REPRESENTING GAME DIALOGUE AS EXPRESSIONS IN FIRST-ORDER LOGIC

(Thesis format: Monograph)

by

Kaylen Wheeler

Graduate Program in Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

© Kaylen Wheeler 2013

Abstract

Despite advancements in graphics, physics, and artificial intelligence, modern video games are still lacking in believable dialogue generation. The more complex and interactive stories in modern games may allow the player to experience different paths in dialogue trees, but such trees are still required to be manually created by authors. Recently, there has been research on methods of creating emergent believable behaviour, but these are lacking true dialogue construction capabilities. Because the mapping of natural language to meaningful computational representations (logical forms) is a difficult problem, an important first step may be to develop a means of representing in-game dialogue as logical expressions. This thesis introduces and describes a system for representing dialogue as first-order logic predicates, demonstrates its equivalence with current dialogue authoring techniques, and shows how this representation is more dynamic and flexible.

Keywords: Game Development, Believable Characters, Game Dialogue, Logic Programming

Contents

Abstract **ii**

List of Figures **vi**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Proposed Solution 2

1.3 Contributions 3

1.4 Thesis Overview 4

2 Background and Previous Work **5**

2.1 Believable Agent Frameworks 5

2.1.1 Believability 5

2.1.2 Frameworks and Techniques 6

2.2 Natural Language Processing (NLP) 7

2.2.1 NLP In Games 7

2.2.2 NLP Research 7

2.3 Current Dialogue Technologies 8

2.3.1 Linear Dialogue 9

2.3.2 Stitching 9

2.3.3 Dialogue Trees 12

2.4 Research Gap 14

3 Technical and Theoretical Background **15**

3.1 Computer Science 15

3.1.1 Believable Agents 15

Theory of Mind 15

3.1.2 Choice of Language and Frameworks 16

3.1.3 Homoiconicity 16

3.1.4 First-Class and Higher-order Functions 18

3.1.5 First-order Logic, MiniKanren, and core.logic 19

3.1.6 Higher-Order Predicates 26

3.1.7 Information Hiding with pldb 28

3.1.8 Explicit Hierarchies and Multimethods 30

3.1.9 A Note on Metadata and Type Annotations 32

3.2 Linguistics 33

3.2.1 Adjacency Pairs 33

4 Design and Implementation **34**

4.1 Overview and System Architecture 34

4.2 Design From Example: Knights and Knaves 36

4.2.1 Implementing Knights and Knaves 37

Thought Functions as Higher-Order Predicates 40

4.3 Adjacency Pairs and Polymorphism 46

4.3.1 Response Functions 46

4.3.2 Message Types 47

4.4 Modifiers 50

4.5 Rules 51

4.6 Combining the Components 55

5 Evaluation and Discussion **57**

5.1 Bobby and Sally 57
5.1.1 Premise 57
5.1.2 Concepts Demonstrated in This Scenario 58
5.1.3 Implementation 58
Message Structure 59
Sally's Thought Function 60
Sally's Rules 60
Bobby's AI 65
5.1.4 Analysis 65
5.2 Lovecraft Country 69
5.2.1 Premise 69
5.2.2 Concepts Demonstrated in this Scenario 69
5.3 Preparation 70
Simplifying the Dialogue Tree 70
5.3.1 Implementation 70
Data Representation 70
Example Run 74
Changing Agent Attributes 76
Adding Agents and Attributes 79
5.3.2 Analysis 82
5.4 Discussion 82

6 Conclusions **83**
6.1 Contributions 84
6.2 Future Work 84

Bibliography **86**

List of Figures

2.1 An example dialogue tree. 13

2.2 An example illustration of a dialogue tree. 14

3.1 Lisp expression and Lisp list. 17

3.2 An example of dynamically evaluating lisp lists. 18

3.3 A simple higher-order function. 19

3.4 Prolog code representing parent and grandparent relationships. 21

3.5 Core.logic code representing parent and grandparent relationships. 22

3.6 The arithmetic operators as well as the *is* operator are non-relational. 24

3.7 Use of non-relational operators in core.logic requires projection. 25

3.8 Shorthand for predicate definitions. 27

3.9 Example of higher-order predicates using the parent and grandparent relations. . 29

3.10 An example of multimethod polymorphism in Clojure. 31

3.11 An illustration of an arbitrary type hierarchy. 32

4.1 System architecture. 35

4.2 Knights and Knaves function definitions. 38

4.3 Knights and Knaves 39

4.4 Newly implemented core functionality to handle higher-order predicates. 41

4.5 Newly implemented functionality more specific to the Knights and Knaves scenario. 42

4.6 The effect of the new code on the interactive prompt. 43

4.7 Some example adjacency pairs from Wikipedia [24], with some modifications. . 47

4.8 Example response function execution. 48
4.9 Code examples for creating modifiers. 52
4.10 An example of a rule, combining a change predicate and a trigger. 53
4.11 An overview of the message processing cycle. 55

5.1 Sally's emotion levels. The red zone represents when she will start crying if asked out. 59
5.2 Sally's thought function. 61
5.3 The predicate portion of the coffee-inc rule, specifying preconditions and update functions. 63
5.4 The trigger portion of the coffee-inc rule, which can be examined by the AI. . . 64
5.5 Bobby and Sally's conversation. 66
5.6 Bobby and sally's conversation. 67
5.7 Bobby's AI process. 68
5.8 The simplified dialogue tree. 71
5.9 The simplified dialogue tree diagram, showing message groupings dotted-line boxes. 72
5.10 Example data map for the Mechanic character, used to represent potential external data model. 73
5.11 Example run of the "Lovecraft Country" scenario that follows the dialogue tree fairly accurately. 75
5.12 Variations on the first dialogue run, corresponding to possible variations in the dialogue tree. 77
5.13 The agent has no pre-existing beliefs about the player's job. 78
5.14 Further descriptions are possible if the threshold has changed. 78
5.15 Examples of variant symbols. 80
5.16 Mechanic's apprentice conversation. 81

Chapter 1

Introduction

In the relatively short time that they have existed, interactive computer games have advanced significantly. In the four decades since the first arcade game was released, microprocessors have literally increased computer clock speeds many thousands of times. This advancement has allowed for the advancement from simple two-dimensional graphics to near-photorealistic three-dimensional environments with realistic physical simulation and AI-controlled characters.

However, compared with the advancement in physical believability, the social and intellectual believability of computer controlled characters has lagged behind [13]. In particular, in-game dialogue has not yet been be simulated in sufficiently believable, dynamic, and emergent ways [21]. There are a number of successful techniques which provide an illusion of dynamicity, but these techniques are lacking in key areas.

Among the most commonly used of such techniques is the dialogue tree, which allows the user to traverse a path of pre-written dialogue lines by choosing how to respond to the computer-controlled player's lines (dialogue trees are further described in Section 2.3.3).

1.1 Problem Statement

As successful as the dialogue tree is, it has its limitations, the most important of which is that the lines of dialogue are, from the computer's perspective, arbitrary strings. For the computer, game dialogue has no meaning except in paths through graphs decorated by sequences of characters. In order to advance the state of the art in dynamicity and generativity of dialogue content, a new, more computationally meaningful representation of dialogue must be introduced. A computationally meaningful dialogue representation will allow programs to manipulate the dialogue, resulting in a much more dynamic and flexible dialogue system.

1.2 Proposed Solution

We propose that expressions in first-order logic be used to represent game dialogue. Such expressions can be used to represent expressions in human language, and in fact have been as the basis of an artificial human language called Lojban [17]. Additionally, first order logic has close connections to computational linguistics and formal grammars, which have been used in some game dialogue systems [8]. More importantly, they have some important properties which are absent in the handwritten strings of dialogue trees; they are:

Computationally Meaningful Expressions in first-order logic have the capacity to be represented as computational forms, which can be interpreted and executed in a logic inferencing system.

Unambiguous A single expression in first-order logic always produces the same answers in the same context. There is no dispute as to the "meaning" of the expression.

Well-Structured All such expressions can be represented in the form of syntax trees, and can therefore be analyzed, constructed, destructured, and modified by computational processes. (See Section 3.1.3.)

1.3 Contributions

The primary contribution of this work is the development of a prototype system implementing a novel approach to the representation of game dialogue as first-order logic expressions. The system, developed using the Clojure programming language [7] and the core.logic [16] implementation of the MiniKanren [6] logic programming language, is a stand-alone application, but is designed with the possibility of being embedded in another, larger application, such as a game.

In addition to being able to approximate the behaviour and capabilities of current dialogue technologies (such as dialogue trees), the system enables the emergence of new interactions from data without explicitly modifying the data. More specifically, whereas many current game authoring technologies are based upon the ability to script *instances* of agent behaviour, the current system allows the scripting of *patterns* of agent behaviour.

The incorporation of computationally meaningful and well-structured first-order logic expressions as a form of communication between agents enables the construction and interpretation of those expressions in a variety of unique ways. Through the creation of functions which modify the interpretation of logical expressions, it is possible to create psychosocial models of personality, emotions, and social relations, which affect the way in which agents respond or react to certain dialogue. Additionally, similar data may be re-used in a variety of ways in order to generate dialogue, and the means of dialogue generation can be altered by altering agent attributes.

Our aim is that the feasibility of this method of dialogue representation is fully and adequately demonstrated, so that a system such as this may be used as a basis for further research in this field.

1.4 Thesis Overview

In the next Chapter of this thesis, an overview of current techniques for dialogue creation in games is presented. In Chapter Three, theoretical background is provided of topics in computer science and linguistics that have contributed to this work. Chapter Four describes how those concepts were worked into the design and implementation of a first-order logic dialogue system using the Clojure programming language [7] and its implementation of the Minikanren logic language [5] [6] known as core.logic [16]. Finally, Chapter five demonstrates that a system based on first-order logic dialogue has much of the same potential for expressiveness as dialogue trees currently possess, but also possesses further potential for dynamicity and extensibility not currently inherent in dialogue trees.

Chapter 2

Background and Previous Work

2.1 Believable Agent Frameworks

A believable agent framework is any framework that simulates the social and emotional behaviours of humans in a believable way. Because the term "believable" is highly subjective, a variety of strategies have been used to create believability, and there are a number of software frameworks based on such strategies.

2.1.1 Believability

When attempting to simulate something as complex and subjectively variable as human behaviour, the questions of what is and is not "believable" become very difficult to answer.

Perhaps the first attempt at describing a way to determine believability of simulated human behaviour is the Turing Test [22]. This test proposes that a machine should be able to fool a human into believing that it is a human by participating in a conversation with it. To our knowledge, no machine has successfully passed the Turing test on a regular basis, and this is unlikely to be true for the foreseeable future.

Because criteria such as the Turing test are too strict to be practical (given the current state of technology), and because computer games necessarily involve an element of surreal-

ism, other interpretations of the term "believable" have been proposed. Acton [1] proposes that, rather than have computer games be fully believable, they should simply lead to willing suspension of disbelief.

Yet another perspective on believability, proposed by Loyall, suggests that believable agents demonstrate "reactivity to the environment, general responsiveness, and [existence] in unpredictable worlds with people and other agents" [8].

2.1.2 Frameworks and Techniques

A number of frameworks exist for the creation of believable agents, each of which covers different aspects of believable agent simulation.

One such system is *Comme il Faut* [14], a system geared toward so-called "social physics", which was used in the experimental game *Prom Week* [13]. The term "social physics" is an analogy to the use of physics-based gameplay and puzzles in modern games. Rather than attempting a completely accurate simulation of human social behaviour, a few limited, intuitive rules of social interaction in the game world are presented to the player to allow them to participate in interesting and engaging gameplay.

Another such system is the FearNot Affective Mind Architecture (FAtiMA) [9]. FAtiMA was developed initially as an engine for *FearNot*, a serious game aimed at teaching school children between ages 8 and 12 how to deal with bullying. FAtiMA has a number of modules, each based around modelling aspects of believable agents. These include modules for modelling Theory of Mind (TOM) [**?**], memory, emotional intelligence, and others.

Despite their advanced social modeling techniques, these systems have several shortcomings, particularly in the area of dialogue generation.[LEFT]

2.2 Natural Language Processing (NLP)

2.2.1 NLP In Games

In its simplest form, natural language input in video games dates back a very long time. Games such as Zork [2] enabled free-form text-based interaction between the user and the game. However, such systems were far from true natural language interaction, and simply scanned the string input for keywords to interpret as commands.

More recently, games such as Facade [12] [10] have attempted to make natural language interaction more immersive and believable, but still do not truly interpret natural language. The input is scanned for certain keywords, which are used as input into a complex dialogue graph. This dialogue graph, although highly dynamic and believable, took several hundred thousand lines of code to fully author [11].

After the early days of adventure games, natural language interaction in games has been uncommon. Most recent games that boast highly interactive and variable dialogue systems (such as Mass Effect [4]) have primarily relied on the players selecting from a finite set of options in order to traverse a dialogue tree (many so-called dialogue "trees" would be more appropriately called "dialogue graphs"). Because the technology of natural language synthesis is still very much in its infancy [27], it is difficult to provide players with believable interaction. The result of this is that games attempting to create immersive and believable behaviour through dialogue interactions require extensive authoring; designers must anticipate each possible situation and the effects it has on the game world and characters. As a result, the property of emergence is lost.

2.2.2 NLP Research

Natural language processing is still an ongoing area of research. Recent developments have led to more advanced search engines and data mining techniques, as well as computer-aided natural language translation (such as Google Translate). However, the problem of mapping natural

language sentences to meaningful forms that can be effectively interpreted by a computer is still a very open problem.

Zettlemoyer [27] has conducted some interesting research on the topic of mapping natural language sentences to λ calculus [19] expressions, which can be easily represented as sentences in first-order logic. His research makes use of combinatory categorical grammars (CCG) [20] generated by advanced machine learning techniques in order to parse sentences of various natural languages.

An example of a mapping of natural language to λ expressions is demonstrated below. Here, the sentences "What states border Texas?" is mapped to an equivalent λ expression. The expression takes as a parameter a single unbound variable x, which must satisfy the condition of being a state and bordering Texas. When the expression is used as a query to a knowledge base, x unifies with all values that satisfy both conditions.

"What states border Texas?"

$$\lambda x.(state(x) \wedge border(x, Texas))$$

$$x = \{NewMexico, Oklahoma, Arkansas, Louisiana\}$$

Although the specifics of the theory and implementation are beyond the scope of this thesis, the important realization is that natural language sentences can be transformed into these forms, as well as the fact that software exists that can accomplish this transformation [23]. Using software to translate logical expressions into natural language could allow the creation of language-independent dynamic dialogue systems that can be easily localized into different languages.

2.3 Current Dialogue Technologies

Dialogue has been part of video gaming since the first text adventure games were created and has taken on many forms since then. Some of the different methods of dialogue presentation

are created with different goals in mind; all are designed to convey information to the player, but not all are created to introduce believability. An overview and analysis of existing dialogue techniques follows, much of which is described in *Game Writing* [3].

2.3.1 Linear Dialogue

Ever since game hardware has first been able to play back believable voice and video data, many games have aimed to produce a so-called "cinematic" experience. Although many advances have been made towards producing dynamic real-time graphics and animations, few advances have contributed to the real-time creation of coherent and believable dialogue.

The result is that linear dialogue is used. Linear dialogue simply refers to dialogue that is played at certain points of progress in the game, regardless of the choices the player makes. Linear dialogue systems can scarcely be called dialogue systems at all, as they are not much more complex than video or audio playback. However, such systems are worth examining as a benchmark for believability. After all, their content is directly created by human authors who understand the subtle parameters of believability (whether or not they can define them explicitly). Linear dialogue, therefore, is the least dynamic, and perhaps the most believable of existing dialogue systems.

2.3.2 Stitching

When audio dialogue conforms to a particular structure, such that certain components of the dialogue can be interchanged while other components stay fixed, the interchanging of components is referred to as "stitching". The concept of stitching is best represented by a familiar example.

Many automated announcement systems, such as those in train stations or airports, use stitching to generate their announcements. Consider the following example:

"Train number ninety three from Windsor to Toronto will arrive at platform three in approximately five minutes."

While this is an accurate transcription of such an announcement, it may be better represented as follows:

"Train number **ninety three** from **Windsor** to **Toronto** will arrive at platform **three** in approximately **five** minutes."

Because of the discrete, "chopped-up" nature of the sentence, it is very obvious to human listeners that there's something wrong with this type of dialogue. It does not "flow" naturally, as it would if being spoken by a real person. Despite the fact that it does not create the illusion of humanity, this type of stitching serves its purpose; people can still clearly understand what's being said, even if they are fully aware that it being said by a machine.

