then went back to the book to try get an idea, then went back to the game to try figure it out by playing.
RichardH	2	4	3	3	2	*N/P*	Yes	3	-
StephenH	3	3	1	4	4	*SP*	No	5	-
JeanLucP	4	2	2	5	5	*SP*	No	4	Why? The method was more streamlined and quick to use.
SilvanaF	2	3	2	4	5	*SP*	No	5	Why? Because there was no need to worry about how to word the rules! Comments Just wish we had more time! It would have been fun to see how things would have developed! Anyway it was fun. Good luck!

Appendix D

Selected code fragments

A selection of source code snippets from the most important areas of the project. The snippets in this appendix only constitute a small part of the entire project. For clarity, sections have been left out and comments omitted. For full versions of the source code, please refer to the included CD.

D.1 Game evaluator

D.1.1 Contract logic grammar

```
module ContractGrammar where

data Time =
      T_Time Integer
    | T_None

data GameState = GameState {
      gs_Name     :: String
    , gs_Time     :: Integer
    , gs_Points   :: Points
    , gs_Players  :: [Player]
    , gs_Rules    :: [Rule]
    , gs_Pending  :: [Rule]
    , gs_Schedule :: [Event]
    , gs_History  :: [Event]
}

type PlayerName = String
type Points = Integer
type Level = Integer
data Player = P_Player {
      p_Name    :: PlayerName
    , p_Points  :: Points
    , p_Level   :: Level
}

type RuleID = Integer
data Rule = Rule RuleID Contract

data Contract =
      C_Empty
    | C_Deontic DeonticExp
    | C_Choice [Contract]
    | C_Always Time Time Contract
    | C_Sometimes Time Time Contract
    | C_Conditional DeonticExp Contract Contract
    | C_Query Query Contract Contract

data DeonticExp =
      DE_Obliged   PlayerName Activity
    | DE_Permitted PlayerName Activity
    | DE_Forbidden PlayerName Activity

data Query =
      Q_Not Query
    | Q_Conjunction Query Query
    | Q_Disjunction Query Query
    | Q_PointsGt PlayerName Points
    | Q_PointsEq PlayerName Points
    | Q_PointsLt PlayerName Points
    | Q_LevelEq PlayerName Level

data Event =
      E_None
    | E_Activity PlayerName Activity
    | E_Time Time

data Activity =
      A_Empty
    | A_Action Action
    | A_Choice [Activity] -- choice / disjunction
    | A_Concurrent [Activity] -- concurrency / conjunction

data Action =
      An_ClimbUp
    | An_ClimbDown
    | An_PickBanana
    | An_ThrowBanana PlayerName
    | An_Enact Rule
    | An_Abolish Rule
```

D.1.2 XML gamestate representation

Schema

This XML schema was created with the help of HiT Software's online XML Tools[1].

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Individual elements -->

  <xs:element name="int">
    <xs:complexType>
      <xs:attribute name="value" type="xs:integer" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="time">
    <xs:complexType>
      <xs:attribute name="value" type="xs:integer" use="optional" />
    </xs:complexType>
  </xs:element>

  <xs:element name="player">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="points" type="xs:integer" use="optional" />
      <xs:attribute name="level" type="xs:integer" use="optional" />
    </xs:complexType>
  </xs:element>

  <xs:element name="rule">
    <xs:complexType>
      <xs:attribute name="id" type="xs:integer" use="required" />
      <xs:sequence>
        <xs:element ref="contract" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="contract">
    <xs:complexType>
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="empty" />
            <xs:enumeration value="deontic" />
```

[1] http://www.hitsw.com/xml_utilites/

```
          <xs:enumeration value="choice" />
          <xs:enumeration value="always" />
          <xs:enumeration value="sometimes" />
          <xs:enumeration value="conditional" />
          <xs:enumeration value="query" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:choice>
      <xs:element ref="time" />
      <xs:element ref="query" />
      <xs:element ref="deontic" />
      <xs:element ref="contract" />
      <xs:element ref="contracts" />
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="contracts">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="contract" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="deontic">
  <xs:complexType>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="forbidden" />
          <xs:enumeration value="obliged" />
          <xs:enumeration value="permitted" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:sequence>
      <xs:element ref="player" />
      <xs:element ref="activity" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="query">
  <xs:complexType>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
```

