THE ALGORITHMIC EXPANSION OF STORIES

by

CRAIG MICHAEL THOMAS

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
October 2010

Copyright © Craig Michael Thomas, 2010
Abstract

This research examines how the contents and structure of a story may be enriched by computational means. A review of pertinent semantic theory and previous work on the structural analysis of folktales is presented. Merits and limitations of several content-generation systems are discussed. The research develops three mechanisms – elaboration, interpolation, and continuity fixes – to enhance story content, address issues of rigid structure, and fix problems with the logical progression of a story.

Elaboration works by adding or modifying information contained within a story to provide detailed descriptions of an event. Interpolation works by adding detail between high-level story elements dictated by a story grammar. Both methods search for appropriate semantic functions contained in a lexicon. Rules are developed to ensure that the selection of functions is consistent with the context of the story. Control strategies for both mechanisms are proposed that restrict the quantity and content of candidate functions. Finally, a method of checking and correcting inconsistencies in story continuity is proposed. Continuity checks are performed using semantic threads that connect an object or character to a sequence of events. Unexplained changes in state or location are fixed with interpolation.

The mechanisms are demonstrated with simple examples drawn from folktales, and the effectiveness of each is discussed. While the thesis focuses on folktales, it forms the basis for further work on the generation of more complex stories in the greater realm of fiction.
Acknowledgements

I would like to thank my supervisors, Dr. Michael Levison and Dr. Greg Lessard for all their expert support and thoughtful guidance. I greatly enjoyed our discussions, every one of which has helped to shape this dissertation, and will influence my thoughts and research for many years to come. I would like to thank the faculty, staff and my fellow graduate students in the School of Computing for making my time here memorable. I would like to thank my parents and families on both sides for their support. Without their encouragement and enthusiasm none of this would ever have been possible. Finally, I would like to thank my wife Shona for listening to my ramblings about all things semantic.

This research was supported by funding assistance from the Natural Sciences and Engineering Research Council of Canada.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Craig Michael Thomas
October, 2010
## Contents

Abstract i

Acknowledgements ii

Statement of Originality iii

Contents iv

List of Tables ix

List of Figures x

Chapter 1: Introduction 1

1.1 Objectives and Contributions . . . . . . . . . . . . . . . . . . . . . . 2
  1.1.1 Content Enrichment . . . . . . . . . . . . . . . . . . . . . . . 3
  1.1.2 Structure Modification . . . . . . . . . . . . . . . . . . . . . . 3
  1.1.3 Continuity Checking . . . . . . . . . . . . . . . . . . . . . . . 4

1.2 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 A Note on Natural Language Generation . . . . . . . . . . . . . . . . 6

1.4 Organization of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . 6

Chapter 2: Background 8

2.1 Layers in Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Semantics and Natural Language Processing . . . . . . . . . . . . . . 9

2.3 Semantic Formalisms . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.3.1 Fundamental Concepts . . . . . . . . . . . . . . . . . . . . . . . 12
  2.3.2 Required Attributes in a Semantic Formalism . . . . . . . . . 17

2.4 Creativity and Content . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  2.4.1 Definitions of Creativity . . . . . . . . . . . . . . . . . . . . . . 18
  2.4.2 Elements of a Good Story . . . . . . . . . . . . . . . . . . . . . 24
  2.4.3 Story Grammars . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.5 Text Planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  2.5.1 Classification of Systems . . . . . . . . . . . . . . . . . . . . . . 31
  2.5.2 Transformational Systems . . . . . . . . . . . . . . . . . . . . . 32
    2.5.2.1 TALE-SPIN . . . . . . . . . . . . . . . . . . . . . . . . 32
    2.5.2.2 MINSTREL . . . . . . . . . . . . . . . . . . . . . . . . 35
    2.5.2.3 MEXICA . . . . . . . . . . . . . . . . . . . . . . . . . 38

Chapter 3: Implementation 39

3.1 Environment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.2 Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . 41

Chapter 4: Experiments 46

4.1 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.2 Experiment Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