Still, stitching techniques have met with some success in creating game dialogue, although it takes some extra effort to make the stitched dialogue appear believable. This is particularly used as a means of dynamically generating commentary for sports games, although this requires a great deal of effort to accomplish properly [3].

Ernest Adams, who wrote a chapter on interchangeable dialogue [3], recounts some of the difficulties inherent in dialogue splicing. Often, individual words or dialogue components must be recorded multiple times with subtle differences. For instance, if a word comes at the beginning, middle, or end of a declarative sentence, or at the end of an interrogative sentence, the intonation will differ considerably. Although this is not often noticed by people in day-to-

day speach, it is noticeable if deviated from. Therefore, multiple variations of single words must be used at different points in each sentence.

Take as an example a hypothetical sports game, in which the announcer's lines are constructed something like this:

- "___ gets the ball."
- "___ passes to ___."

In addition, there are a list of player names that can be substituted for the blank spaces:

- Smith
- Jones
- etc...

Imagine the two sentences with the gaps filled in:

- "Smith gets the ball."
- "Jones passes to Smith."

When reading the two sentences, if you listen carefully, you will find that the intonation on the two instances of "Smith" differs considerably. Using a single audio snippet to represent all instances when Smith's name comes up would make the dialogue sound unnatural. Therefore, multiple instances of each name must be recorded and used at the appropriate time.

Additionally, in everyday speech, words are not discreet; there are never clean breaks between words. This adds another dimension of variability that must be accounted for in recordings.

Also, players will tire of hearing repetitive and un-creative commentary. So, in a sports game for example, multiple lines are often recorded with effectively the same meaning. When

a player is tackled for example, variations such as "He'll feel that one in the morning" and "Oh, that's gotta hurt" may be recorded.

Ultimately dialogue stitching techniques, although believable in certain situations, are only applicable to those particular contexts. The dialogue, although it can be modified dynamically, must still have a basic pre-written structure.

Of course, some of the limitations of audio recording are not present if one chooses to represent game dialogue as text. However, the primary issue remains the same; dialogue must conform to a relatively rigid structure with small, interchangeable parts.

2.3.3 Dialogue Trees

The use of so-called "dialogue trees" is considered the state of the art in modern dynamic game dialogue [3]. In fact, the term "Tree" is often a misnomer. Wikipedia defines a "tree" as an "undirected graph in which any two vertices are connected by exactly one simple path" [25].

Most dialogue trees are in fact directed graphs which may or may not have cycles. However, the term "dialogue tree" will be used here for consistency with other literature.

Fundamentally, a dialogue tree is a graph, and is similar in structure to a finite-state machine. Each node represents a line of dialogue that a non-player character says to the player, and each transition represents a possible choice made by the player from that node. An example dialogue tree, from the book *Game Writing: Narrative Skills for Video Games* [3] is shown in Figure 2.1 and Figure 2.2.

Dialogue trees, although commonly used in current video games, suffer from some fundamental limitations. In particular, each node and each transition must be manually created by the game authors. This means that there is a direct correlation between the amount of variability in dialogue and the amount of work put into authoring. In fact, the process of dialogue tree authoring has been described by some authors as "burdensome" and "highly constrained" [14].

M1 : "You a cop?"

- A "No, I'm a claims adjuster." [go to M3]
- B "Private investigator. I have a few questions." [go to M2]
- C "Yeah, I'm a plain-clothes cop. I've got questions for you." [go to M2]

M2 : "Nah, you're dressed too smart. I reckon you work for the insurers."

- A "Lucky guess." [go to M3]
- B "That's for me to know, and you to guess at." [go to M3]

M3 : "I reckon this counts as an act of God."

- A "What happened?" [go to M4]
- B "I don't believe in God. I reckon this was an act of man." [go to M3a]

M3a : "Yeah, right – you didn't see those lights coming down from the sky – you didn't see those bodies... the blood..." [go to M4]

M4 : "It was horrible... I don't even want to think about it."

- A "I'll come back tomorrow. Give you a chance to recover."[go to M9]
- B "I need to know what happened, or we can write off your claim here and now." [go to M5]

M5 : "I drove up to the farm that night. It was quiet. I mean, dead quiet. Even the cicadas weren't making a sound, and those critters are deafening most nights." [go to M6]

M6 : "When I got close, the engine just died on me I started checking the sparks, and then it happened." [go to M7]

M7 : "Lights, I mean, something glowing, just pouring down from the heavens on the farmhouse. It was beautiful and terrible and... I don't know. But, when I go to the farm... My God..."

- A "This is obviously very difficult for you. I'll come back tomorrow." [go to M9]
- B "Go on. You're just getting to the good bit." [go to M8]

M8 : "They were lying around outside the farmhouse... I thought maybe a cow had been struck by lightning. But they weren't. They didn't even look human..." [go to M9]

M9 : "I'm sorry... I've never... I just..."

- A "This has obviously been hard on you. Get a good night's sleep... I'll se you in the morning." [End]
- B "Pull yourself together, man! I'll come back tomorrow morning - but you'd better not start blubbering when I come back, or you can kiss the insurance money goodbye." [End]

Figure 2.1: An example dialogue tree.

Figure 2.2: An example illustration of a dialogue tree.

2.4 Research Gap

Despite the many advances in these different areas, there has been little research focusing on how to represent and generate meaning in dialogue without natural language. NLP researchers tend to find ways of representing and translating meaning from natural language to a computational form, while research in game dialogue tends to be about ways that natural language can be used in believable ways.

If fully dynamic dialogue is ever to be achieved in games, a synthesis of these two research areas is necessary. In particular, dialogue must be generated, interpreted, and understood in its computational form before being translated into natural language.

Chapter 3

Technical and Theoretical Background

The design of the system brings together a number of concepts from topics as diverse as functional and logic programming, object orientation, and language pragmatics. It is important for the reader to understand these concepts before the design and implementation can be properly understood. If any of the concepts are already familiar, feel free to skip any of these sections.

3.1 Computer Science

3.1.1 Believable Agents

Theory of Mind

In a multi-agent system such as this, each agent has its own thoughts, goals, and behaviour. The term *Theory of Mind*, or TOM, is used to describe a means for agents to not only maintain information about their own beliefs and behaviour, but to have information about the beliefs and behaviours of other agents [**?**].

3.1.2 Choice of Language and Frameworks

Because of the nature of this project, a programming language that allowed a great deal of flexibility and potential for unconventional and novel methods of abstraction was required. Also, due to the requirements for first-order logic processing, access to an inference engine was required.

The language chosen for implementation was Clojure [7]. Clojure is a relatively new (created in 2007) dialect of the Lisp programming language. It emphasizes immutable data structures and functional programming style over imperative coding, and uses the Java Virtual Machine (JVM) as an execution environment.

Because of its roots in Lisp, Clojure allows for multiple methods of abstraction. Most important among them is the capacity to use higher-order functions (functions that accept other functions as arguments) as well as its unique object polymorphism capabilities and its homoiconicity. Homoiconicity refers to the fact that the core data structures of the language are also used to represent the executable code of the language. Therefore, it is possible to construct executable code at run-time, as well as to write macros (functions that transform code at compile-time) that allow for the creation of embedded domain-specific languages.

One such domain-specific language comes from the clojure.core.logic library [16], a Clojure-based adaptation of the MiniKanren language [6] [5] which was initially written in Scheme, another dialect of Lisp. The core.logic library allows for native Clojure functions to be used as logical predicates, and includes a powerful and flexible inferencing system.

3.1.3 Homoiconicity

This section explains the concepts of homoiconicity. If you are already familiar with this, feel free to skip this section.

Homoiconicity, also referred to as the “code-as-data” philosophy, is the ability of certain programming languages to use the same structures to represent data and executable code. Di-

alects of Lisp are some of the most common homoiconic languages, although homoiconicity is also present in other languages, such as Prolog. In this section, we will deal with Lisp, and Clojure more specifically.

```
;; Here is Lisp Code!
;; Computes 1 + 2 + 3 * (5 - 4)
(+ 1 2 (* 3 (- 5 4)))

;; Here is a Lisp list!
;; It contains numbers and another list
'(1 2 3 (4 5) 6 7)
```

Figure 3.1: Lisp expression and Lisp list.

Figure 3.1 shows two examples of Lisp code. On the first line is a mathematical expression. In order to understand this, it is worth noting that all Lisp code is written in prefix notation. That is, the function always comes first. No operator-precedence rules exist in Lisp. The most deeply nested expressions are evaluated first, and their results are passed upward.

The first line shows an example of a mathematical expression, using addition (+), multiplication (*), and subtraction (-) operators. The second line shows an example of a list. In this example, most of the list elements are numbers, except for the fourth element, '(4 5), which is itself another list. Aggregate data structures in Lisp are not strongly typed, and thus allow for arbitrary levels of nesting (i.e. lists within lists).

It is worth noting the similarity between the list structure and the executable code. In fact, the executable code *is* a list structure. The primary difference between the two is that the executable code conforms to a particular pattern: the first element of each list is a function. The rules for code execution in Lisp are simple: take the function at the head of the list and apply it to the arguments in the remainder of the list. If one of those arguments is itself a list, evaluate that list first.

One notable exception is the specification of a list as a data object. The single quote prefixing the second list in Figure 3.1 is the quote operator. It preserves lists as data and prevents them from being executed. Because functions are themselves first-class objects (i.e. they can

be constructed at run-time, stored in other data structures, passed as arguments, etc.), arbitrary lists can be constructed that contain functions. These lists can then be evaluated.

```
;; Here is a lisp list that could be executed!
(def ex-list '(+ 1 2 (* 3 (- 5 4))))

;; And here is its execution
(eval ex-list) ;; Returns 6

;; Here it is modified
;; Takes the tail(rest) of the original list, and prepends the
;; - operator in place of the + operator
(def mod-ex-list (cons - (rest ex-list))) ;;Yields (- 1 2 (* 3 (- 5 4)))

;; Evaluated again
(eval mod-ex-list) ;; Returns -4
```

Figure 3.2: An example of dynamically evaluating lisp lists.

As Figure 3.2 shows, it is possible to evaluate lists at run-time as well as modify those lists to yield different results. Although this is a simple example, arbitrarily complex structures can be created and evaluated. Special functions can be written which take code structures as input and produce code structures as output, and can be run in an intermediary step of the Lisp interpreter. Such functions are known as "macros", and are an important component in the creation of custom language (such as MiniKanren) as subsets of Lisp dialects.

3.1.4 First-Class and Higher-order Functions

This section explains the concepts of first-class and higher-order functions. If you are already familiar with these, feel free to skip this section.

In programming language design, the term "first class citizen" refers to any object that [26]

- can be stored in variables and data structures,
- can be passed as a parameter to a subroutine,

- can be returned as the result of a subroutine,
- can be constructed at run-time, and
- has intrinsic identity (independent of any given name).

The term "first class function" refers to functions that satisfy the above properties. Clojure, as well as other Lisps, are known for their built-in support of first-class functions.

The term "higher-order function" refers to any function which takes as its argument another function. A very simple and common example of such a function is the *map* function, a staple of many functional programming languages. The *map* function take a function of one argument and applies it to each element of a list, returning the results in a new list.

```
;; Here is a square function
;; It takes one argument and multiplies it by itself
(defn sq [x] (* x x))

;; Here is an inverse function
;; Divides one by x
(defn inv [x] (/ 1 x))

;; We will map the functions over a list of integers
(map sq '(1 2 3 4 5)) ;; Returns (1 4 9 16 25)
                      ;; These are the squares of 1 to 5
(map inv '(1 2 3 4 5)) ;; Returns (1 1/2 1/3 1/4 1/5)
                       ;; These are the inverses of 1 to 5
```

Figure 3.3: A simple higher-order function.

Figure 3.3 shows how map can be applied to the same list, using different functions to produce different results. This level of abstraction allows dynamic manipulation of data without having to re-write boilerplate code such as loops.

3.1.5 First-order Logic, MiniKanren, and core.logic

Without going into too much detail, this section will introduce some key information about first-order logic, and in particular the clojure.core.logic library and the MiniKanren language

on which it is based. Some familiarity with Prolog will be helpful in understanding the concepts presented here, but it is not necessary.

MiniKanren is a domain-specific language of the Lisp dialect Scheme, and is introduced and described in The Reasoned Schemer [6]. The core.logic library [16] for the Clojure programming language is a port of MiniKanren from Scheme to Clojure, with a few adjustments. Core.logic introduces logic variables, as well as a number of functions and macros to work with them. By doing this, logical predicates can be modeled as native Clojure functions (specifically, such functions take logic variables as inputs and produce functions of variable substitutions as outputs).

Motivation for Use It is important, before we go further, to justify our choice of using core.logic over another, more conventional and well-known logic language such as Prolog. The primary motivation was that Clojure, and thus core.logic, run on the JVM, while Prolog is its own, closed system. While Prolog has libraries for interacting with C++ and Java, the data representation of objects in the Prolog system is fundamentally different from that of other languages, and thus conversions of the data between languages are necessary to make full use of the Prolog inferencing engine. Core.logic, however, has the capability to store any Clojure object (and therefore, any Java object and any first-class function) in its database. Additionally, it is possible to extend the system by introducing new operators. Methods can be written that can convert any Clojure sequence to a set of substitutions for a logic variable. Since the ability to convert Java Collections into sequences is included in the default libraries of Clojure, this effectively means that, with minimal effort, almost all structured data from any Java program can be queried with this system.

The use of core.logic will be demonstrated with an example. In this example, simple parent-child relationships will be used. Consider the code represented in Figures 3.4 and 3.5.

Both Figures 3.4 and 3.5 represent the same logical predicates and relations, but in different programming languages. Because Prolog is more straightforward and is more widely known

```
parent(dave,kaylen).
parent(frank, dave).

% X is parent of Z, and Z is parent of Y
% Therefore X is grandparent of Y
grandparent(X,Y):- parent(X,Z),
                   parent(Z,Y).

% ====================================
% Interactive prompt
?- grandparent(frank, dave)
False.

?- grandparent(frank, kaylen)
True.

?- grandparent(X,Y).
X = frank,
Y = kaylen.
```

Figure 3.4: Prolog code representing parent and grandparent relationships.

than core.logic, we will begin with the Prolog example.

In the Prolog example, two facts are asserted: "Dave is the parent of Kaylen", and "Frank is the parent of Dave". Additionally, a predicate is defined that uses the parent relation to create a grandparent relation. The concept is simple: given some X and some Y, there must exist a Z such that X is the parent of Z and Z is the parent of Y. If these conditions hold, then X is said to be the grandparent of Y.

Executing the grandparent relation in the interactive prompt, we see a few different results. First of all, we can ask the system true/false questions. We can ask questions such as "Is frank the grandparent of dave?" (False), or "Is frank the grandparent of kaylen?" (True). However, the capabilities of logic programming extend beyond simple true/false questions. We can also ask questions such as "If X is the grandparent of Y, then who are X and Y?" (X=frank, Y= kaylen). In this case, X and Y are variables, which in Prolog are introduced simply by starting an identifier with an upper-case letter.

```
(defrel parent x y)
(facts parent '[[dave kaylen]
                [frank dave]])

;; Note that, by using defn, we are creating
;; a normal Clojure function.  It just happens
;; to work on logic variables as parameters.
(defn grandparent
    [x y]
    (fresh [z] ;; New variables must be explicitly introduced.
        (parent x z)
        (parent z y)))

;; ==========================================
;; Interactive prompt
user> (run* [q] ;; The output variable q is introduced
          (fresh [x y]
              (grandparent x y)
              (== q [x y])))

;; Result
([frank kaylen])
```

Figure 3.5: Core.logic code representing parent and grandparent relationships.

It is worth noting that there is an additional unbound variable, Z, which satisfies the substitution $Z = dave$. However, because the scope of Z is limited to the execution of the predicate, and is not returned as output, it is not shown. We will call variables such as Z, which are introduced in a limited scope, transient variables. In Prolog, transient variables are implicitly declared whenever they are used in a given scope.

Now, we will move on the Clojure core.logic example, shown in Figure 3.5. There are many similarities, but also some important differences, and these differences, although they may seem inconvenient, actually work to the advantage of the proposed system's design.

The first line declares the *parent* relation and its arity (number of parameters). For relations which are intended to store facts rather than act as predicates, the *defrel* function is used. After *defrel*, a number of facts are asserted into the parent relationship. A vector of vectors is used for this purpose (vectors are enclosed in square brackets, and are similar to lists). Each individual vector in the list is a tuple which represents a particular fact associated with a relation.