```
        <xs:restriction base="xs:string">
          <xs:enumeration value="not" />
          <xs:enumeration value="conjunction" />
          <xs:enumeration value="disjunction" />
          <xs:enumeration value="points_gt" />
          <xs:enumeration value="points_eq" />
          <xs:enumeration value="points_lt" />
          <xs:enumeration value="level_eq" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:choice>
      <xs:sequence>
        <xs:element ref="query" maxOccurs="2" />
      </xs:sequence>
      <xs:sequence>
        <xs:element ref="player" />
        <xs:element ref="int" />
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="activity">
  <xs:complexType>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="empty" />
          <xs:enumeration value="action" />
          <xs:enumeration value="choice" />
          <xs:enumeration value="concurrent" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:choice>
      <xs:sequence>
        <xs:element ref="activity" minOccurs="1" />
      </xs:sequence>
      <xs:sequence>
        <xs:element ref="action" />
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="action">
  <xs:complexType>
```

```
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="enact" />
          <xs:enumeration value="abolish" />
          <xs:enumeration value="climb_down" />
          <xs:enumeration value="climb_up" />
          <xs:enumeration value="pick_banana" />
          <xs:enumeration value="throw_banana" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:choice>
      <xs:element ref="player" />
      <xs:element ref="rule" />
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="event">
  <xs:complexType>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="none" />
          <xs:enumeration value="time" />
          <xs:enumeration value="activity" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:choice>
      <xs:element ref="time" />
      <xs:sequence>
          <xs:element ref="player" />
          <xs:element ref="activity" />
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:element>

<!-- Overall gamestate structure -->

<xs:element name="gamestate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="players" />
      <xs:element ref="rules" />
      <xs:element ref="pending" />
      <xs:element ref="schedule" />
      <xs:element ref="history" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="points" type="xs:integer" use="required" />
    <xs:attribute name="time" type="xs:integer" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="players">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="player" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="rules">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="pending">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="schedule">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="event" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="history">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="event" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
</xs:schema>
```

Gamestate example

```
<gamestate name="Banana Bonanza" points="27" time="3">
  <players>
    <player level="2" name="JohnL" points="1" />
    <player level="1" name="RingoS" points="5" />
    <player level="0" name="GeorgeH" points="0" />
  </players>
  <rules>
    <rule id="101">
      <contract type="deontic">
        <deontic type="permitted">
          <player name="GENERIC" />
          <activity type="action">
            <action type="climb_up" />
          </activity>
        </deontic>
      </contract>
    </rule>
    <rule id="201">
      <contract type="sometimes">
        <time />
        <time value="5" />
        <contract type="deontic">
          <deontic type="obliged">
            <player name="RingoS" />
            <activity type="action">
              <action type="pick_banana" />
            </activity>
          </deontic>
        </contract>
      </contract>
    </rule>
  </rules>
  <pending>
    <rule id="20101">
      <contract type="deontic">
        <deontic type="obliged">
          <player name="RingoS" />
          <activity type="empty" />
        </deontic>
      </contract>
    </rule>
  </pending>
  <schedule>
    <event type="time">
      <time value="1" />
    </event>
  </schedule>
  <history>
    <event type="activity">
      <player name="GeorgeH" />
      <activity type="action">
        <action type="climb_up" />
      </activity>
    </event>
  </history>
</gamestate>
```

D.2 Grammatical Framework

D.2.1 Abstract contract grammar

```
abstract Contract = {
    flags startcat = Contract ;
    flags coding = utf8 ;

    cat
        Time ;
        PlayerName ; Level ;
        Contract ; [Contract]{2} ;
        DeonticExp ;
        Query ;
        Activity ; [Activity]{2} ;
        Action ;

    fun
        T_Time : Int -> Time ;
        T_None : Time ;

        P_Player : String -> PlayerName ;
        P_Generic : PlayerName ;
        L_0, L_1, L_2, L_3 : Level;

        C_Empty : Contract ;
        C_Deontic : DeonticExp -> Contract ;
        C_Choice : [Contract] -> Contract ;
        C_Always : Time -> Time -> Contract -> Contract ;
        C_Sometimes : Time -> Time -> Contract -> Contract ;
        C_Conditional : DeonticExp -> Contract -> Contract -> Contract ;
        C_Query : Query -> Contract -> Contract -> Contract ;
        DE_Obliged : PlayerName -> Activity -> DeonticExp ;
        DE_Permitted : PlayerName -> Activity -> DeonticExp ;
        DE_Forbidden : PlayerName -> Activity -> DeonticExp ;