Chapter 5: Conclusion 53

5.1 Summary of Contributions . . . . . . . . . . . . . . . . . . . . . . . . 53

5.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

Appendices A: Sample Output . . . . . . . . . . . . . . . . . . . . . . . . 57

Appendices B: Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.2.2.1 Function Hierarchy ........................................ 103
4.2.2.2 Dramatis Personae ........................................ 106
4.2.2.3 A Note on the Model ...................................... 107
4.2.2.4 A Partial Generation Example ......................... 107
4.3 Discussion ...................................................... 110
4.3.1 Lack of Preparation ......................................... 110
4.3.2 High-Level Descriptions .................................. 112
4.3.3 Structure and Content Predictability ........................ 113
4.3.4 Dramatis Personae and Random Expansion ............... 113
4.4 Summary ....................................................... 114

Chapter 5: Elaboration .............................................. 115
5.1 Elaboration Problems in Generation ............................. 115
5.2 A Brute-Force Solution ......................................... 117
5.3 Fine Tuning ...................................................... 121
  5.3.1 Pre-conditions ............................................ 121
    5.3.1.1 Necessary Pre-conditions and Context ................. 122
    5.3.1.2 Chains of Pre-conditions ............................ 123
    5.3.1.3 Pre-condition Syntax .................................. 123
    5.3.1.4 Guarded Choices as Pre-conditions .................... 124
    5.3.1.5 Multiple Contexts ..................................... 125
    5.3.1.6 The Default Context .................................. 127
  5.3.2 Post-conditions ............................................ 128
    5.3.2.1 Satisfying a Pre-condition with a Post-condition .. 129
  5.3.3 The Pre-Condition Problem ................................ 130
  5.3.4 Tree Modifications ......................................... 131
    5.3.4.1 Searching for Hooks .................................. 132
    5.3.4.2 Tree Additions ......................................... 134
    5.3.4.3 Issues When Using Hooks and Tree Additions ...... 140
  5.3.5 Pre-computing Topoi Expansions ............................ 140
  5.3.6 Tagging Topoi .............................................. 143
    5.3.6.1 Some Additional Consequences of Tagging Topoi .... 145
  5.4 Controlling Expansion ......................................... 146
    5.4.1 Elaboration Attribute .................................... 146
    5.4.2 Context Selection ......................................... 147
    5.4.3 Depth of Expansion ...................................... 148
    5.4.4 Optional Pre-conditions ................................ 149
    5.4.5 Interactive Elaboration .................................. 151
  5.5 Discussion ...................................................... 151
    5.5.1 Differences from MEXICA ................................. 152
    5.5.2 Tagging Pre- and Post-conditions ....................... 153
    5.5.3 Elaboration is Expansion ................................ 154
    5.5.4 Continuity Conflicts ..................................... 154
    5.5.5 Narrative Speed .......................................... 155
    5.5.6 Creativity ................................................ 156
# Chapter 6: Interpolation

6.1 Interpolating Between Plot Points ................................. 157
   6.1.1 Active and Dead Links ........................................ 159
   6.1.2 Simple Interpolation ......................................... 160
   6.1.3 Satisfying a Dead Post-Condition Link ..................... 165
   6.1.4 Satisfying a Dead Pre-Condition and Post-Condition Link .. 166
6.2 Complex Interpolation .............................................. 168
   6.2.1 Multiple Link Satisfaction ................................... 168
      6.2.1.1 Satisfying Multiple Links with a Single Function .. 170
      6.2.1.2 Satisfying Multiple Links with Multiple Functions . 175
      6.2.1.3 Link Order ............................................. 176
   6.2.2 Long-Distance Link Satisfaction ............................. 177
6.3 An Example .......................................................... 179
6.4 Discussion ........................................................... 181
   6.4.1 Component Reuse ............................................... 181
   6.4.2 Insertion of a Story ........................................... 182
   6.4.3 No Backtracking ............................................... 182
   6.4.4 Impact on Content Creation .................................. 182
   6.4.5 Structure Creativity .......................................... 183
6.5 Summary ............................................................. 183