The grandparent predicate is defined using the *defn* special form, which results in the definition of a normal Clojure function. The fact that core.logic is embedded as a sub-language of Clojure and that predicates are represented as native Clojure functions means that they gain all the flexibility and dynamicity inherent in Clojure.

The function (or predicate, if you prefer) is defined as taking 2 parameters, x and y, similar to the Prolog function definition. However, unlike Prolog, the transient variable z must be explicitly introduced through use of the *fresh* macro. The *fresh* macro allows the introduction of any number of new, unbound variables, and creates a block in which they may be used.

The use of the interactive prompt also shows some differences in the way predicates are executed. Because the concept of multiple return values is not built in to Clojure like it is in Prolog, a different approach must be used. In this case, the *run** macro is used. This macro creates a block and introduces a context in which logical predicates may be executed. Instead of returning multiple possible variable bindings, a list of bindings is returned. For ease of representation in Clojure (and other Lisps), each element of the list represents a binding to a

single variable, which is declared a the beginning of the *run** block.

In this case, the primary variable is called q (as is often the convention in core.logic and MiniKanren). Other, transient variables must be introduced with *fresh*, and in order to get information on multiple variables returned, q must be unified with a composite structure of the other variables. In this case, the == operator is used to unify the variable q to a two-element vector containing x and y. It is worth noting that any q can be unified with any kind of structure, such as a set, hash-map, list, vector, etc.

Projection Because Prolog has the exclusive purpose of being a logic language, no distinction is made (at least from the average programmer's point of view) of logic variables and the data they represent. This allows programmers to use non-relational operators (i.e. normal functions) directly with logic variables.

```
% Interactive Prompt
?- X=1, Y= 2, Z is X + Y.
X = 1,
Y = 2,
Z = 3.
```

Figure 3.6: The arithmetic operators as well as the *is* operator are non-relational.

Figures 3.6 and 3.7 demonstrate the differences in the usage of non-relational operators in both Prolog and core.logic. The + operator is an example of a simple, non-relational function that is well understood. The Prolog example is very straightforward. First, X is unified with 1, Y is unified with 2, and the *is* operator is used to store the result of $X+Y$ in Z. The substitutions, as one might expect, give $Z = 3$.

The core.logic example is different for one very important reason: core.logic relational operators work with logic variables rather than native Clojure data. In this context, x and y do not directly represent the values 1 and 2, but rather represent other objects that store those values. In other words, there is a level of indirection that, while implicit in Prolog, must be dealt with explicitly in core.logic. The reason the initial execution fails in Figure 3.7 is that

```
;; Interactive prompt
user> (run* [q]
        (fresh [x y z]
               (== x 1)
               (== y 2)
               (== z (+ x y))
       ;; The final statement makes a hash map
       ;; Hash maps look like this: {k1 v1 k2 v2 k3 v3}
       ;; and create an associative map where each k is associated
       ;; with the corresponding v
               (== q {'x x 'y y 'z z})))
; Evaluation aborted.
;; Evaluation was aborted because an error occurred!!

;;
user> (run* [q]
        (fresh [x y z]
               (== x 1)
               (== y 2)
               (project [x y] ;; We have to look at the values
                              ;; inside x and y
                       (== z (+ x y))
                       (== q {'x x 'y y 'z z}))))
({x 1, y 2, z 3})
```

Figure 3.7: Use of non-relational operators in core.logic requires projection.

Clojure's native + operator expects to be working directly with numerical data, and is instead given a logical operator, causing a type mismatch error.

This is the purpose of the *project* macro. This macro takes any number of existing logical variables as parameters and "opens them up" to reveal the data they contain. More accurately, *project* introduces a new set of local variables that override the outer scope. The x and y inside the *project* block are not the same as those outside the block; the "projections" in the local scope share names with the variables they are projecting from the outer scope.

In the final execution in Figure 3.7, the *project* block results in the projection of x and y from logical variables to integers. However, q and z are not projected, and therefore can be referenced as logical variables. The unification operator (==) is used to unify z with 3 and unify q with a hash-map of the results.

Notation Because of the limitation of a single output variable, and the frequent necessity to introduce fresh variables, we developed a shorthand notation for defining predicates. It is shown in Figure 3.8.

3.1.6 Higher-Order Predicates

Disclaimer There is a slight overlap in terminology between the fields of functional programming and predicate logic. Please note that the type of logic system being used is still first-order logic. There do exist second-order and other logic systems, sometimes referred to as "higher-order", which are unrelated to the "higher-order" functions of functional programming. The term "higher-order predicate" here refers to the modelling of first-order logic predicates as higher-order functions in a functional programming context.

Using the capacity for creating first class and higher-order functions inherent in Clojure, and applying them to logic programming, it is possible to create higher-order predicates. Higher-order predicates are in fact just a special case of higher-order functions where the functions involved operate on logical variables.

```
;; The shorthand
;; Note that, as a matter of convention, when the return variable
;; isn't used directly, it is named _.  The underscore is a valid
;; variable name, just like any other.  It's just used by
;; convention for values we don't care about.
(predicate   _ ;; Return variable
             [x y z];; Fresh variables
             [z [y x]];; Return value, unified with q
             ;; Conditions...
             (== x 1)
             (== y 2)
             (== z 3))

;; Transforms into...
(fn [_]
    (fresh [x y z]
           (== x 1)
           (== y 2)
           (== z 3)
           (== _ [z [y x]])))

;; The shorthand :vec is used for vectors of the fresh vars
(predicate _ [x y z] :vec
             (== x 1) (== y 2) (== z 3))

;; Transformed...
(fn [_] (fresh [x y z]
             (== x 1) (== y 2) (== z 3)
             (== _ [x y z])))

;; The shorthand :map is used for maps of var names to values
(predicate _ [x y z] :map
             (== x 1) (== y 2) (== z 3))

;; Transformed...
(fn [_] (fresh [x y z]
             (== x 1) (== y 2) (== z 3)
             (== _ {'x x, 'y y, 'z z})))
```

Figure 3.8: Shorthand for predicate definitions.

To demonstrate their usage, let's return to the parent and grandparent relations from Figure 3.5. Notice that both *parent* and *grandparent* are two-parameter predicates that represent some kind of ancestry relationship. Therefore, in many cases, they might be used in the same context. As an example, consider the source listing in Figure 3.9.

The source in Figure 3.9 shows how multiple predicates can be used and passed to other predicates. The first lines define a few relations that will be used to store facts. We have already seen the *parent* relation. The relation *afunc* is used to store predicates. The relation *funcname* is used to associate functions with names. This serves no purpose other than to make the output clearer to the user.

Moving on to the predicate definitions, we see the grandparent relation, just as is it was in the previous example. There is also a four-parameter predicate called *ancrel* (standing for *ancestor relation*). This is an example of a higher order predicate. It takes as its parameters four logic variables: p represents a predicate, x and y represent parameters to the predicate, and q represents the outermost output variables (typically, this will be declared by the *run** block).

First, the relation *funcname* is used to obtain the predicate's name and store it in the variable n. Then, p is projected. The projection is necessary because, in order for the Clojure runtime system to execute a function, it must be in its native form; logic variables cannot be cast to functions. Inside the project block, p is applied to arguments x and y, and q is unified with a hash-map of results, including the function name for the sake of output clarity. We can see the result of using the higher-order predicate in the interactive prompt; all possible types of ancestry (*parent* and *grandparent*) and all pairs that satisfy them are returned.

3.1.7 Information Hiding with pldb

In order to properly construct a world model in which agents had individual minds, some capacity for information hiding is required. Each agent requires its own personal database of facts as well as access to the global database. This capability is not inherent in core.logic's default implementation, but was provided by the pldb system, an open-source extension of

```
;; Relation for parents
(defrel parent x y)
;; Relation for listing functions
(defrel afunc f)
;; Relation for associating functions with names
(defrel funcname f n)

;; Grandparent relation
(defn grandparent [x y]
  (fresh [z]
      (parent x z)
      (parent z y)))

;; p : the predicate
;; x : the ancestor
;; y : the descendant
;; q : output variables
(defn ancrel [p x y q]
    (fresh [n]
       (funcname p n)
       (project [p]
                (p x y)
                (== q {'func n 'x x 'y y}))))

;; Declaring facts
(facts afunc
      [[grandparent]
       [parent]])
(facts funcname
      [[grandparent 'grandparent]
       [parent 'parent]])
(facts parent
     '[[dave kaylen]
       [frank dave]])

;; Interactive prompt
user> (run* [q]
          (fresh [p x y]
               (afunc p)
               (ancrel p x y q)))

({func parent, x dave, y kaylen}
{func parent, x frank, y dave}
{func grandparent, x frank, y kaylen})
```

Figure 3.9: Example of higher-order predicates using the parent and grandparent relations.

core.logic [18].

Using pldb, it is possible to define database relations in addition to normal relations. With database relations, facts are asserted in a particular database context, and therefore affect only local changes. Using a unique database context for each agent allows for a different set of facts, and therefore different (potentially conflicting) knowledge for each agent.

3.1.8 Explicit Hierarchies and Multimethods

Most current object-oriented programming languages provide some sort of runtime polymorphism. Typically, this involves a set of functions with the same name or type signature defined in several classes. These classes are in a hierarchy, with sub-classes descending from super-classes. If a derived class inherits from a base class, the base class's version of the function will be executed. If the base class does not explicitly have its own implementation of the function, its most recent ancestor's implementation will be executed. This is all very well known and commonly used among object-oriented programmers.

Multimethods, Clojure's version of polymorphic functions, use a much less common technique, which can replicate the above-described behaviour and can extend it in arbitrary ways. With typical object-orientation, the type (or class) of the first parameter of a method (the caller object, typically called *this*) is the only data that is taken into account during dynamic dispatch. Clojure's multimethod system allows the programmer to provide their own dispatch function, therefore allowing arbitrary means of hierarchy-based dynamic dispatch.

Additionally, a set of hierarchies can be explicitly defined by the user in addition to, and in extension of, the existing class hierarchy. The objects in the hierarchy need not be classes or data types, but can simply be symbols or keywords. Also, it is possible to write code in core.logic that allows relational queries to be executed on such hierarchies.

Figure 3.10 shows an example of a simple user-defined hierarchy and its use in defining a simple polymorphic multimethod. The *derive* function is used to create a hierarchy, which is illustrated as a graph in Figure 3.11. Additionally, a function *get-type-data* is defined. This

```
;; Gets type data from an object
(defn get-type-data [o]
   (get o :type))

;; the type ::foo is at the top of the hierarchy
(derive ::bar ::foo) ;; Bar descends from foo
(derive ::baz ::bar) ;; Baz descends from bar
(derive ::qux ::foo) ;; Qux descends from foo

;; To print info, we dispatch on get-type-data
(defmulti print-info
          get-type-data)

;; Define the actual methods with dispatch values
(defmethod print-foo ::foo [o]
          (prinltn "We got a foo!"))

(defmethod print-foo ::bar [o]
          (prinltn "We got a bar!"))

;; Interactive prompt
;; ==========================
user> (print-foo {:type ::foo})
We got a foo!
user> (print-foo {:type ::bar})
We got a bar!
user> (print-foo {:type ::baz}) ;; Most recent ancestor is bar
We got a bar!
user> (print-foo {:type ::qux}) ;; Most recent ancestor is foo
We got a foo!
```

Figure 3.10: An example of multimethod polymorphism in Clojure. The hierarchies do not necessarily have to represent types. They can be anything!

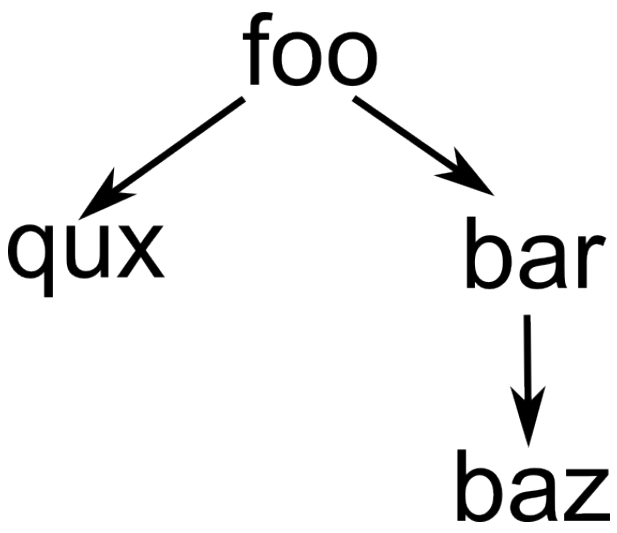

Figure 3.11: An illustration of an arbitrary type hierarchy.

function assumes that the data provided is an associative map, and looks up the data at key *:type*.

A multimethod is defined with *get-type-data* as its dispatch function. Two print functions are defined for dispatch values *::foo* and *::bar*. In the interactive prompt, we can see the result of the execution of the multimethods. First, the data is passed to the dispatch function, which looks up the object at key *:type*. Depending on that key, a method is selected. If a method for the exact dispatch value cannot be found, it will default to that values most recent ancestor (i.e. calling the method with a dispatch value of *::baz* will result in the *::bar* method being executed).

3.1.9 A Note on Metadata and Type Annotations

It is worth noting that all Clojure objects (with a few exceptions) allow for the attachment of metadata. Metadata does not affect the value of an object, or the ways that object may be used, but allows the object to carry with itself extra information. Use of metadata is entirely optional, but can be used in any way the user sees fit. Our system will primarily use metadata for the attachment of type annotations (i.e. adding the key *:type* to the metadata map rather than the object itself).

3.2 Linguistics

3.2.1 Adjacency Pairs

Adjacency pairs are a concept in pragmatics, a branch of linguistics. Simply put, adjacency pairs refer to pairs of utterances by two speakers which correspond in some way [15].

A more practical description can be provided by a few examples:

- "Hello" → "Hi"
- "How are you?" → "I'm fine. How are you?"
- "What time is it?" → "Five-thirty."

Pragmaticists and conversation analysts use the concept of adjacency pairs to characterize much of human conversation; dialogue often proceeds in $\{statment \rightarrow response\}$ pairs. More accurately, the two parts are referred to as the First Pair Part (FPP) and Second Pair Part (SPP). Interestingly, these pairs can be said to be typed; that is, although the specific SPP's may differ in response to a particular FPP, the types of those SPP's remain the same.

The types of the previously shown adjacency pairs may be described as follows:

- $Greeting \rightarrow Greeting$
- $Inquiry < Status > \rightarrow Status$
- $Inquiry < Time > \rightarrow Time$

For instance, when asked "What time is it?", "5:30" , "Noon", "Midnight", and "I'm not sure" all constitute felicitous responses. A response such as "No thank you", on the other hand, is not felicitous. A question about the time of day expects a response about the time of day.

The fact that adjacency pairs have associated data types means that they can be easily modeled with programming languages that deal with data types and type hierarchies. The use of types and multimethod polymorphism in implementing adjacency pairs is more fully described in Section 4.3.

Chapter 4

Design and Implementation

In order to discover the requirements necessary to implement a first-order logic-based dialogue system, a number of concepts and example scenarios were considered. In attempting the implementation of these, the requirements and design of the system revealed itself.

4.1 Overview and System Architecture

The system described in this Chapter is intended to be used as a basis for other research and experimentation, and is therefore designed to be modular and extensible. An overview of the components is provided here, and each of the components will be described in more detail throughout the chapter.

Figure 4.1 shows the major components of the system. There are two main areas of data, Facts and Relations and Behaviour Data. Facts and relations contains the database contexts of each agent, as well as relations supporting the access and definition of that data. The behaviour data, on the other hand, represents behavioural data. This data consists primarily of predicates and other first-class functions. Recall that the functions used in this system are *first-class* objects, and therefore can be represented as data. Additionally, such functions need not be hard-coded into the system, but can be loaded and interpreted at run-time.

Any agent attributes, such as personality data, emotional state, or social relationships be-

tween agents, are provided by a combination of facts, relations, and behaviour data. State can be represented with by facts in the agent database, or by some attribute in an external system, and the way this state is interpreted and used to affect agent behaviour arises from the specifics of the thought, response, and rule functions.

The two data modules are tied together by their interaction with the core.logic system. If any external system exists (and is implemented on the JVM), accessor predicates can be defined to facilitate interaction between the logic system and the external system. Additionally, although a theory-of-mind module is not explicitly defined, it can be achieved by storing references to the behaviour data of other agents in an agent's database context.