        Q_Not : Query -> Query ;
        Q_Conjunction : Query -> Query -> Query ;
        Q_Disjunction : Query -> Query -> Query ;
        Q_PointsGt : PlayerName -> Int -> Query ;
        Q_PointsEq : PlayerName -> Int -> Query ;
        Q_PointsLt : PlayerName -> Int -> Query ;
        Q_LevelEq : PlayerName -> Level -> Query ;

        A_Empty : Activity ;
        A_Action : Action -> Activity ;
        A_Choice : [Activity] -> Activity ;
        A_Concurrent : [Activity] -> Activity ;

        An_ClimbUp : Action ;
        An_ClimbDown : Action ;
        An_PickBanana : Action ;
        An_ThrowBanana : PlayerName -> Action ;
        An_Enact : Action ;
        An_Abolish : Action ;
}
```

D.2.2 complete() JavaScript extension

This code extends the Parser class as defined in the GF JavaScript library by adding to it the complete() function. To achieve this, the internal ParseState class is also extended in a similar way.

```
/**
 * Generate list of suggestions given an input string
 */
Parser.prototype.complete = function (input, cat) {

    // Parameter defaults
    if (input == null) input = "";
    if (cat == null) cat = grammar.abstract.startcat;

    // Tokenise input string & remove empty tokens
    var tokens = input.trim().split(" ");
    for (var i = tokens.length - 1; i >= 0; i--) {
        if (tokens[i] == "") { tokens.splice(i, 1); }
    }

    // Init parse state objects.
    // ps2 is used for testing if final token is parsable or not.
    var ps = new ParseState(this, cat);
    var ps2 = new ParseState(this, cat);

    // Iterate over tokens, feed one by one to parser
    // See which can be consumed (complete a category)
    var consumed = new Array();
    var remaining = new Array();
    var nextUnconsumedIndex = 0;
    // Get value for nextUnconsumedIndex
    for (var i = 0; i < tokens.length; i++) {
        if (ps2.next(tokens[i])) {
            if (ps2.items.value != null) {
                nextUnconsumedIndex = i + 1;
            }
            // else keep going; we may reach another consumable point
        } else {
            break;
        }
    }
    delete(ps2);

    // Consume up until nextUnconsumedIndex
    for (var i = 0; i < nextUnconsumedIndex; i++) {
        var token = tokens.shift();
        ps.next(token);
        consumed.push(token);
    }
    remaining = tokens; // whatever's left

    // Parse is successful so far, now get suggestions
    var acc = ps.complete(remaining);

    // Put into suggestion list, avoiding duplicates
    var suggs = new Array();
    var suggStrings = new Array();
    for (var a = 0; a < acc.length; a++) {
        var s = acc[a].join();
        if (!arrayContains(s, suggStrings)) {
            suggs.push(acc[a]);
            suggStrings.push(s);
        }
    }
    delete(suggStrings);

    // Handle special cases, ie: player
    // (Literals just remain as <string>, <int>, <float>)
    for (var s = 0; s < suggs.length ; s++) {
        switch(suggs[s][0]) {
            case "player":
                suggs[s][0] = "<player>";
                break;
        }
    }

    // Note: return used tokens too
    return { "consumed" : consumed, "suggestions" : suggs };
}

/**
 * For a ParseState and a partial input, return all possible completions
 * Based closely on ParseState.next()
 * currentTokens could be empty or a partial string
 */
ParseState.prototype.complete = function (currentTokens) {

    // Initialise accumulator for suggestions
    accTokens = new Array();

    this.process(
        // Items
        this.items.value,

        // Deal with literal categories
        function (fid) {
            switch (fid) {
                // String
                case -1:
                    accTokens.push(["<string>"]);
                    return;
                // Integer
                case -2:
                    accTokens.push(["<int>"]);
                    return;
                // Float
                case -3:
                    accTokens.push(["<float>"]);
                    return;
            }
            return null;
        },

        /**
         * Takes an array of tokens and populates accumulator
         */
        function (tokens, item) {
            // Convert tokens from object to list
            var tokens1 = new Array();
            for (var i = 0; i < tokens.length; i++) tokens1[i] = tokens[i];
            if (currentTokens == "" || tokens1.join(" ").indexOf(
                currentTokens.join(" ")) == 0) {
                accTokens.push(tokens1);
            }
        }
    );