# Chapter 7: Continuity

7.1 Continuity Conflicts ............................................... 185
   7.1.1 Preventing Conflicts During Generation ..................... 187
      7.1.1.1 Underspecified Pre-conditions ......................... 188
      7.1.1.2 Runaway Elaboration .................................... 190
   7.2 Fixing Continuity Problems After Generation .................. 192
      7.2.1 Calculating Continuity ..................................... 192
      7.2.2 Using Interpolation to Fix Continuity Problems ........ 193
      7.2.3 Justification of a Thread-Based Approach ............... 197
   7.3 Discussion ......................................................... 197
      7.3.1 Irreconcilable Continuity Errors .......................... 199
      7.3.2 An Iterative Process ....................................... 199
   7.4 Summary ........................................................... 200

# Chapter 8: Conclusions and Recommendations for Future Work

8.1 Conclusions ......................................................... 201
8.2 Future Work ........................................................ 203
   8.2.1 Enhancing Tree Additions ................................... 203
   8.2.2 Pragmatics, Control and Reader Chronologies ............. 203
   8.2.3 Topoi Hierarchies .............................................. 205
   8.2.4 Story Difference ............................................. 206
   8.2.5 Elaboration and Interpolation in Other Systems ........... 207
List of Tables

2.1 A listing of the first three story grammar rules in a general story grammar, reproduced from Rumelhart (1977). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.1 A listing of the functions that occur in folktales, reproduced from Propp (1968). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
# List of Figures

2.1 A stratified description for the sentence “Caesar dies.” . . . . . . . 9
2.2 The field of natural language processing and the role of semantic expressions, reproduced from Levison et al. (forthcoming). . . . . . 10
2.3 An example of the conceptual dependency diagram for the sentence “Caesar walked home from the Senate”. . . . . . . . . . . . . . 14
2.4 An example of the DRS construction process for the sentences “Brutus stabs Caesar. He dies.” . . . . . . . . . . . . . . . . . . . . . . . 16
2.5 An example of an evidence rhetorical relationship, reproduced from Mann and Thompson (1988). . . . . . . . . . . . . . . . . . . 17
2.6 The general pipeline process used to generate text from semantics, adapted from Reiter (1994) and Stone et al. (2003). . . . . . . 29

3.1 The kidnap function, with labelled branches. . . . . . . . . . . . . 67
3.2 A tree depicting a function expansion. Here, one concept is replaced with another. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
3.3 A tree depicting function expansion. Within each node is a tree that represents the formal argument structure of each function. . . . 68
3.4 A hybrid tree depicting function expansion. . . . . . . . . . . . . . 69
3.5 The process of building a semantic tree. . . . . . . . . . . . . . . . 70
3.6 A simple semantic tree relating to a story. . . . . . . . . . . . . . . 71
3.7 A simple semantic tree relating to a story, provided without drawing the slist node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.8 A fragment of a simple story involving Brutus and Caesar. . . . . . 75
3.9 A simple semantic tree with a story thread. . . . . . . . . . . . . . . 76
3.10 A simple semantic tree with a story thread that starts at the end of the tale. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
3.11 A simple semantic tree with a story thread that uses only topoi. . . 78
3.12 An example of a semantic tree with a character thread. . . . . . . 80