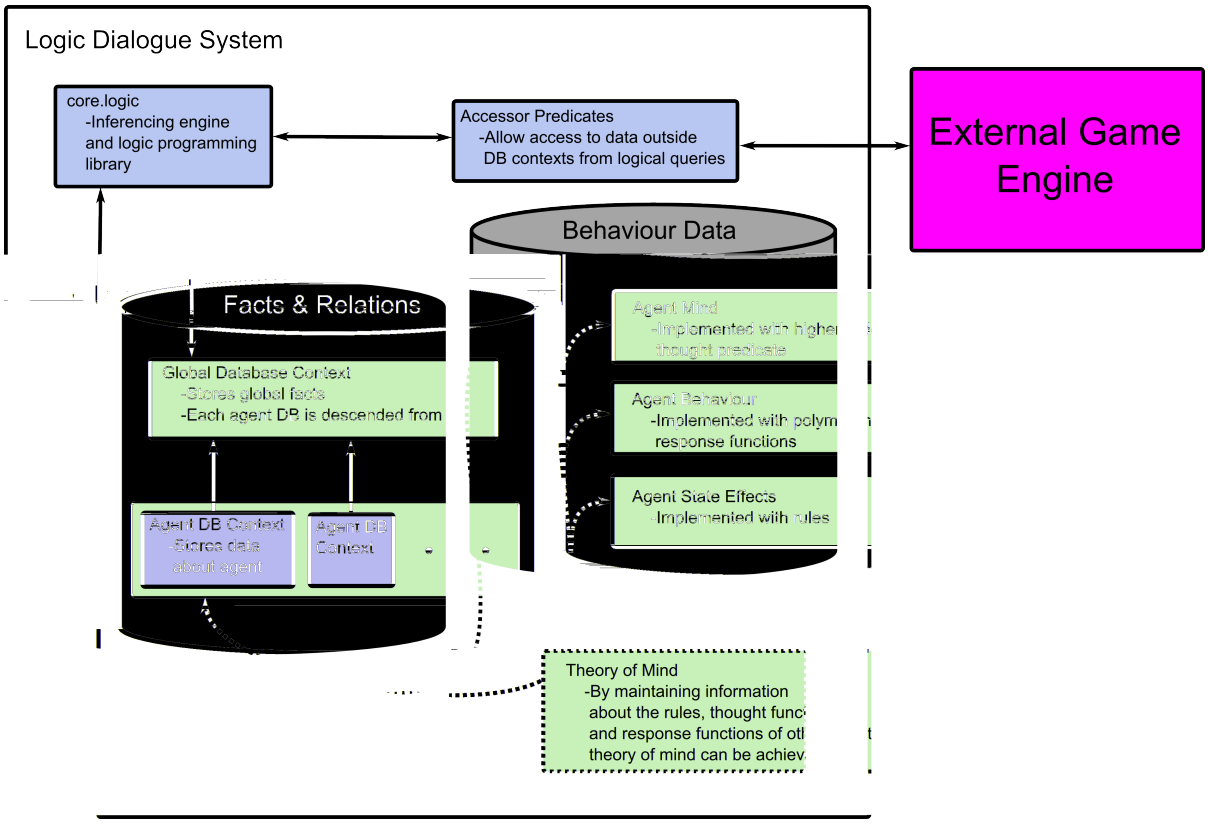

Figure 4.1: System architecture.

4.2 Design From Example: Knights and Knaves

The so-called "Knights and Knaves" puzzle, and in particular its solution, offers an ideal simple test scenario for both dialogue construction, agent mind, and theory of mind implementation. The puzzle has numerous variations, but often involves a fictional island. On this island, inhabitants are either knights or knaves. Knights always tell the truth and knaves always lie.

One variation involves two men standing by a fork in the road. One of the paths leads to freedom and the other leads to death. It is known that one of these men is a knight and the other is a knave, but not which is which. (The men themselves know which of them is which, but not the person being asked the riddle.) The puzzle involves determining which path leads to freedom by asking exactly one question to exactly one of the men.

Directly asking which path leads to freedom yields no new information; the knight will suggest the correct path and the knave will suggest the incorrect path. Also, due to the limitation of asking exactly one question, one cannot ask another to which the answer is already known to determine which one is the liar. It would not be possible, for instance, to ask "Does 2+2 = 4?" to determine who is the liar.

The solution, therefore, is to find a question to which both men would give the same answer. (If you wish to solve the puzzle yourself, be warned that the next sentence reveals the answer!) The appropriate question is "If I were to ask the other man which path led to freedom, what would he say?", and then take the other path. The knight would anticipate that the knave would lie, and therefore would answer truthfully about a lie. The knave would lie about the knight telling the truth. In either case, the answer given would end up being the path that lead to death, so the solution is to do the opposite of the answer.

In fact, this solution is in a way a variation of the two-question solution (e.g. asking "what is 2+2?" first). However, there is an important difference. By using higher-order questioning, and embedding a question within another question, we're able to achieve the effect of asking two questions by only asking one.

It is because of this higher-order capacity for questioning that this example was selected as

a basis for design; if a system could be devised to appropriately model this scenario, it would then have a capacity for higher-order dialogue.

Another very important property of this solution is that it involves a very simple theory of mind component; each agent must not only know how they answer questions, but must have their own mental model about how the other agent would answer a question.

4.2.1 Implementing Knights and Knaves

At first glance, the implementation of a knights and knaves-like scenario in first order logic may seem straightforward; after all, the answers of the knights and knaves involves simple data lookup and some simple postprocessing – either keeping the answer as it is or giving the opposite answer. However, as we will see, more is involved if one wants to incorportate higher-order questions and theory of mind.

Figures 4.2 and 4.3 show an initial, "naïve" framework for representing the knights and knaves scenario.

A few important points need explanation here. First of all, a set of relations are defined. The *paths* relation stores data about which path leads where, *opposite* sets up pairs of opposite objects to enable the knave agent to lie (it has to know which answer is the wrong one), and finally *mindf*. The *mindf* relation associates a predicate with an agent. In this case, the relations represented are simply *knightanswer* or *knaveanswer*, which are simple predicates, wrappers for either direct unification (truth) or for finding the opposite (lie).

Although this design does work in a sense, it does not provide the appropriate information hiding that one might expect in this scenario. Additionally, there's no way of enforcing the single-question constraint, and no way of sending a single, composite question to an agent and getting back a single answer.

The final query executed in Figure 4.3 could be transcribed something like the following:

User: "Jack, which path leads to freedom?"

Jack: "Right"

```
;; Relation to determine where paths lead
(defrel paths x y)

;; The person's mind predicate
;; The f stands for function, and nothing else
(defrel mind-f x p)

;; Relation defining opposites, so that the knaves
;; know how to lie about their answers
(defrel opposite x y)
(facts opposite
   '[[Left Right]
     [Right Left]
     [Death Freedom]
     [Freedom Death]
     [Jack Bill]
     [Bill Jack]])

;; Predicates to represent how knights and knaves answer
;; questions differently.

(defn knightanswer [a b]
   (== a b))

(defn knaveanswer [a b]
   (opposite a b))

;; Interactive prompt
;; ====================
;; Initializing the paths
;; Left leads to freedom and right leads to death
user> (facts paths
             '[[Left Freedom]
               [Right Death]])

;; We will say that Bill is the knight
;; And Jack is the knave
user> (facts mind-f
             [['Bill knightanswer]
              ['Jack knaveanswer]])
```

Figure 4.2: Knights and Knaves function definitions.

```
;; Asking a simple question
;; We will say we're asking it to Jack,
;; and pretend we don't know the answers
user> (run* [q]
          (fresh [x p]
              (paths x 'Freedom)
              (mind-f 'Jack p)
              (project [p]
                       (p x q))))
;; Result
(Right)

;; Asking the solution,
;; "If I asked the other guy, what would he say?"
;; Doesn't really work with this system
user> (run* [q]
        (fresh [x y p p2]
               (paths x 'Freedom)
               (mind-f 'Bill p)
               (mind-f 'Jack p2)
               (project [p p2]
                        (p2 x y)
                        (p y q))))
;; Result
;; Note that interchanging Jack and Bill will
;; Yield the same result
(Right)
```

Figure 4.3: Knights and Knaves

User: “Bill, what did he just say?”

Bill: “Right”

In order to improve this system, we need two important components: information hiding and the capacity for higher-order questioning. This will allow the user to use a single composite query, directed at either agent, and receive the same answer.

Starting with implementing information hiding, we can convert some relations to database relations. The *mindf* function should be converted, because only the agents should have access to information about the other agents’ minds. The *paths* relation will also be converted, so that the user does not have access to that information. In more complex systems, this could lead to implementation of more complex theory-of-mind. Also, a new database relation called *other* is introduced. This relation is a single-parameter predicate, and will yield the name of the other agent when called in a particular agent context. (i.e. in Jack’s context, calling *(other x)* will result in the substitution $x = Bill$ and vice-versa).

In order to properly combine the features of higher-order questioning, information hiding, and thought functions, thought functions are directly modeled as higher-order predicates.

Thought Functions as Higher-Order Predicates

There are a number of ways to address agent minds, and a number of believable agent systems take different approaches [8–10, 14]. However, for the purposes of this dialogue system, the agent mind is conceptualized as a speech filter; it takes some input query and executes it, as well as pre- and post-processing the results of that query. The specifics of the pre- and post-processing determine the behaviour of the agent’s mind. (For instance, running a query, and then finding the opposite of the result, is the knave’s thought function.)

In order for this to work, utterances should be characterized as functions. More specifically, in this example, they are all characterized as single-parameter predicates. Single-parameter

predicates have the advantage that they can be executed within a *run** block and have their results returned (see Section 3.1.5). Therefore, the sender of the predicate does not need to have any information about facts or database context; the sender simply sends an utterance (function), which the receiver can execute in its own context, and provide a return value.

The extra code required for this implementation is shown in Figures 4.4 and 4.5. Note that, for purposes of clarity and succinctness, some functions are described with comments rather than written in their entirety.

```
(def *databases*) ;; A map of agent names to agent databases
(def *minds*) ;; A map of agent names to agent mind functions

;; Allows the introduction of a block in which a single agent's
;; database context is used.
(defmacro with-agent);;No source

;; Allows modification of an agent's database context
(defmacro mod-agent);;No source

;; This is a helper function that allows wrapping of a higher-order
;; predicate in a run* block.
(defn make-mind
    [agent f]
    (fn [p]
        (with-agent agent
            (run* [q]
                (f p q)))))

;; Receives a symbol representing an agent as well as a predicate
;; It looks up the agent's mind function and passes the predicate
;; to it, returning the result
(defn ask-agent
    [agent p]
    ((get @*minds* agent) ;; Lookup the mind function
       p)) ;; Apply it to the predicate
```

Figure 4.4: Newly implemented core functionality to handle higher-order predicates.

Figures 4.4, 4.5, and 4.6 represent three sections of the code that makes the scenario work properly. First of all, Figure 4.4 shows some functionality that must be embedded in the core system in order to enable this scenario to be authored.

```
;; Mind-function database relation
(db-rel mind-f x f)

;; Database relation which yields the other person
(db-rel other x)

;; Paths is converted to a database relation so
;; the user can't ask about it
(db-rel paths x y)

;; Here are the mind predicates, truth and lie

;; This one takes a predicate p and variable q and
;; applies p to q
(defn truth
  [p q]
  (p q))

;; Takes a predicate p and a variable q, and returns
;; the opposite of applying p to q
(defn lie
  [p q]
  (fresh [x]
         (p x)
         (opposite x q)))


;; This randomly initializes which path leads where and
;; which agent is a knight/knave
(defn rand-init);; No source
```

Figure 4.5: Newly implemented functionality more specific to the Knights and Knaves scenario.

```
;; Interactive Prompt

user> (rand-init) ;; Initialize it randomly, so we don't know!
nil

;; Let's try asking the database directly about
;; which path leads where!
user> (run* [q]
         (paths q 'Freedom))
;; What's this?  Nothing is returned!?
()

;; Let's try embedding our question in a function, and sending it
;; to an agent
user> (ask-agent 'Jack
         (fn [x] (paths x 'Freedom)))
;; Jack tells us one thing
(Right)

user> (ask-agent 'Bill
         (fn [x] (paths x 'Freedom)))
;; Bill tells us another
(Left)

;; Here's the solution that should be asked!!
user> (def solution
          (fn [q]
               (fresh [o p]
                      (other o)
                      (mind-f o p)
                      (project [p]
                               (p (fn [x] (paths x 'Freedom)) q)))))

user> (ask-agent 'Bill solution)
(Right)

user> (ask-agent 'Jack solution)
(Right)
```

Figure 4.6: The effect of the new code on the interactive prompt.

At the beginning, it provides some globally defined collections of database and mind-functions relating to agents, **databases** and **minds**. The *with-agent* and *mod-agent* macros enable the introduction of code blocks for working with a particular agent. Any code written within one of those blocks will use by default the database context of their given agent. The *mod-agent* macro is specifically for modifications to the databases (i.e. asserting new facts).

The *make-mind* function is a helper function, but it is important. It takes a higher order predicates and "lifts" it, in a sense, wrapping it in its own *run** block and its own *with-agent* block. This allows the agent mind function to be used outside of a logic programming context. It's also worth noting that *make-mind* does not need to be used to create an agent mind function. This allows for user-specified preprocessing of input or post-processing of output when the predicate is executed.

Finally, the *ask-agent* function provides the primary interface for dialogue in this scenario. This function takes 2 arguments: an agent's name and a predicate. It looks up the agent's mind function, and applies it to the predicate, resulting in a list of valid substitutions for the single parameter. Examples of this are shown in Figure 4.6.

Figure 4.5 shows code more specific to the knights and knaves scenario. It provides the definition of database relations: *mind-f* for the mind-functions of each agent, *other* for the other agent, and *paths* to store which path leads where. These are specifically defined as database relations so that only the agents will have access to them, and in some cases may have different values (for instance, when executing *(other x)*, $x = Bill$ or $x = Jack$ depending on who you ask).

The mind functions *truth* and *lie* are also shown, and these are simple examples of agent minds being modeled as higher-order predicates. Each of them takes a predicate and a logic variable and applies some logical operations to them. The *truth* function simply applies the predicates to the variable, yielding an accurate substitution. The *lie* function, on the other hand, applies the predicate to a transient variable x, and unifies the opposite of x with q, effectively yielding a false substitution for q.

It may be simpler to understand the use of these functions if used interactively. Figure 4.6 shows an interactive session annotated with comments. First, *rand-init* is called to randomly initialize the paths and the agents; we do not know who lies or tells the truth, or which path leads where.

The second line shows an attempt at a direct query to the database to determine which path leads to freedom. However, no results are returned. This is because path information is no longer stored in the default database, but requires an agent's database context to make sense.

The solution to this is to construct a function in which the question is asked. The next lines after show the user asking Jack and Bill the same question: "Which path leads to freedom?". More accurately, it could be rendered something like this "I'm giving you a function *f(x)*, such that there exists some path from x to Freedom. Apply f within your own database context and give me back all valid substitutions for x.".

As can be clearly seen, their answers differ. This is to be expected, as one is the liar and one is the truth-teller. Recall that the solution to the problem is "If I were to ask the other person, what would he say?". The predicate equivalent of that question is stored in the variable called *solution*. In order to properly understand it, it should be broken down into its components. First of all, the whole thing is a single-parameter function of an argument q, which is a logic variable. Fresh variables called o and p are introduced. The other person(of whoever is being asked) is o, and the mind function of o is p. Next, the mind function p is applied to the question "Which path leads to freedom?". So, rendering this as English might come out something like this: "Here is a function *f(q)*. Let o be the other person and let p be the mind function of o. Apply p to the question 'Which path leads to freedom?' and to q, and return all valid substitutions of q.".

The use of such higher-order predicates allows arbitrary levels of question-nesting, where each question can be expressed in the uniform format of a single-parameter predicate. However, as will be seen in the next section, modifications to this system are required if dialogue is to be more fully represented.

4.3 Adjacency Pairs and Polymorphism

Adjacency Pairs (described in Section 3.2.1) are a construct used in linguistics to describe the way in which conversations proceed. An important aspect of these pairs is that utterances can be said to be "typed". That is, there is a data type associated with each utterance that restricts the ways in which that utterance may be responded to.

When modeling adjacency pair-based dialogue using any programming language, it is useful to use the data type and polymorphism semantics inherent in the language. Clojure was chosen for its ability to represent arbitrary, user-defined type hierarchies as well as its flexible polymorphism capabilities. For more details, see Section 3.1.8.