    // Return matches
    return accTokens;
}
```

Glossary of terms

ACE Attempto Controlled English (Fuchs et al., 2008)

AJAX Asynchronous Java And XML

APE Attempto Parsing Engine (Fuchs et al., 2008)

API Application Programming Interface

BNF Backus-Naur Form

CASE Computer-Aided Software Engineering

CELT Controlled English to Logic Translation (Pease and Murray, 2003)

CFG Context-Free Grammar

CLAN $\mathcal{CL}$ ANalyser (Fenech et al., 2009c)

CL Contract Logic (in the general sense). Note that $\mathcal{CL}$ refers to the specific *contract language* as defined in (Prisacariu and Schneider, 2007).

CNL Controlled Natural Language

CoCoME Common Component Modelling Example

CPE Computer Processable English (Pulman, 1996)

CSP Communicating Sequential Processes

CTD Contrary-To-Duty

CTP Contrary-To-Permission

CTL Computation Tree Logic

DCG Definite Clause Grammar (Pereira and Warren, 1980)

DRS Discourse Representation Structure

FM Fridge Magnets (guided input method)

Gamestate An object or data structure which encodes the entire state of a game which is currently in play. Includes such things as players' points, active rules and past events.

GF Grammatical Framework (Ranta, 2004)

GHC Glasgow Haskell Compiler

GWT Google Web Toolkit

GPL GNU General Public License. For details visit `http://www.gnu.org/licenses/gpl.html`

HPSG Head-driven Phrase Structure Grammar (Pollard and Sag, 1994)

I/O Input / Output

ISP Internet Service Provider

JPA Java Persistence API

JSNI JavaScript Native Interface

kb/s Kilobits per second

KIF Knowledge Interchange Format

LFG Lexical Functional Grammars

LGPL GNU Lesser General Public License. For details visit `http://www.gnu.org/licenses/lgpl.html`

LKIF Legal Knowledge Interchange Format

LTL Linear Temporal Logic

MT Machine Translation

NL Natural Language

NLP Natural Language Processing

NP Noun Phrase (part of speech)

OCL Object Constraint Language

OO Object-Oriented

OPP Obligation, Permission and Prohibition

OWL Web Ontology Language

PENG Processable English (Schwitter, 2002)

POJO Plain Old Java Object

PDL Propositional Dynamic Logic

QoS Quality-of-Service

RACE Attempto Reasoner tool for the ACE language (Fuchs et al., 2008).

RPC Remote Procedure Call

SDL Standard Deontic Logic

SLA Service-Level Agreement

SOA Service-Oriented Architecture

SP Suggest Panel (guided input method)

SWRL Semantic Web Rule Language

TL Temporal Logic

UI User Interface

UML Unified Modelling Language

VP Verb Phrase (part of speech)

XML Extensible Markup Language

Bibliography

Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL '09)*, number April, pages 69–76, Morristown, NJ, USA, 2009. Association for Computational Linguistics.

Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006.

Ron Cole, Joseph Mariani, Hans Uszkoreit, Giovanni Batista Varile, Annie Zaenen, Antonio Zampolli, and Victor Zue. Survey of the State of the Art in Human Language Technology, 1997. URL `http://www.dfki.de/~{}hansu/HLT-Survey.pdf`.

Robert Dale, Hermann Moisl, and H. L. Somers. *Handbook of natural language processing*. CRC Press, 2000.

Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Conflict Analysis of Deontic Contracts. In *Workshop in ICT (WICT '08)*. University of Malta, 2008.

Stephen Fenech, Gordon J. Pace, Joseph C. Okika, Anders P. Ravn, and Gerardo Schneider. On the Specification of Full Contracts. *Electronic Notes in Theoretical Computer Science*, 253(1):39–55, 2009a.

Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. *CLAN: A Tool for Contract Analysis and Conflict Discovery*, volume 5799 of *Lecture Notes in Computer Science*, pages 90–96. Springer, Berlin, Heidelberg, 2009b.

Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. *Automatic Conflict Detection on Contracts*, volume 5684 of *Lecture Notes in Computer Science*, pages 200–214. Springer, Berlin, Heidelberg, 2009c.

Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. *Attempto Controlled English for Knowledge Representation*, volume 5224 of *Lecture Notes in Computer Science*, pages 104–124. Springer, Berlin, Heidelberg, 2008.

REINER HÄHNLE, KRISTOFER JOHANNISSON, AND AARNE RANTA. *An Authoring Tool for Informal and Formal Requirements Specifications*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2002.