4.1 The first move combination, reproduced from Propp (1968). . . . 88
4.2 The second move combination, reproduced from Propp (1968). . . 88
6.1 Part of a plot from a Proppian folktale. 158
6.2 The set of links that occur between the functions hero_reaction and provision_of_mobj. 160
6.3 The first step of interpolation, inserting an anonymous function x that may form an active link. 161
6.4 The final step of interpolation, insertion of a function which upholds the active link requirements of the surrounding functions. 163
6.5 The new links that become available after interpolation is complete. 164
6.6 The insertion of a function which satisfies the is_merchant(hero) post-condition of the function complete_task. 167
6.7 The insertion of a function to satisfy multiple links. 169
6.8 A trimmed semantic tree focusing on provision_of_mobj. 170
6.9 A trimmed semantic tree focusing on provision_of_mobj, with satisfied links to the function donor_get_object_and_travel. 175
6.10 A trimmed semantic tree focusing on provision_of_mobj. 175
6.11 A simple semantic tree, after the steal function has been inserted to satisfy one of the dead links. 176
6.12 A simple semantic tree, after the move_toward function has been inserted to satisfy one of the dead links. 177
6.13 A semantic tree attempting to satisfy the dead link possess(prince, money) stemming from the hero_buys_travel function. 178
6.14 An example of a semantic tree where a continuity conflict has occurred due to the failure of using the nearest neighbour when satisfying a dead link. 179

7.1 Part of a plot from a Proppian folktale containing a continuity conflict. 186
7.2 Part of a plot from a Proppian folktale, with a thread represented for the magic_sword. 193
7.3 Part of a plot from a Proppian folktale, with a thread represented for the magic_sword, and a function inserted to fix the continuity problem. 195
7.4 Part of a plot from a Proppian folktale, with a thread represented for the magic_sword, and a function inserted to fix the continuity problem. 196
7.5 Part of a plot from a Proppian folktale, with a thread represented for the magic_sword. 198
8.1 The topos villainy, which can be expanded into kidnap, pillage or steal.
Chapter 1

Introduction

The generation of semantic content is required in a variety of computational environments. For example, a computer may need to answer questions posed by a human user about the structure of a database (McKeown, 1985), and this may require that the content of the discussion and any response formulated be planned and then created by some computational process. In computer-based entertainment, works of interactive fiction in the form of video games have been and will continue to be an important industry. Games such as *Adventure* (Crowther and Woods, 1977) and *Zork* (Infocom, 1980) demonstrate the success of telling stories in a game, but their pre-defined content limits their replay value. Incorporating a device capable of producing new and unique plots in such games would increase their overall value. In computer assisted learning environments, content generation may be used to tell a story (Levison and Lessard, 2004), and when combined with diagnostic tools such as VINCI (Levison and Lessard, 1992), to diagnose the language errors a learner may make.

In the specific area of computer-generated stories, at least two different approaches for generating semantic content are found (Peinado and Gervás, 2006). The first relies on the creation of a virtual world, where the characters and their interactions are
simulated. Some example systems include TALE-SPIN (Meehan, 1976) and MINSTREL (Turner, 1992). The events in the story typically reflect the series of actions a character undertakes to achieve a goal. The second relies on the definition of a rigidly defined story structure produced by a story grammar, which builds the events of a story by performing a series of successive refinements on a set of story primitives, until a detailed story emerges. Examples include GESTER (Pemberton, 1989) and DEFACTO (Sgouros, 1999).

A significant limitation of both types of system is the lack of detail that is produced in regards to story events. Typically, these approaches produce only high-level descriptions, with focus on the manipulation of existing knowledge and content. Relatively little work is directed to providing strategies that can produce more detailed semantic statements.

This research further develops the story grammar approach. Currently, the rigidity required in this approach to produce well-formed and coherent stories results in the production of the same basic story structures over successive executions (Pérez y Pérez and Sharples, 2004). This repetition leads to predictability, which has a negative impact on the creativity of a story. This research is focused on developing methods to enrich semantic content, and to enhance the creativity of computer-generated folktales.

1.1 Objectives and Contributions

The purpose of this research is to explore mechanisms for enhancing the structure and content of computer-generated stories, and specifically, to develop strategies for content generation in the area of computer-generated folktales structured using a story grammar. Its three primary objectives are discussed below. While each of these objectives is considered in isolation, there is a certain amount of overlap and
interrelation among them.