Additionally, since messages may take a variety of different forms, a uniform data structure was not used to represent messages. Rather, messages can take the form of any object, ranging from simple symbols to functions to complex nested data structures. All that is necessary for a message to be used by the system is that it is annotated with the appropriate metadata.

In order to handle adjacency pair-like dialogue, agents in the system are given polymorphic *response functions*, which work in addition to their thought functions.

4.3.1 Response Functions

Whereas thought functions handle the processing of predicates in the agent's database context, response functions handle the construction and sending of messages after the though process is complete. They have access to both data about the message received and the results of the agents thought function, if it was executed.

The implementation of polymorphic response functions as they relate to adjacency pairs is perhaps better illustrated with an example. Figure 4.7 shows some example adjacency pairs and their associated type-signatures. Figure 4.8 shows an example of response function execution using the adjacency pairs as data.

- "Heya" → "Oh hi."
  greeting → greeting
- "Would you like to visit the museum with me this evening?" → "I'd love to"
  offer → acceptance/rejection
- "Would you like to visit the museum with me this evening?" → "No way, jerkface! Museums are for nerds!"
  offer → acceptance/rejection
- "Your phone is over there" → "I know"
  inform → acknowledge

Figure 4.7: Some example adjacency pairs from Wikipedia [24], with some modifications.

4.3.2 Message Types

A hierarchy of message types as well as a means of annotating them with appropriate type data is necessary to take advantage of multimethod polymorphism. Although the message type hierarchy is an open and extensible set for users authoring scenarios, a set of default message types is available that have special meanings within the functioning of the system. They are described below.

Message Generic supertype of all other messages. A response function that handles a dispatch value of Message will accept any message type.

Marker The marker type does not represent a speech act, but is rather a control mechanism. It consists of two symbols: Start and Stop. These symbols are used in the message queue as an invitation for starting or stopping a conversation. The system could be initialized with a Start message in the message queue directed at a particular agent, indicating that that agent should speak first. An agent would "respond" to this message by initiating a conversation.

Predicate Predicate is a special message type; annotating a message with this type does not only affect the selection of an agent's response function, but also affects the operation of the message processing system.

```
(t-def heya ["Heya"] ::Greeting)
(t-def oh-hi ["Oh hi"] ::Greeting)
(t-def visit-museum ["Would you like to visit the museum with me this evening?"]
       ::Offer)
(t-def love-to ["I'd love to"] ::Acc-Rej)
(t-def jerkface ["No way, jerkface!  Museums are for nerds!"] ::Acc-Rej)
(t-def phone-there ["Your phone is over there!"] ::Inform)
(t-def i-know ["I know"] ::Knowledge)

(def agent-mood (atom nil))

(defmulti respond
  (fn [o]
    (get (meta o) :type)))

(defmethod respond ::Greeting
  [msg]
  oh-hi)

(defmethod respond ::Offer
  [msg]
  (cond
   (= @agent-mood :good) love-to
   (= @agent-mood :bad) jerkface))

(defmethod respond ::Inform
  [msg]
  i-know)

(defn initialize
  [mood]
  (reset! agent-mood mood))
```

Figure 4.8: Example response function execution.

Marking an object as a predicate signals the system to process it using the agent's thought function before sending it to the response function.

At first glance, because of predicates' natural association with queries, it may seem that predicates can only be used as interrogative statements. However, it is possible to use them to generate side-effects in an agent's database by having them assert facts. More details on how this is possible are covered in Section4.4.

Response The response message type denotes a response to a predicate. More technically, it is usually a list of substitutions for the output variable of a predicate. The response forms the second part of a generic question-answer adjacency pair that is initiated by the predicate.

When given a Response, an agent will typically update its own knowledge base with the information presented within the response, and subsequently steer the conversation in whatever direction it sees fit.

Affirmation Affirmations are simple messages such as "yes" or "no", which indicate that an agent has accepted the result of a message. Typically, these are used when a predicate generates only side-effects or unbound variables.

For example, the following query equates to something like "Is your mother Anna?".

```
(fn [q]
    (fresh [n]
        (*agent-name* n)
        (mother n 'Anna)))
```

This query may succeed or fail, but if it succeeds it still leaves the primary output variable q unbound. The fact that the result is composed entirely of unbound variables means that it could be responded to with an affirmation (i.e. "Yes, Anna is my mother." rather than "q could be anything.").

4.4 Modifiers

An implementation difficulty arose when attempting to introduce assertions of facts into the system. Because the database contexts of individual agents are immutable data structures, they cannot be changed while queries are being executed. Rather than changing the database context itself, agents are provided with mutable references to immutable structures. These references can be changed to point to new database contexts, representing updated versions of the original contexts.

In order to solve the issue of the introduction of side-effects, a set of special structures with particular metadata was introduced which could be generated as the result of a query. If any such data structure were found in the results of a query, they would be filtered out and used to update the database context.

Collectively, these structures are referred to as Modifiers. Although each of them is different in the specific structure used to represent them, they all carry metadata that identifies them as Modifiers, as well as containing a set of relations which they modify. The *type* field of the metadata contains one of the types described below. Code examples for the creation of these modifiers are shown in Figure 4.9.

Assertion The assertion is perhaps one of the most basic types. It consists of a list, the first element of which is a relation, and the rest of which are the fields of that relation. When executed, it enters a new fact into the database corresponding to the tuple provided.

Retraction A retraction is identical in structure to an assertion, but is annotated with different type metadata. Its execution results in the retraction of the tuple provided.

Change Changing data in a logic programming system is difficult because of the nature of the data representation. One does not simply change the value of a variable. In a logic programming system, both values can exist simultaneously as valid substitutions for a variable.

The change, therefore, is much more complicated than either assertions or retractions. It consists of a predicate paired with an update function. The predicate, when executed, returns a map, which is passed into the update function, which in turn returns another map.

The maps mentioned above have relations as their keys and argument tuples as their values. The initial maps generated by the predicate are used for retraction, then passed to the update function. The update function then modifies the tuples and returns a new map, which is used for assertion.

The net effect of this is that arbitrary modifications of existing tuples can take place by retracting data, modifying it, and then re-asserting it.

4.5 Rules

The final component, in addition to higher-order predicates and polymorphic message processing, is the addition of rules to the system. Rules provide a means of allowing agents to change their internal states in response to certain conditions.

A rule is described as a combination of a change modifier an a trigger. The modifier describes how the agent's state is affected by the condition, and the trigger provides a means for other agents to activate the rule. More specifically, it describes a message format which other agents can use to generate a message that will trigger the rule.

Figure 4.10 shows an example of how a rule might be defined. This particular rule is from a simple test scenario. In the test scenario, one agent (Jack) is attempting to make another agent (Bill) feel bad. Part of the state of both agents involves their emotion-level, which is initialized to zero. In order to make Bill feel bad, Jack must attempt to minimize his emotion level, and he has a number of options for doing so.

First, let's review the code in Figure 4.10. First, some relations are defined and facts are placed into a default database context. The default context contains information about which

```
;; Here are some relations
(defrel changefoo tf)
(defrel foo x y)

(def foo-change
    (ch-pred [foo]
        _ [x y] ;; Special notation to define fresh vars x and y
        {foo [x y]} ;; Return a map from foo to [x y]
        ( (changefoo true) ;; Changefoo needs to be true
          (foo x y) ) ;; And foo x y needs to exist
        (fn [{[x y] foo}]
            {foo [(+ x 1) (+ y 1)]})))
;; Interactive prompt
> (run* [q]
    (fresh [x y]
       (foo x y)
       (== q [x y])))
() ;; No results!

> (run* [q]
    (== q (ast foo 1 2)))
((foo 1 2)) ;; Here's a tuple

> (map exec-mod *1) ;; Run exec-mod on all the results
nil                     ;; Note: *1 is shorthand for "previous result"
> (run* [q]
    (fresh [x y]
       (foo x y)
       (== q [x y])))
([1 2]) ;; We have executed the assertion!
> (exec-mod foochange)
nil
> (run*... ) ;; Same query as above
([1 2]) ;; Didn't work, but why?

> (run* [q] (== q (ast changefoo true)))
((changefoo true)) ;; Another assertion

> (map exec-mod *1) ;; Execute the changes
(nil)

> (exec-mod foochange)
nil
> (run* ...) ;; Same query as above
([2 3]) ;; It has been successfully incremented.
```

Figure 4.9: Code examples for creating modifiers.

```
;;;; Relations
(db-rel emotion-level f)
(db-rel mother x y)
(db-rel is-pretty n)
(db-rel is-nice n)
(db-rel is-fat n)
(db-rel is-ugly n)
(db-rel nice-pred)
(db-rel mean-pred)

;;;; Default Database - modified with new facts
(def default-db
  (db-facts empty-db [nice-pred is-pretty] [nice-pred is-nice]
                     [mean-pred is-fat] [mean-pred is-ugly]))

(def mother-insult-rule
    "This rule makes you feel bad if your mother is insulted."
  (make-rule
   ;; The change modifier
   (ch-pred [emotion-level]
        ;; Fresh variables
        _ [p m n f]
        ;; Return value
        {emotion-level [f]}
        ;; Rule Condition
        ( (relso p)        ;; p has been modified this update
          (mean-pred p)    ;; p is a mean predicate
          (*agent-name* n) ;; the current agent is n
          (mother n m)     ;; n's mother is m
          (emotion-level f);; f is the emotion level
          (project [p]
                   (p m)) );; p is true of m
        ;; Rule side-effect
        (fn [ m ]
          (let [[f] (get m emotion-level)]
             ;; Return an updated map
             {emotion-level [(- f 0.5)]})))

   ;; Rule message trigger
   ;; If this rule is selected for use, the other agent needs
   ;; a way to trigger it.  This predicate will be used.
   (predicate _ [p m n] (ast p m)
            (*agent-name* n)
            (mother n m)
            (mean-pred p))))
```

Figure 4.10: An example of a rule, combining a change predicate and a trigger.

predicates are classified as "nice", and which are classified as "mean". Clearly, *is-pretty* and *is-nice* are nice, and *is-fat* and *is-ugly* are mean.

The mother insult rule is defined with some conditions. First of all, *(relso p)* indicates that the relation p has been modified this update cycle. This is in order to prevent the rule from being triggered over and over again. Next, *p* must be a "mean" predicate. Then, *n* is the name of the agent currently executing the rule (in this case it will be Bill), and *m* is the mother of *n*. If all of these conditions are satisfied, then *f* is unified with the current emotion level, and the final condition checks to see if the predicate *p* is true of *m*.

Only if all of these conditions hold true is it possible for the rule condition to yield a return value. In this case, the return value is a map, which maps the *emotion-level* relation to the current 1-tuple ([f]) representing its numeric value. When this map is returned, it is used for retraction. Each relation, and the tuple associated with it, are removed from the database. For instance, if the value {*emotion-level [0.0]*} is obtained, then the fact *(emotion-level 0.0)* is retracted from the database.

After this retraction, the value {*emotion-level [0.0]*} is sent to the rule's side-effect function, which subtracts 0.5 from the numeric value, returning the value {*emotion-level [-0.5]*}, which then results in the assertion of the fact *(emotion-level -0.5)*.

By examining the side effects of Bill's rules, Jack can choose an optimal action to make him feel bad. For instance, in this case, Jack might initially obtain information on Bill's emotion level. Once that information is obtained, he can pass it to the rule side effect function to examine how it is changed. Running the function will reveal that the level is decreased by 0.5. Therefore, since this is an adequate result, Jack will use the trigger to send a message to Bill. The trigger, in this case, is a simple predicate asserting that some mean predicate *p* is true of Bill's mother.

4.6 Combining the Components

After the three primary components – higher-order predicates (mind-functions), polymorphic message processors (response functions), and rules were developed, they were combined into a simple game-like system.

In the current system, discrete update-cycles (or steps) are used to advance the progress of the simulation. In each step, an agent receives a message, processes it, and responds to it. A diagram showing the operation of the system is shown in Figure 4.11.

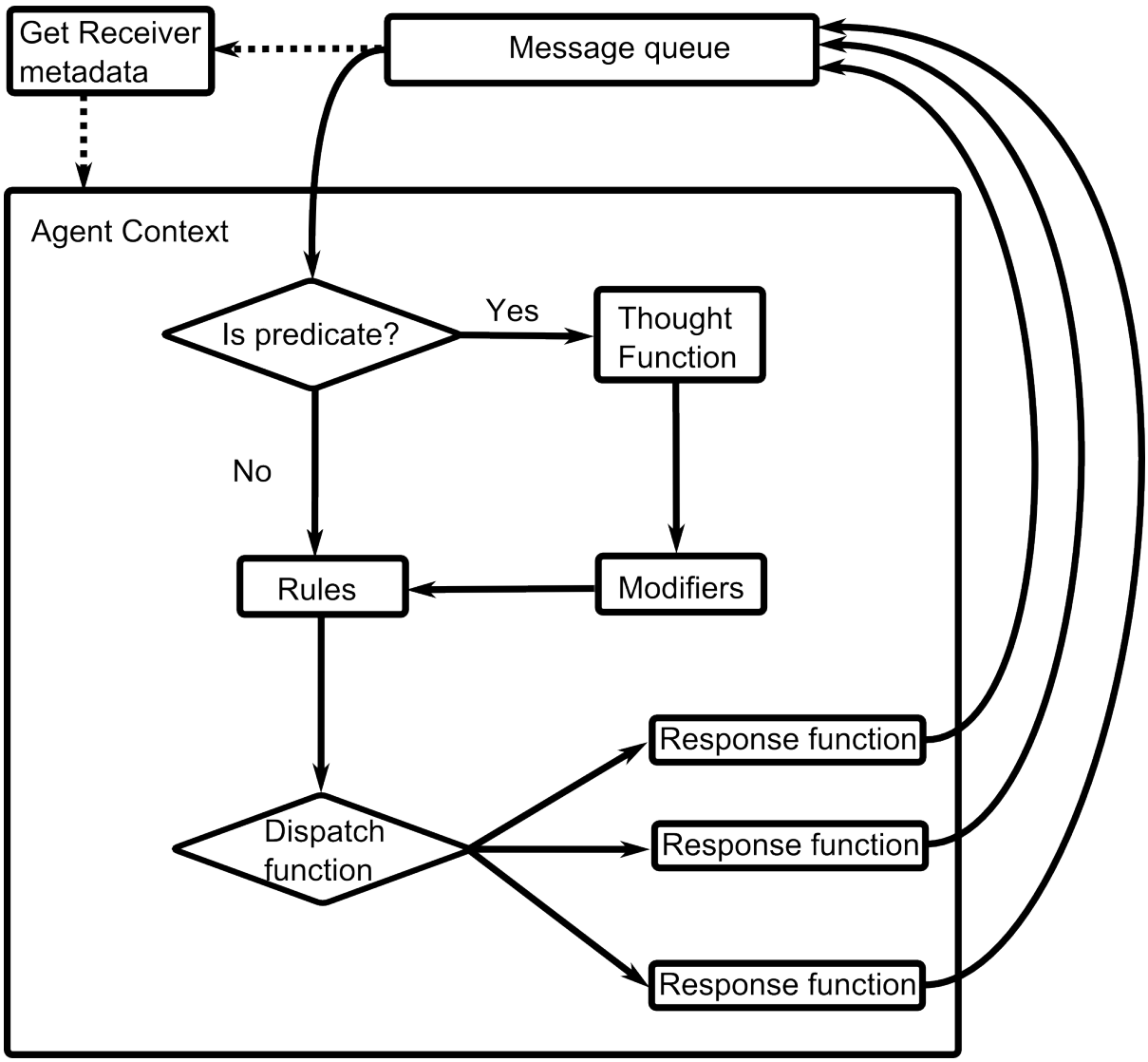

Figure 4.11: An overview of the message processing cycle.

The first thing that happens in any given update cycle is that a message is popped off of the global message queue. The Receiver metadata is used to determine which agent is to process the message. That agent's context is used for the rest of the operations. If the message is a predicate, it is passed to the agent's thought function, and the results of the thought function are potentially processed as modifiers. Either way, rules are executed once per update cycle, and then the message is passed to a dispatch function, which selects one of many possible response functions. After response has finished, the response message is pushed into the global message queue, and the process starts again.