THOMAS HALLGREN AND AARNE RANTA. *An extensible proof text editor*, volume 1955 of *Lecture Notes in Computer Science*, pages 70–84. Springer Verlag, Heidelberg, 2000.

KRISTOFER JOHANNISSON. *Formal and Informal Software Specifications*. Phd thesis, Chalmers University of Technology and Göteborg University, 2005.

DANIEL JURAFSKY AND JAMES H. MARTIN. *Speech and Language Processing*. Pearson Education, New Jersey, 2nd edition, 2009.

KAAREL KALJURAND. Attempto Project. `http://attempto.ifi.uzh.ch/site/`, 2009. Accessed 21/05/2010.

SZYMON KLARMAN, RINKE HOEKSTRA, AND MARC BRON. Versions and Applicability of Concept Definitions in Legal Ontologies. In *Proceedings of OWL: Experiences and Directions (OWLED 2008 DC)*, 2008.

J.-J. CH. MEYER, F.P.M. DIGNUM, AND R.J. WIERINGA. The Paradoxes of Deontic Logic Revisited: A Computer Science Perspective. Technical Report UU-CS-1994-38, Dept. of Computer Science, Utrecht University, Utrecht, The Netherlands, 1994.

GORDON J. PACE AND MICHAEL ROSNER. *A Controlled Language for the Specification of Contracts*, volume 5972 of *Lecture Notes in Artificial Intelligence*. Springer, 2010.

GORDON J. PACE AND GERARDO SCHNEIDER. *Challenges in the Specification of Full Contracts*, volume 5423 of *Lecture Notes in Computer Science*, pages 292–306. Springer, Berlin, Heidelberg, 2009.

GORDON J. PACE, CRISTIAN PRISACARIU, AND GERARDO SCHNEIDER. Model Checking Contracts - a case study. In Namjoshi Kedar and Tomohiro Yoneda, editors, *5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, number 1, pages 82–97, Tokyo, 2007. Springer.

R. S. PATIL, R. E. FIKES, PETER F. PATEL-SCHNEIDER, D. MCKAY, T. FININ, T. GRUBER, AND R. NECHES. The DARPA Knowledge Sharing Effort: Progress Report. In *KR92 (Proceedings of the Third International Conference on Knowledge Representation and Reasoning)*, Palo Alto, 1992. Morgan Kaufmann.

ADAM PEASE AND WILLIAM MURRAY. An English to logic translator for ontology-based knowledge representation languages. In *International Conference on Natural Language Processing and Knowledge Engineering*, pages 777–783. IEEE, 2003.

Fernando C. N. Pereira and David H. D. Warren. Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial intelligence*, 13(2):231–278, 1980.

Mark E. Phair and Adam Bliss. PerlNomic: Rule Making and Enforcement in Digital Shared Spaces. In *Online Deliberation 2005 / DIAC-2005*, Stanford, CA, USA, 2005.

Carl Pollard and Ivan A. Sag. *Head-driven phrase structure grammar*. University of Chicago Press, Chicago, 1994.

Cristian Prisacariu and Gerardo Schneider. *A Formal Language for Electronic Contracts*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer, Berlin, Heidelberg, 2007.

Stephen G. Pulman. Controlled Language for Knowledge Representation. In *Proceedings of the first international workshop on controlled language applications, Katholieke Universiteit*, pages 233–242, 1996.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(02):145–189, 2004.

Aarne Ranta. The GF Resource Grammar Library. *Linguistic Issues in Language Technology*, 2(2), 2009.

Aarne Ranta and Krasimir Angelov. Implementing Controlled Languages in GF. In *CNL-2009, CEUR Workshop Proceedings*, 2009.

Rolf Schwitter. English as a Formal Specification Language. In *Proceedings of the Thirteenth International Workshop on Database and Expert Systems Applications (DEXA 2002)*, pages 228–232, Aix-en-Provence, France, 2002.

Stuart M. Shieber. The design of a computer language for linguistic information. *Proceedings of the 22nd annual meeting on Association for Computational Linguistics*, pages 362–366, 1984.

Peter Suber. *Nomic: A Game of Self-Amendment*. Peter Lang Publishing, 1990.

Bernard Vauquois. A survey of formal grammars and algorithms for recognition and transformation in mechanical translation. In *IFIP Congress (2)*, pages 1114–1122, 1968.

Gerard A. W. Vreeswijk. Formalizing Nomic: working on a theory of communication with modifiable rules of procedure. Technical Report CS 95-02, Dept. of Computer Science, University of Limburg, Maastricht, The Netherlands, 1995.

WebALT Consortium. Final Report of the WebALT Project, 2007. URL `http://webalt.math.helsinki.fi/`.