1.1.1 Content Enrichment

The first objective is to develop a method to enrich the semantic content found in a story. Lack of detail can result in unsatisfying and simplistic stories. Methods are needed that are capable of producing appropriate amounts of detail where needed. However, the selection of detail must be appropriate in terms of context and quantity. For example, while a folktale could simply state that a witch kidnaps a princess, describing the circumstances and details involved in the event provides a more satisfying story for the reader on the condition that the detail ‘fits’ with the overall context, and does not distract the reader from the overall story line.

To achieve this objective, a process of elaboration is developed to enrich the set of semantic details that relate to a given story event. To achieve this, the functions that define story primitives are enhanced to contain pre-conditions and post-conditions. These provide a control mechanism to determine if an enhanced description is suitable given the surrounding context of the story. If the required pre-conditions do not exist in the story context, the elaboration mechanism can search a library of functions, select one that provides the necessary conditions, add it to the story, and then continue with the elaboration. Five different methods are proposed to control the extent of elaboration.

1.1.2 Structure Modification

The second objective is to develop a method to enhance the creativity of a rigidly defined story by modifying its structure. Variations in plot structure can add to the appeal of a story, and keep readers interested. Methods are needed to ensure that novel elements are incorporated into the structure of a story. For example, a series of folktales may state that a hero travels to the location of an evil witch in order
to rescue a princess, and that a fight ensues. Varying the structure of the story by adding the details of the journey between these two major plot points can result in a story that is more interesting and differs from the others in the series.

To achieve this objective, a process of interpolation is developed to reduce the predictability of a story grammar by modifying story structures. Rules based on pre- and post-conditions are developed to identify an appropriate set of expressions that may be inserted in between two successive story events. These rules ensure that the consistency of the story is upheld, while providing variations on the underlying structure.

1.1.3 Continuity Checking

The third objective is to develop a strategy to check for inconsistencies in the continuity of the semantic content that is generated. Continuity may be defined as the logical progression of events that explain how a change in state comes about. Errors in continuity result in inconsistencies in the story which may irritate readers and make story comprehension difficult. Consider for example a folktale, in which the hero marries a princess, but the princess is dead. In this simple example, there is an inconsistency which may confuse the reader. Methods need to be developed to check for and repair these types of inconsistencies.

To achieve this objective, a method is developed to check that automatically generated stories are consistent and free of continuity errors. To detect continuity errors, semantic threads are checked to ensure that state information such as location, ownership, and existence of each semantic entity is consistent and explainable within the story’s semantics. Errors or inconsistencies that are detected are fixed by providing interpolations between inconsistent expressions on the semantic tree.

While this research focuses on folktales due to their relative simplicity, the underlying concepts developed form the basis for further developments in a much broader
and complex set of applications.

1.2 Scope

While this work effectively describes how specific content enrichment and structure modifications may be accomplished through elaboration and interpolation, an equally important question is why we would want to use these processes. Answering this related question involves issues of rhetoric and pragmatics. Early work by Hovy (1987) suggests that in order for a generation system to achieve some pragmatic effect, it must be directed at some level by a set of rhetorical goals. For example, a pragmatic effect would be to make the listener feel more distant from the speaker, which could be accomplished by setting a rhetorical goal to make the tone of the resultant text more formal. Different rhetorical goals can have an impact on the type and amount of content that a text-planner produces. More recent work by DiMarco et al. (2008) examines the complexities involved in generating patient-specific information from pre-defined content. This type of tailoring involves aspects of pragmatics and style in the selection of appropriate content for the user, as well as during the “repair and polish” process to ensure the completed document is well formed.

In effect, further examining why we would use the elaboration and interpolation processes would provide insight into the structures and controls needed to tell us where and when to use these processes with respect to various pragmatic and rhetorical goals. While this work is compatible with the idea that a set of higher level processes and structures are responsible for directing elaboration and interpolation, a complete definition of them would require a much larger linguistically-oriented study. Additionally, more work would be required to determine whether elaboration and interpolation alone are sufficient to accomplish various rhetorical goals (see Chapter 8). An exploration of these topics and additional control mechanisms is beyond the
1.3 A Note on Natural Language Generation