Chapter 5

Evaluation and Discussion

In order to demonstrate the feasibility of first-order logic expressions as a means of representing dialogue in games, we will examine a few example scenarios. Using each of these scenarios, we will show that the current system has many of the attributes of current technologies, such as dialogue trees, and can mimic the behaviour of dialogue trees. Additionally, we will show that, because of the fact that first-order logic expressions are computationally meaningful and well-structured, there is more flexibility and dynamicity inherent in the current system than there is in dialogue trees.

5.1 Bobby and Sally

5.1.1 Premise

Sally Sunshine has just broken up with her boyfriend, and is sitting alone in Jack Hipster's Seattle-Style Coffee Cafe. Bobby Blammo, who is aware of the situation, wants to ask Sally out on a date. The problem is that Sally, being emotionally sensitive, is likely to burst into tears when the topic of dating or relationships comes up. Bobby must find a way to change Sally's emotional state to a better one through the use of dialogue. There are two ways Bobby can do this: by helping Sally forget her problems and cheering her up, or by reducing her sense of

judgement by buying her caffeine.

5.1.2 Concepts Demonstrated in This Scenario

Ability for Autonomous Agents to Participate in Dialogue Bobby and Sally are both autonomous agents which make their own decisions as for how to act. The user acts only to advance the scenario in this case.

Changing Outcomes Based on Agent Attributes Bobby can be put into either "nice mode" or "jerk mode", which affects his strategy to ask Sally out, and therefore also affects Sally's emotional state.

5.1.3 Implementation

As this scenario is intended to demonstrate the capacity for the system to handle autonomous agents, user interaction will not be a part of this system. Instead, the agent Bobby will take the place of the user, with Sally's behaviour being mostly reactionary to Bobby's behaviour. A rudimentary "AI" will be developed to handle Bobby's actions.

Sally's behaviour is relatively simple, and is demonstrated in Figure 5.1. Sally maintains a simple state, consisting primarily of an emotion level. Zero represents neutral emotions, negative numbers represent "bad" emotions, and positive numbers represent "good" emotions. The emotion level is delimited by two thresholds: a lower threshold of -2.0 and an upper threshold of 2.0.

A number of rules are implemented (which are covered in more detail shortly) that dictate how Sally's emotions change. Talking about things she enjoys (for instance, butterflies, kittens, or hagfish (we will pretend she's a marine biologist)) will cause an increase in emotion. As Sally receives coffees, her emotions will also be affected. If she does not like the coffee, it causes her emotions to go down. However, if she does like the coffee, her emotions will go up, but only to a certain point. After consuming 5 or more coffees of any kind, she will begin to

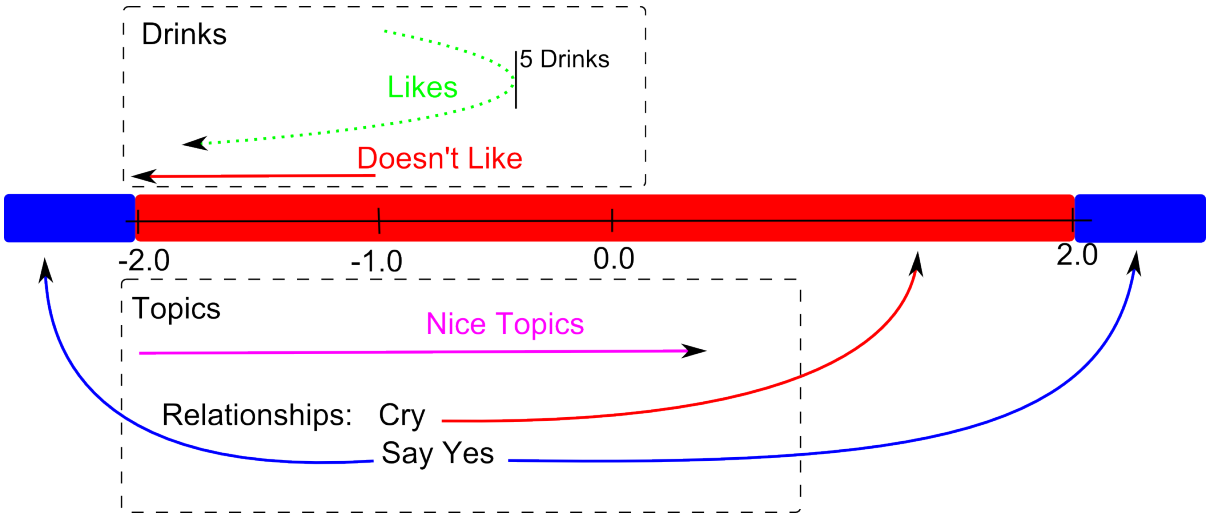

Figure 5.1: Sally's emotion levels. The red zone represents when she will start crying if asked out.

feel unwell, lowering her emotion level. It is not illustrated in the image, but a higher number of coffees causes the emotional increase of talking about "nice" things to increase (i.e. after she's hyped-up on caffeine, it's easier to cheer her up).

There are three possible scenarios that end the interaction. If the topic of relationships is brought up at all while Sally's emotions are between the thresholds, her only response to any further questions will be to cry. However, if asked outside the thresholds, she will say yes. For the sake of example, let's say that higher emotion levels lead Sally to like Bobby, while lower emotion levels with high caffeine content lead her to make quick decisions, and she will say yes to Bobby even if she does not really want to (in her un-caffeinated state of mind).

Message Structure

For this particular scenario, an extension to the message metadata was introduced. Messages now carry emotion and topic information. For emotion information, the floating-point emotion value of the current agent is associated with the *:emotion* field of the metadata. Conceptually, this is a crude representation of an agent's "tone of voice", form which other agents can judge their emotions. The *:topic* field of the message's metadata stores a tag identifying what is being talked about (i.e. relationships, or one of the many "nice" topics).

Sally's Thought Function

Sally's thought function is displayed in Figure 5.2. Recall that thought functions (discussed in Section 4.2.1) are implemented as higher-order predicates. The parameter p is a predicate, and the parameter q is a logic variable.

The first step in this function is to isolate the topic information from the predicate. (Recall that when a predicate message is sent, the message's format is that of the predicate itself. Because functions are Clojure objects, they can have metadata just like any other object.) Once the topic data is isolated, if it exists (i.e. is not nil), then an assertion that the topic has indeed been talked about is added as a possible substitution for q. Afterwards, one of three conditions is committed to:

Sally is crying The only other valid substitution for q is the symbol "Crying". She's too upset to talk.

The message topic is "Relationships" In this case, she may or may not start crying, in which case she keeps crying (by asserting that is-crying is true). If she does not start to cry, she answers the query as normal.

Other In all other conditions, she simply answers the query by applying the predicate p to the logic variable q.

Sally's Rules

The rules in this particular scenario are like other rules for the most part, except that a custom format is used for triggers, so that rules could be more easily examined and triggered by the AI.

There are two primary rules that are used for sally: the coffee rule and the nice-topic rule. Roughly, the coffee rule states that Sally's emotion level will increase by 0.1 if given a coffee that she likes, and decrease by 0.1 if given a coffee that she does not like. However, any coffees

```
(defn sally-thought

  [p q] ;; Parameter list

  (fresh [f]
         (let [;; t is the topic of p's metadata
               t (get (meta p) :topic)]

           (conde
            ( (if (not= t nil)
                (== q (ast talked-about t))
                fail) )

            ;; conda is different from conde
            ;; when the head of one clause executes, conda commits to
            ;; that clause, even if it fails further down the road.
            ;; Similar to Prolog's cut operator (comma)
            ( (conda
               ;; No matter what, if sally is crying, she STAYS crying
               ( (is-crying true)
                 (== q Crying)
                 )
               ;; If relationships are talked about, then they may or
               ;; may not cause sally to cry
               ( (== t Relationships)
                 (emotion-level f)
                 ;; We have to project f in order to use < and >,
                 ;; which are non-relational operators (i.e. normal
                 ;; functions).  It's messy, but needs to be done!
                 (project [f] (cond
                               (< f sally-lower-threshold) (p q)
                               (> f sally-upper-threshold) (p q)
                               :else (conde
                                      ( (== q Crying) )
                                      ( (== q (ast is-crying true)) )))))
               ;; If sally's not crying and not being asked about
               ;; relationships, then the predicate is executed as normal
               ( (p q) )))))))
```

Figure 5.2: Sally's thought function.

after the fifth will *always* cause a decrease. The nice-topic rule states that Sally's emotions will increase if Bobby talks about something she likes (her favourite topics are butterflies, kittens, ponies, lazors, and hagfish). However, the increase in emotion is higher if Sally is more caffeinated. So, at a certain point (if Bobby is being nice), talking to Sally becomes more emotionally optimal than buying her coffees.

One of the rules, called *coffee-inc*, will be examined in detail. This rule deals with affecting Sally's number of coffees and emotion level.

Beginning with Figure 5.3, after variable declarations, the main part of the predicate begins. The line *(relso have-coffee)* confirms that the *have-coffee* relation has been updated. If that is true, then d is a coffee that has been had, nd is the number of coffees so far, and f is the current emotion-level. Afterwards, one of two conditions is true: either Sally likes the coffee d, or she hates the coffee d. In either case, a map of relations to tuples is unified with q.

When the rule executes, each of these tuples will be retracted (removed) from its corresponding relation. The map is then passed to the update function, where the values are changed, and re-asserted into the database. Looking more closely, you will find that not all tuples that are retracted are re-asserted. The *have-coffee* relation is retracted, indicating that the coffee has been consumed, but is not re-asserted or changed.

In Figure 5.4, we see the triggers. Triggers are presented as a list (technically, a vector) of maps with fields *:ex-val* and *:ast*. The *:ex-val* field stores a function that can be used to predict the results of using this trigger, and the *:ast* field stores a message that could be used to affect the change.

In this particular case, there are two triggers. Each takes two parameters: a floating-point value f and an integer value n (note that no type signatures are present, as this is a dynamically-typed programming language). The *:ex-val* functions mirror the different outcomes of the update function.

```
(def coffee-inc-rule
  "Rule that deals with coffees."
  (make-rule
   (ch-pred ;; Relations that are allowed to be modified by this rule.
            [num-liked-coffees num-hated-coffees emotion-level]
            ;; Variable declarations
            q [f d n nd] :def
            ;; Here's the meat of the predicate
            ( (relso have-coffee)
              (have-coffee d)
              (is-coffee d)
              (num-coffees nd)
              (emotion-level f)
              (conda
               ( (hates-coffee d)
                 (num-hated-coffees n)
                 (== q {:nd nd
                        num-hated-coffees [n]
                        emotion-level [f]
                        have-coffee [d] ; Single out this coffee for retraction
                        }) )
               ( (likes-coffee d)
                 (num-liked-coffees n)
                 (== q {:nd nd
                        num-liked-coffees [n]
                        emotion-level [f]
                        have-coffee [d] ; Single out this coffee for retraction
                        }))) )
            ;; Update function follows
            (fn [{[nl] num-liked-coffees
                  [nh] num-hated-coffees
                  [f] emotion-level
                  nd :nd
                  :as the-map}]
              (cond
               nl {num-liked-coffees [(inc nl)]
                   emotion-level (if (< nd 4) [(+ f 0.1)]
                                              [(- f 0.1)])}
               nh {num-hated-coffees [(inc nh)]
                   emotion-level (if (< nd 4) [(- f 0.1)]
                                              [(- f 0.3)])})))
```

Figure 5.3: The predicate portion of the coffee-inc rule, specifying preconditions and update functions.

```
;; Here are the triggers
[
 ;; The liked-coffee trigger
 {:ex-val (fn [f n]
            (if (< n 4)
                (+ f 0.1)
                (- f 0.1)))
  :ast (predicate _ [d]
                  (ast have-coffee d)
                  (is-coffee d)
                  (likes-coffee d))}
  ;; The hated-coffee trigger
  {:ex-val (fn [f n]
             (if (< n 4)
               (- f 0.1)
               (- f 0.3)))

   :ast (predicate _ [d]
                   (ast have-coffee d)
                   (is-coffee d)
                   (hates-coffee d))
   }
  ]))
```

Figure 5.4: The trigger portion of the coffee-inc rule, which can be examined by the AI.

Bobby's AI

Bobby's AI implementation is very closely tied to the trigger format of the rules. The Bobby "AI" (the term AI is used very loosely here) is a simple system that attempts to either maximize or minimize Sally's emotion level. After the level is above or below one of the thresholds, Bobby proceeds to ask her out. Bobby can be placed in one of two modes: *:nice* or *:jerk*, which affect his behaviour.

Conceptually, Bobby's AI is simple, and is shown in Figure 5.7. If Bobby does not yet have information about Sally's rules, he will ask for her rules. This can be thought of as a very crude version of "tell me about yourself". However, it's more like "give me a set of rules that deterministically predicts your emotional state" (life might be easier if such questions could be asked).

Once the rules are obtained, the triggers from each of the rules are aggregated into a single list. The number of coffees (n) and emotion level (f) of Sally would have been obtained earlier by accessing the metadata of Sally's last message. After the triggers are aggregated, the :ex-val function of each is applied to f and n in order to predict the effect on Sally's emotions.

From there, it's fairly simple. If Bobby is being a jerk, the trigger with the minimum value is selected, and if he's being nice, the trigger with the maximum value is selected.

5.1.4 Analysis

An example run is shown in Figures 5.5 and 5.6. Bobby begins by saying "Hello". Sally responds with her own greeting. Bobby then asks for Sally's rules, which are returned. Now that he has the rules, he deduces that the best option for now is to give her a coffee that she likes. This continues until she has four coffees, at which point talking about a nice topic becomes more optimal. By the time Hagfish are talked about, Sally's emotion level is above the threshold of 2.6. At this point, Bobby asks her to go out, and she says yes.

```
Bobby:
Hello
Emotions:  nil
Coffees:  nil

Sally:
Hello
Emotions:  -1.0 ;; She is sad
Coffees:  0

Bobby:
(clojure.core/fn [q] (ruleso q))
Emotions:  nil
Coffees:  nil

Sally:
;; These are the rules... they're very complicated
(#< clojure.lang.AFunction\$1@842f22>
... )
Emotions:  -1.0
Coffees:  0

Bobby:
;; Have a coffee that you like
(fn
  [_]
  (fresh [d]
      (is-coffee d)
      (likes-coffee d)
      ((fn [_ d] (== _ (ast have-coffee d))) _ d)))
Emotions:  nil
Coffees:  nil

Sally:
((ast have-coffee Latte)) ;; I drank a Latte
Emotions:  -0.9 ;; Emotion level has gone up!
Coffees:  1
```

Figure 5.5: Bobby and Sally's conversation.

```
;; This continues until...
Sally:
((ast have-coffee Latte)) ;; I drank a Latte
Emotions:  -0.6 ;; Emotion level has gone up again!
Coffees:  4 ;; She's getting more caffeinated

;; Bobby switches gears!
;; Bobby: picks a nice topic to talk about
(fn [q] (fresh [t]
           (nice-topic t)
           (== q (ast talked-about t))))

Sally:
( (ast talked-about Butterflies) )
Emotions: 0.0 Coffees: 4

Sally:
( (ast talked-about Kittens) )
Emotions: 0.6 Coffees: 4

;; Bobby continues to send the same message, with
;; a different result each time!
Sally:
( (ast talked-about Ponies) )
Emotions: 1.2 Coffees: 4

Sally:
( (ast talked-about Lazors) )
Emotions: 1.8 Coffees: 4

Sally:
( (ast talked-about Hagfish) )
Emotions: 2.6 Coffees: 4

;; The threshold has been reached!
Bobby:
(fn [q] (go-out Sally Bobby q)) ;; Will Sally go out with Bobby?
Emotions:  nil Coffees:  nil

Sally:
( Yes )
Emotions: 2.6 Coffees: 4
```

Figure 5.6: Bobby and sally's conversation.