Many of the semantic statements presented in this thesis have corresponding English instantiations as an example of the type of natural language output that could be expected if produced by a computer. It is important to note however, that the English output is produced by a human, rather than a natural language generation (NLG) system. While an automated system capable of producing the required English sentences from a semantic specification would have been preferable, the act of automatically generating natural language is far from being a trivial process. Despite many different NLG systems currently available (see Bateman and Zock (2008) for examples of different systems), there are still a large number of nuances inherent to natural languages that computational systems are simply not capable of faithfully reproducing at this point in time. The reader is reminded that the focus of this thesis is on the semantic expressions and the issues of elaboration, interpolation, and continuity, rather than the quality of the natural language statements presented in the examples.

1.4 Organization of the Thesis

Chapter 1 provides an introduction to the subject, the objectives, the contributions, the applications, and an overview of the thesis.

Chapter 2 contains a general introduction to the topics of semantic formalisms, creativity, and text planning systems.

Chapter 3 introduces the generation framework that is used as the underlying basis of this research.

Chapter 4 shows how a simple morphological analysis of traditional folktales can
be re-purposed for use in our generation framework, and the limitations inherent in that approach.

Chapter 5 introduces the elaboration mechanism, describes its use and advantages within a story grammar framework, and provides a detailed example of how it can be used to generate enhanced content.

Chapter 6 introduces the interpolation mechanism, describes how it is used to creatively extend stories that have been generated using a story grammar, and provides a generation example that makes use of it.

Chapter 7 discusses the importance of continuity, and introduces a mechanism capable of detecting whether continuity errors have occurred, as well as a method for correcting them.

Chapter 8 provides some concluding remarks and discusses some of the open problems and directions for future work.
Chapter 2

Background

Deciding what to say in a story, as well as how to say it, has been the subject of a great deal of work in literary analysis, logic, linguistics, philosophy and computer science to mention only these. Each field has made contributions to our understanding of natural language. Together they provide the basic building blocks from which a text-planning system can be constructed. The purpose of this chapter is to briefly explore several specific areas of natural language processing, semantics, and text planning with the goal of situating this particular thesis in relation to other research.

2.1 Layers in Languages

While many models of natural language have been developed, most of them share at least one commonality: stratification. Models are said to be stratified in the sense that there are many layers of representation that make up a complete linguistic description (Winograd, 1983). Each layer has its own, possibly unique, set of units and structural configurations that contribute a portion of information to a sentence or a text. Figure 2.1 demonstrates the layers of representation involved in the sentence “Caesar dies”.

As described by Winograd, the highest layer of representation, the semantic layer, encodes the meaning of the utterance. Individual units called sememes convey basic
Figure 2.1: A stratified description for the sentence “Caesar dies.”

concepts, and may be combined to create complex meaning. Depending on the semantic theory employed, sememes may have no internal structure, as in the case of the basic concepts captured in Conceptual Structures (Jackendoff, 1983), or they may have meaning only when interpreted relative to some world model, as in the predicates in Intensional Logic (Montague, 1974). Units at the semantic layer are then mapped onto the syntactic units of a given natural language. Two different units in the syntax layer, lexemes and morphemes, encode word forms, and encode various inflections and morphological agreements respectively. The final layer, the phonological layer, encodes auditory information using a series of phonemes, the primitive units that correspond to human speech.

Sentences themselves are involved in various hierarchical arrangements, and the semantic layer of representation may be responsible for its structuring. For example, the sentence “Caesar dies” may be embedded within a paragraph of text with the overall purpose of describing the exact nature of how the dictator’s death came to be. Similarly, this particular paragraph may be embedded in a larger text which tells the full story of Julius Caesar’s life. To capture these relations, a full descriptive theory of a natural language should include mechanisms for dealing with paragraphs and entire texts.

2.2 Semantics and Natural Language Processing

While each layer of representation is important to our theoretical understanding of natural languages, the focus of this work is purely on the semantic layer, and how
semantic layer entities can be transformed. The central role of semantics is depicted in Figure 2.2, which provides a general overview of some of the other mechanisms involved in natural language processing.