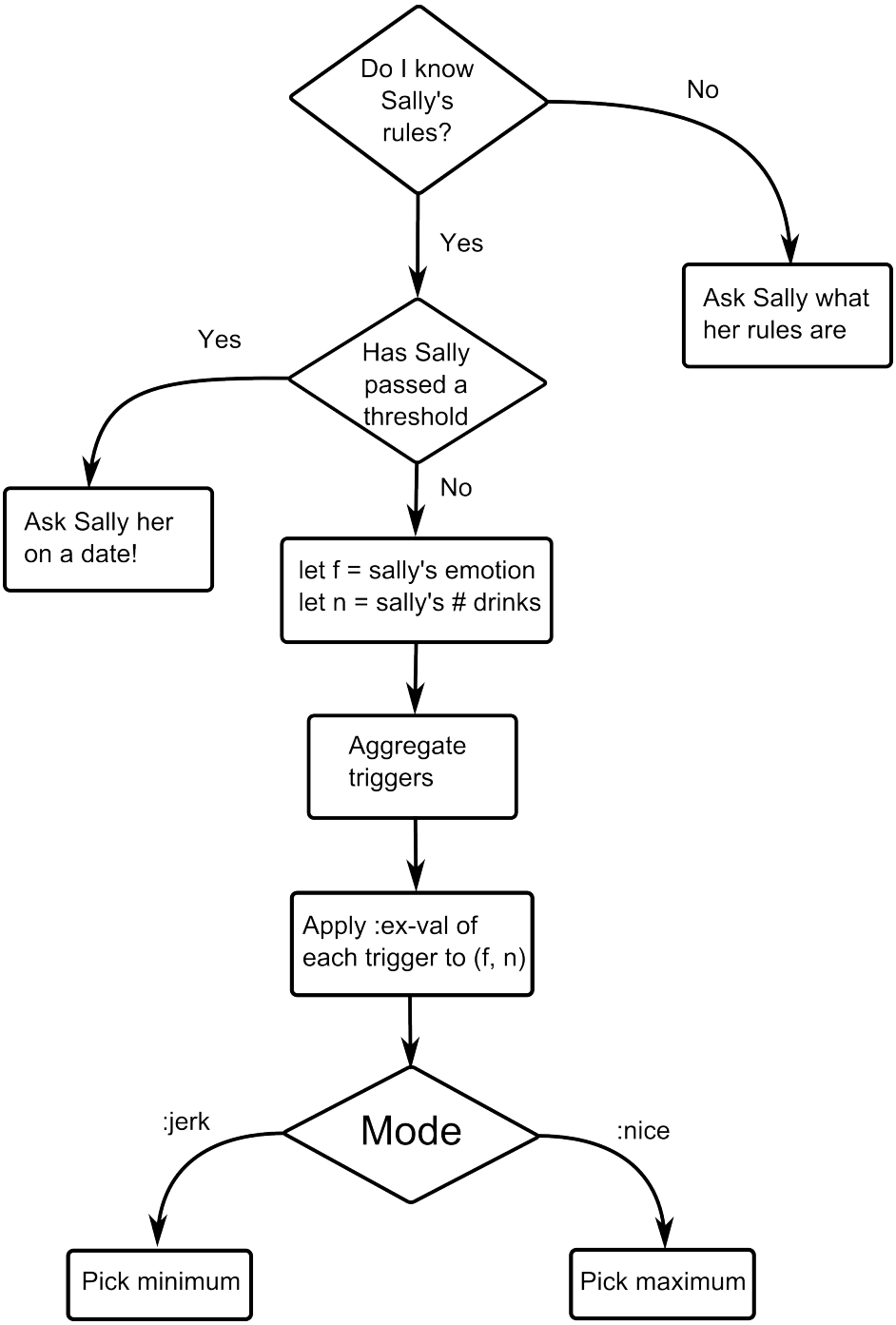

Figure 5.7: Bobby's AI process.

5.2 Lovecraft Country

5.2.1 Premise

The book *Game Writing: Narrative Skills for Video Games* [3] contains an example dialogue tree in its chapter on dialogue engines. The same dialogue tree was shown at the end of Chapter 2, in Figures 2.1 and 2.2.

The dialogue tree is described in the book as "a conversation between a protagonist who is an insurance investigator and a mechanic in a game based around otherwordly events in a small town (in the manner of an HP Lovecraft story)" [3].

5.2.2 Concepts Demonstrated in this Scenario

Expressive Equivalence with Dialogue Trees Because an example dialogue tree is being used as the basis for this example, replication of the behaviour of the dialogue tree will demonstrate that this system can mimic the behaviour of dialogue trees.

Changing Outcomes Based on Agent Attributes By changing the attributes of the Mechanic agent, as well as introducing a new Apprentice agent to take his place with a different set of attributes, the structure and content of the dialogue is changed.

Interaction With Other Systems An effort was made to intentionally separate aspects of the data representation used in this scenario from those in other scenarios. This was done in order to simulate the presence of an external system with an external data model, and do show that the logic programming aspect of this system was flexible enough to interact with potential external data models.

5.3 Preparation

Simplifying the Dialogue Tree

A closer examination of the example dialogue tree suggests that it may be simplified without losing any information. Many of what appear to be distinct choices lead to the same node. These can easily be pruned. Additionally, some sets of nodes form linear sequences (such as nodes M5, M6, M7). These nodes can be represented as single nodes. Therefore, we will be working with the dialogue tree shown in Figures 5.8 and 5.9.

Identifying Message Groupings Because this system works best when agents communicate in an alternating way, the dialogue tree will be modified to reflect that. Sequences of messages by the same agent will be grouped together. Where possible, they will be combined into a single node (as M5, M6, and M7 in the previous tree have all fused to form M5 in this tree). However, they will be added to groups where this is not possible. Message groupings are shown in dotted-line boxes in the diagram. These groupings encapsulate sequences of messages that can be represented as a single message. For instance, M6 and M7 represent a message group. Depending on the player's choice at the previous node, the NPC's next message will either be M7 or a combination of M6 + M7, but in either case will be a single message.

5.3.1 Implementation

Data Representation

Agent attributes in this scenario were described mostly with hash maps rather than by using relations. One such map is shown in Figure 5.10. Certain basic attributes, such as the agent's job and emotion level are present in the map, as well as some more advanced data.

The *:event-log* and *:general-descs* fields are used as a stand-in for potential agent memory models. The *:event-log* field, in particular, is worth noting. This represents a time-stamped log of events in the agent's memory. For instance, at 21:00:00 (9:00 PM), the agent drove to the

M1 : “You a cop?”

- A “No, I’m a claims adjuster.” [go to M3]
- B “Private investigator. I have a few questions.” [go to M2]

M2 : “Nah, you’re dressed too smart. I reckon you work for the insurers.”

- A “Lucky guess.” [go to M3]

M3 : “I reckon this counts as an act of God.”

- A “What happened?” [go to M4]
- B “I don’t believe in God. I reckon this was an act of man.” [go to M3a]

M3a : “Yeah, right – you didn’t see those lights coming down from the sky – you didn’t see those bodies... the blood...” [go to M4]

M4 : “It was horrible... I don’t even want to think about it.”

- A “I’ll come back tomorrow. Give you a chance to recover.”[go to M7]
- B “I need to know what happened, or we can write off your claim here and now.” [go to M5]

M5 : “I drove up to the farm that night. It was quiet. I mean, dead quiet. Even the cicadas weren’t making a sound, and those critters are deafening most nights. When I got close, the engine just died on me. I started checking the sparks, and then it happened. Lights, I mean, something glowing, just pouring down from the heavens on the farmhouse. It was beautiful and terrible and... I don’t know. But, when I go to the farm... My God...”

- A “This is obviously very difficult for you. I’ll come back tomorrow.” [go to M7]
- B “Go on. You’re just getting to the good bit.” [go to M6]

M6 : “They were lying around outside the farmhouse... I thought maybe a cow had been struck by lightning. But they weren’t. They didn’t even look human...” [go to M7]

M7 : “I’m sorry... I’ve never... I just...”

- A “This has obviously been hard on you. Get a good night’s sleep... I’ll se you in the morning.” [End]

Figure 5.8: The simplified dialogue tree.

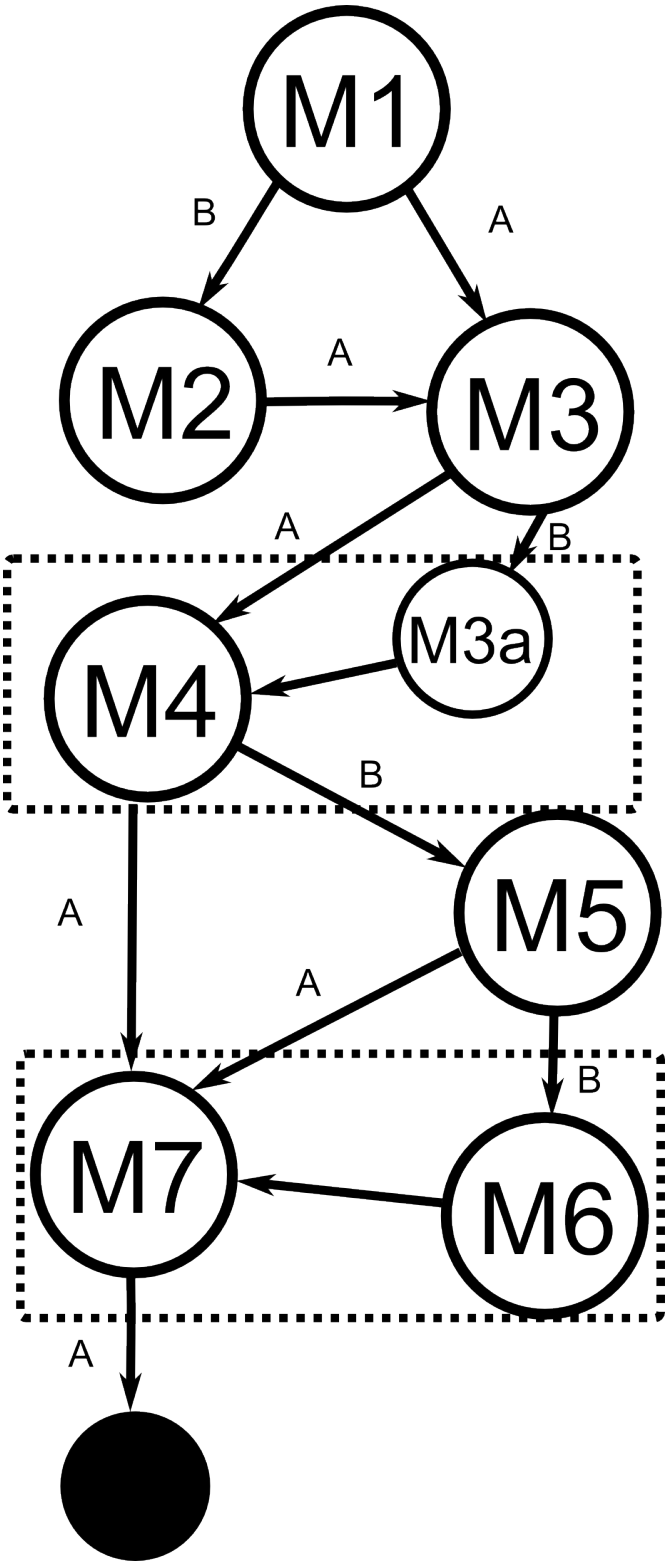

Figure 5.9: The simplified dialogue tree diagram, showing message groupings dotted-line boxes.

```
Mechanic     {:job MechJob
             :described-events #{}
             :emotion-thresholds {Stop-Talking #(> % -0.8)
                                  Breakdown #(> % -2.0)}
             ;; A set of general descriptors
             :general-descs {1 {:desc [Horrible Last-Night-Event]
                                  :e-val -0.1}
                             2  {:desc [Bloody Last-Night-Event]
                                   :e-val -0.1}
                             3  {:desc [Supernatural Last-Night-Event]
                                   :e-val -0.1}
                             }
             :event-log {[21 00 00] {:desc [Drove-To Farmhouse]
                                     :e-val -0.1}
                         [21  1 00] {:desc [Quiet Farmhouse]
                                     :e-val -0.1}
                         [21  2 00] {:desc [Quiet Cicada]
                                     :e-val -0.1}
                         [21  5 00] {:desc [Broke Engine]
                                     :e-val -0.1}
                         [21  7 00] {:desc [Came-From Glowing-Thing Sky]
                                     :e-val -0.2}
                         [21  8 00] {:desc [Afraid Self]
                                     :e-val -0.15}
                         [21  9 00] {:desc [Near Farmhouse Bodies]
                                     :e-val -0.3}
                         [21 10 00] {:desc [Inhuman Bodies]
                                     :e-val -0.3}}
             :emotion-level 0.0
             }
```

Figure 5.10: Example data map for the Mechanic character, used to represent potential external data model.

farm house, at 21:01:00, the agent noted that the farm house was quiet, and so on. Each event has a simple description (*:desc*) as well as an associated emotion value (*:e-val*, which describes how happy or unhappy the agent is to recall the event. The *:general-descs* field is very similar to the event log, but contains general descriptors of events which are not time-stamped (the indices 1,2,3 exist for implementation reasons).

A predicate called *desc-events* is provided. It takes two parameters: a range *[mn mx]* of times used to filter memories from the event log, and a logic variable q. Calling this predicate from a logic programming context will result in q being unified with each event in the range the order that they occurred.

Emotion Thresholds In the original dialogue tree, when the mechanic is asked to describe the events that took place, he hesitates after describing most of them. The player can then choose to force the mechanic to further describe the events or can say goodbye, ending the conversation. In order to replicate this behaviour, the mechanic is given an emotion threshold for events that can be described in a single utterance. If the combined *:e-val* values exceed this threshold, the tag "Cant-Continue" is appended to the end of the message.

Additionally, another overall emotion threshold was added, which causes the agent to enter a hysterical state if the overall emotional trauma (of all events described so far) reaches a certain threshold. In the default case, however, this does not occur.

Example Run

Figure 5.11 shows an example run that corresponds approximately to the following path through the dialogue tree in Figure 5.9: M1, M2, M3, M3a, M4, M5, M6. By the end of the conversation, the mechanic's emotion level is too low, and he descends into a state of hysteria ("I'm sorry... I never... I just...").

The primary difference between this example run and the actual dialogue tree is that this model is somewhat more dynamic than the dialogue tree model. Attributes of the agent can

```
Current message: (fn [_] (agent-attribute
*current-agent* [:job] CopJob))
;; You a cop?

Respond: Yes
Current message: ((fn [_] (event-descriptor
Last-Night-Event Act-Of-God)) [:deny CopJob ] )
;; I don't believe you. (I deny that you are a cop.)
;; Was last night's event an act of God?

Respond: No
Current message: (nil [Horrible Last-Night-Event ] [Bloody
  Last-Night-Event ] [Supernatural Last-Night-Event ] )
;; Last night's event was horrible and bloody and supernatural
;; (i.e. it must have been an act of God)

Respond: (predicate q [] :def (desc-events last-night q))
;; What happened last night

Current message: ([Drove-To Farmhouse ] [Quiet Farmhouse ] [Quiet
  Cicada ] [Broke Engine ] [Came-From Glowing-Thing Sky ] [Afraid
  Self ] Cant-Continue )
;; I drove to the farm house.  It was quiet.  The cicada's weren't
;; making a sound.  The engine broke.  A glowing thing came from
;; the sky.  I was afraid. (that's all)

Respond: (predicate q [] :def (desc-events last-night q))
;; (keep describing) What happened last night?

Current message: ([Near Farmhouse Bodies ] [Inhuman Bodies ]
Cant-Continue )
;; Bodies were near the farm house.  They weren't human.

Respond: (predicate q [] :def (desc-events last-night q))
;; keep describing...

Current message: (Hysteria )
;; The mechanic can't continue describing.  It's too emotionally taxing.

Respond: Goodbye
Current message: Goodbye
```

Figure 5.11: Example run of the "Lovecraft Country" scenario that follows the dialogue tree fairly accurately.

be changed in order to produce outcomes, but have been specifically tuned here to simulate the example dialogue tree. For instance, there is a pre-existing belief in the Mechanic agent's knowledge base which states that he believes the main character is an insurance investigator, so it does not matter if the player says "Yes" or "No" if asked that they are a police officer, the Mechanic's beliefs will stay the same.

In Figure 5.12, we see some variations on the original path through the dialogue tree. We can see that, in this case, rather than claiming to be a cop, the player says no, prompting the Mechanic to say "I accept that you're an investigator", as would be the case if the dialogue went from M1 to M3 directly, rather than going to M2.

Additionally, by not denying that the event was an "act of god", the mechanic character is able to skip some of the description from M3a, but the rest of the dialogue flows properly. Note that, at the end of the conversation, the Mechanic simply states that nothing is left to say, rather than becoming hysterical. This is because, by skipping the description in M3a, the emotion threshold was not reached.

Changing Agent Attributes

By changing some of the agent's attributes, as well as adding or removing data, the dialogue tree's structure can be deviated from. Figures 5.13 and 5.14 shows some variations.

In Figure 5.13, the Mechanic's pre-existing beliefs about the Investigator character's job are removed, so when the user responds "Yes" to being asked whether or not they are a cop, that explanation is accepted. Additionally, the descriptor "Homocide", rather than "Act of God" is associated with the cop job.

In Figure 5.14, the effect of lowering the Mechanic's emotional threshold is shown (a low threshold means the mechanic can withstand more negative emotion). Additionally, a few more time-stamped events are added to the agent's "memory". By increasing the emotional fortitude of the mechanic, he is able to describe more to the player, divulging that the bodies were faceless, ugly, and looked like Kevin Bacon (a terrifying thought indeed!).

```
Current message: (fn [_] (agent-attribute *current-agent*
[:job] CopJob))
Current message: No

Current message: ((fn [_] (event-descriptor Last-Night-Event
Act-Of-God)) [:accept InvJob ] )

Current message: ((fn [_] (event-descriptor Last-Night-Event
Act-Of-God)) [:accept InvJob ] )

Respond: Yes

Current message: ( Yes )

Respond: (predicate q [] :def (desc-events last-night q))

Current message: ([Drove-To Farmhouse ] [Quiet Farmhouse ] [Quiet
  Cicada ] [Broke Engine ] [Came-From Glowing-Thing Sky ] [Afraid
  Self ] Cant-Continue )
Here's what you think:  nil

Respond: (predicate q [] :def (desc-events last-night q))

Current message: ([Near Farmhouse Bodies ] [Inhuman Bodies ]
Cant-Continue )

Respond: (predicate q [] :def (desc-events last-night q))

Respond: (predicate q [] :def (desc-events last-night q))

Current message: ()
;; Nothing left to describe

Respond: Goodbye
Current message: Goodbye
```

Figure 5.12: Variations on the first dialogue run, corresponding to possible variations in the dialogue tree.

```
;; Example 1
Current message: (fn [_] (agent-attribute
*current-agent* [:job] CopJob))
Here's what you think:  ()
Respond: Yes

Current message: ([:accept CopJob ] (fn [_]
(event-descriptor Last-Night-Event Homocide)) )
```

Figure 5.13: The agent has no pre-existing beliefs about the player's job.

```
Current message: (clojure.core/fn [q] (desc-events last-night q))
Current message: ([Drove-To Farmhouse ] [Quiet Farmhouse ] [Quiet
  Cicada ] [Broke Engine ] [Came-From Glowing-Thing Sky ] [Afraid
  Self ] Cant-Continue )
Respond: (predicate q [] :def (desc-events last-night q))
Current message: ([Near Farmhouse Bodies ] [Inhuman Bodies ]
Cant-Continue )
Respond: (predicate q [] :def (desc-events last-night q))
Current message: ([Faceless Bodies ] [Ugly Bodies ] Cant-Continue )
Respond: (predicate q [] :def (desc-events last-night q))
Current message: ([Looked-Like Bodies Kevin-Bacon ] )
```

Figure 5.14: Further descriptions are possible if the threshold has changed.

Adding Agents and Attributes

Any number of such properties may be changed in order to create a number of potentially unique dialogue tree-like paths. Additionally, a real game engine, the data, such as the event log or current emotional state and knowledge of agents could be modified dynamically throughout the course of gameplay, creating unique and unforeseen variations in dialogue structure.

In this section, we will introduce the concept of multiple agents and show how, using the same data, combined with different agent attributes, they are able to provide varying descriptions of the same situation.

Firstly, wherever an emotional reaction is demanded of the agent, rather than use definite descriptors such as “Afraid” or “Supernatural”, we can introduce a set of symbols that represent certain “variant” reactions.

Figure 5.15 shows some examples of how specific descriptors can come to be replaced with variant descriptors. If we mark such descriptors with metadata identifying them as variants, then functions can easily be defined which postprocess these variants to more concrete identifiers based on agent attributes.

In the following examples, we will introduce some new attributes, “bravery” and “manliness”, which affect how the agents respond to certain variant identifiers. We will introduce a new agent, the “Apprentice”, representing the mechanic’s apprentice. He has the same event descriptors as the mechanic, but different attributes, which cause him to describe the same event differently. An example conversation is shown in Figure 5.16.

In this conversation, the events described are the same, but the agent’s attributes change how he responds. Firstly, his emotion thresholds are lower, which causes him to be able to describe more at once. Also, his bravery is higher, which causes him to replace terms like “Afraid” or “Horrible” that the mechanic would have used with other terms like “Unafraid” and “Pretty bad”. Additionally, the higher manliness attribute causes him to describe the bodies as looking like Bruce Campbell (rather than a pretty boy like Kevin Bacon).

```
:general-descs
{ 1 {:desc [Horrible Last-Night-Event] :e-val -0.1}
  2  {:desc [Bloody Last-Night-Event] :e-val -0.1}
  3  {:desc [Supernatural Last-Night-Event] :e-val -0.1}
}

;; Define some symbols
;; The def-m-type macro (created for this project) is like "enum"
;; in other languages,  except that it defines a set of symbols
;; which all have type metadata, and are derived from another
;; type.
;; This one introduces ::VarDesc, which descends from ::Variant,
;; and includes all further symbols.
(def-m-type VarDesc ::Variant
    StrangenessOpinion
    GoreOpinion
    BadnessOpinion)

;; If it's not a variant, just have it return itself
(defmethod agent-attr-sym :default
    [_ s] s)

;; This function checks the manliness and bravery of an agent.
;; An actor impression should be Bruce Campbell if you're manly,
;; but Kevin Bacon if you're girly!
;; Similarly, your fear reaction depends on the bravery value.
(defmethod agent-attr-sym Variant
  [{br :bravery
    ml :manliness
   :as agent} sym]
  (case sym
       Actor-Impression (cond  (> ml 0.5) Bruce-Campbell
                               (<= ml 0.5) Kevin-Bacon)
       FearReaction (cond (> br 0.5) Unafraid
                          (<= br 0.5) Afraid )
       ;;; ... etc
       ))

;; Replaced with:
:general-descs
{ 1 {:desc [BadnessOpinion Last-Night-Event] :e-val -0.1}
  2  {:desc [GoreOpinion Last-Night-Event] :e-val -0.1}
  3  {:desc [StrangenessOpinion Last-Night-Event] :e-val -0.1}
}
```

Figure 5.15: Examples of variant symbols.

```
Current message: (fn [_] (agent-attribute
*current-agent* [:job] CopJob))
;; "You a cop?"

Respond: No

Current message: ([[:accept InvJob ] Act-Of-God ] [:accept InvJob
] (fn [_] (event-descriptor Last-Night-Event
Act-Of-God)) )
;; "You're an insurance agent.  Do you think last night's event
;; was an act of god?"

Respond: No

Current message: ((Pretty-Bad Last-Night-Event ) (Kind-of-Gross
Last-Night-Event ) (Weird Last-Night-Event ) )
;; "Well, it was pretty bad, and kind of gross, and weird."

Respond: (predicate q [] :def (desc-events last-night q))
;; "What happened last night?"

Current message: ((Drove-To Farmhouse ) (Quiet Farmhouse ) (Quiet
Cicada ) (Broke Engine ) (Came-From Glowing-Thing Sky ) (Unafraid
Self ) (Near Farmhouse Bodies ) (Inhuman Bodies ) Cant-Continue )
;; "We drove up to the farmhouse, and it was quiet.  The cicadas
;; weren't making a sound.  The engine broke, and then some glowing
;; thing came down from the sky.  But I wasn't afraid.  There were
;; some inhuman bodies near the farmhouse..."

Respond: (predicate q [] :def (desc-events last-night q))
;; "Keep going."

Current message: ((Faceless Bodies ) (Ugly Bodies ) (Looked-Like
Bodies Bruce-Campbell ) )
;; "They were faceless, and ugly, and kind of looked like Bruce Campbell."

Respond: (predicate q [] :def (desc-events last-night q))
Current message: ()
;; Nothing left to describe.

Respond: Goodbye
Current message: Goodbye
```

Figure 5.16: Mechanic's apprentice conversation.

5.3.2 Analysis

In the example runes shown above, we have demonstrated how the simple and rigid structure of a dialogue tree can be not only replicated, but also extended and varied without requiring a re-write of the tree. Additionally, we have shown that dialogue structures can be generated from data accumulated in an agent's memory rather than by authors. Such data can also be postprocessed to match an agent's worldview and interpretation for further variability.

5.4 Discussion

In this Chapter, we have shown the capabilities of the current system with two scenarios. The first scenario showed that the computational meaningfulness of the first-order logic message format enabled autonomous agents to better understand how to interact using dialogue.

Expressive equivalence with dialogue trees was demonstrated in the second scenario. It is important to demonstrate this in order to show that the current design is capable not only of new behaviour, but of replicating old behaviour. This fact will allow it to be more successfully used in future research.

Each of these scenarios demonstrated the capability of the system to modify its behaviour as the attributes of agents were modified. This is a key feature of any system aiming to produce emergent and dynamic game dialogue; there must be ways to change the way content is presented without changing the content itself.

Chapter 6

Conclusions

In conclusion, we have presented, designed, and developed a framework which allows the use of first-order logic to represent dialogue in games. This system introduces a dialogue format that is computationally meaningful, unambiguous and well-structured, and not only replicates the behaviour of dialogue trees, a current and widely-used game dialogue technology, but also allow for its extension in novel ways.

By demonstrating scenarios of its usage in the previous chapter, we have shown that the properties of the proposed message format allow for the introduction of new and novel features.

The computational meaningfulness allows the effects of the code to be analyzed and predicted, allowing autonomous agents to analyze the effects of sending messages to other agents, allowing dialogue generation. This was demonstrated in the "Bobby and Sally" scenario.

Additionally, the fact that the message format was both computationally meaningful and well-structured meant that the structure of messages could be analyzed by the program in order to be modified, adding meaningful variability to the messages that were sent (i.e. the changing of agent attributes in both the "Bobby and Sally" and "Lovecraft Country" scenarios).

Also, the "Lovecraft Country" scenario demonstrated that, given an existing dialogue tree, the behaviour of that tree could be replicated, effectively meaning that the expressive power of current dialogue tree systems is subsumed by the proposed system.

The combination of these attributes led to the demonstration that the current system has the capabilities of current dialogue technologies, but includes much more capability. Because of this, we believe that it has been demonstrated that the representation of game dialogue as first-order logic expressions can be used as a basis for further research into dynamic dialogue systems.

6.1 Contributions

Examples demonstrate both the effectiveness of our prototype and the feasibility of our approach to dialogue representation and construction. We demonstrate that not only can our approach deliver capabilities of what is currently widely-used in game development, but that the capacity of the dialogue representation to be computationally analyzed and constructed, as well as the ability for existing content to be re-used in emergent ways without requiring it to be re-authored, provides considerably more expressive power than is currently possible. Put another way, this system allows patterns of agents' interactions with dialogue to be specified, rather than specifying the dialogue itself.

6.2 Future Work

It is important to consider future work, as the current system is only a demonstration of how first-order logic expressions can be used as a basis for further research and experimentation with believable agents.

We believe work in the creation of authoring tools to be important. The current system relies too much on direct knowledge of logic programs. Many concepts that should be simple, such as the authoring of rules to increase or decrease emotion levels, are obfuscated by the necessity of working with complex structures such as assertion and retraction Modifiers. Because of the flexibility inherent in the type of programming language used for the implementation of

this system, code generation is simple. Therefore, it is feasible to create tools (such as GUI tools, or even code macros) that would allow those less knowledgeable of the implementation details to accomplish authoring of rules, response functions, and thought functions in the current system.

Also, work into more complex message analysis, rule analysis, and message construction is an important component of future work. Ideally, a fully dynamic dialogue system would be able to generate expressions "from scratch" based on an analysis of the world and other agents, rather than relying on rule trigger templates as the current system does. Such a system, however, would require advanced algorithms and perhaps machine learning techniques, and is beyond the scope of the current thesis.

Such message construction and analysis techniques could benefit from a fuller and more populated set of conversational elements, as well as a means of producing logical dialogue representations what are more closely tied with the structures of natural language. This could be done in conjunction with NLP experts or linguists, and would allow the current system to be more formally and accurately tested against techniques such as dialogue trees.

Another important future step is the embedding of this system in an existing game engine. Having data that is generated outside the context of the system would allow various dialogue generation techniques to be tested against data and agent representations that exist in current games, and to see how well the system is able to fit in with current game technologies.

Bibliography

[1] G. Acton, "Playing The Role : Towards An Action Selection Architecture For Believable Behaviour In Non Player Characters and Interactive Agents," 2009.

[2] T. Anderson, M. Blank, B. Daniels, and D. Lebling, "Zork," 1980.

[3] C. Bateman, Ed., *Game Writing: Narrative Skills for Video Games*, Boston, 2006.

[4] BioWare, *Mass Effect*. Microsoft Game Studios, 2007.

[5] W. E. Byrd, "Relational Programming in miniKanren : Techniques , Applications , and Implementations," 2009.

[6] D. P. Friedman, W. E. Byrd, and O. Kiselyov, *The Reasoned Schemer*. Cambridge, Massachusetts: MIT Press, 2005.

[7] R. Hickey, "Clojure," 2013. [Online]. Available: http://www.clojure.org

[8] A. B. Loyall and J. Bates, "Believable Agents : Building Interactive Personalities," Ph.D. dissertation, Stanford University, 1997.

[9] S. Mascarenhas and A. Paiva, "FAtiMA Modular : Towards an Agent Architecture with a Generic Appraisal Framework," Porto Salvo, Portugal, 2011.

[10] M. Mateas and a. Stern, "A behavior language for story-based believable agents," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 39–47, Jul. 2002. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1024751

[11] M. Mateas and A. Stern, "Writing Façade: A Case Study in Procedural Authorship Michael Mateas and Andrew Stern," 2007.

[12] M. Mateas, A. Stern, and G. Tech, "Façade : An Experiment in Building a Fully-Realized Interactive Drama," *Game Developers Conference (GDC '03)*, vol. 2, 2003.

[13] J. McCoy, M. Treanor, and B. Samuel, "Prom Week: social physics as gameplay," Santa Cruz, California, pp. 1–4, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2159425

[14] J. Mccoy, M. Treanor, B. Samuel, N. Wardrip-fruin, and M. Mateas, "Comme il Faut : A System for Authoring Playable Social Models," 2010.

[15] J. L. Mey, *Pragmatics: an introduction*, 2nd ed. Blackwell, 1993.

[16] D. Nolen, "core.logic." [Online]. Available: https://github.com/clojure/core.logic

[17] R. Powell, "Lojban Introductory Brochure," 2012. [Online]. Available: http://www.lojban.org/tiki/Lojban+Introductory+Brochure

[18] N. Richards, "pldb." [Online]. Available: https://github.com/threatgrid/pldb

[19] R. Rojas, "A tutorial introduction to the lambda calculus," pp. 1–9, 1997. [Online]. Available: http://www.utdallas.edu/ gupta/courses/apl/lambda.pdf

[20] M. Steedman and J. Baldridge, "Combinatory categorial grammar," *Non-Transformational Syntax: Formal and Explicit Models of Grammar*, pp. 1–62, 2011.

[21] P. Tero, "Creating Better NPCs," *Gamasutra*, 2013. [Online]. Available: http://gamasutra.com/blogs/PaulTero/20130318/188686/Creating_Better_NPCs.php

[22] A. M. Turing, "I.Computing Machinery and Intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, 1950. [Online]. Available: http://mind.oxfordjournals.org/cgi/doi/10.1093/mind/LIX.236.433

[23] M. White and J. Baldridge, "OpenCCG," 2011. [Online]. Available: http://sourceforge.net/projects/openccg/

[24] Wikipedia, "Adjacency Pairs." [Online]. Available: http://en.wikipedia.org/wiki/Adjacency_pairs

[25] ——, "Tree(Graph Theory)." [Online]. Available: http://en.wikipedia.org/wiki/Tree_(graph_theory)

[26] ——, "First-class Citizen," 2013. [Online]. Available: http://en.wikipedia.org/wiki/First-class_citizen

[27] L. Zettlemoyer and M. Collins, "Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars," 2012. [Online]. Available: http://arxiv.org/abs/1207.1420

Curriculum Vitae

Name:	Kaylen Wheeler
Post-secondary Education and Degrees:	Brock University St. Catharines, Ontario, Canada 2006-2011 B.Sc.
	The University of Western Ontario London, Ontario, Canada 2011-2013 M.Sc.
Honours and Awards:	Silicon Knights Scholarships 2010
	Ontario Graduate Scholarship 2011-2012
Related Work Experience:	Teaching Assistant The University of Western Ontario 2011-2012