At the centre of this diagram lie language independent representations called *semantic expressions* (SEs) that capture the meaning of natural language utterances. These are constructed according to the rules of a *semantic formalism*. SEs may be written by humans, or may be automatically generated by some computational tool. A set of SEs may form the input to a natural language generator (NLG), a process that provides human language output in the form of spoken or written text. In turn, human speech or written text may be used as input by a natural language understander (NLU), which would produce a set of SEs that encode for various meanings. Other automated processes, in conjunction with generalized and domain-specific
knowledge bases, may perform tasks based upon the content of SEs. For example, a spoken-dialogue system is representative of the loop involving an NLU, NLG and various knowledge bases. A human may ask the system a series of questions in a natural language, which the computer would convert to a set of SEs using an NLU. An interpreter process would examine the content of the SEs, and depending on the content of the question, would consult a set of knowledge bases in order to formulate a response. The response would be encoded as a set of SEs that would be provided to an NLG, which would express the answer in some form of natural language.

It is important to note that the NLU, NLG and other semantic processes are non-trivial in nature, and much time and effort has been spent on the design and development of various natural language processing systems. Work done by Zock and Adorni (1996) attempts to categorize and describe various NLG systems, and the most up to date list by Bateman and Zock (2008) contains more than 200 systems or formalisms that have been developed to examine various aspects of natural languages. Clearly there are a wide variety of problems in natural language processing that continue to be actively pursued. The issues of NLG and NLU aside, the portion of the figure that is of interest in this research involves human or machine-created semantic expressions.

2.3 Semantic Formalisms

As noted above, the purpose of a semantic formalism is to clearly and unambiguously describe meaning in a language-independent way. Previous work by the author has examined a number of different semantic formalisms in detail (Thomas, 2009a,b). While an extended description of the inner workings of each of these formalisms is not necessary in this context, there are some important concepts which merit mentioning here. In turn, this examination will clarify where the semantic formalism used in this
work (see Chapter 3) is situated in relation to other formalisms.

2.3.1 Fundamental Concepts

Semantic formalisms can be broadly categorized based upon the approach they take to representing knowledge. Some formalisms make use of truth relative to a model of the world in order to express meaning. For example, as explained by Dowty (1981), Intensional Logic (Montague, 1974) defines a model $M$ which is made up of: $A$, a domain of entities, $W$, a set of possible worlds, $T$, a set of times, $<$, an ordering on the time domain, and $F$, a mapping function which assigns truth values to predicates that range over entities and other predicates, relative to both a possible world and a time. The end result is that, in Intensional Logic, predicates may have a different value for any given time and in any given world. For example, the sentence “Brutus is an honourable man” is captured by means of the following expression:

$$\neg\text{honourable}'(\text{brutus'}) \land \text{male}'(\text{brutus'})$$

The label brutus’ is drawn from the set {brutus’, caesar’, pompeia’}. Each is simply a label for a constant term, of which there may be an infinite number, and each corresponds to the real world identities Gaius Julius Caesar, Marcus Junius Brutus and Pompeia Sulla respectively. The predicate male’ is understood to represent a function $f_{\text{male}}$ that is known as the characteristic function of the male’ set. The function is responsible for mapping entities within the set, onto a value of 0 or 1. For example, the characteristic function would provide the following mapping: {brutus’ $\rightarrow$ 1, caesar’ $\rightarrow$ 1, pompeia’ $\rightarrow$ 0}. However, the characteristic function for the male’ set contains more than a single mapping for each constant. For every time inside the $T$ domain, paired with every world inside the $W$ domain, there exists a (possibly different) mapping. For example, within the honourable’ function, the mapping for brutus’ may be 1 at a time before Caesar’s murder, but may be a 0 afterwards. In
order to convey this fact, predicates and constants are usually evaluated relative to a time and world coordinate, as indicated by \([\text{male}('brutus')])_{M,w,t,g} = 1. Here, the \([\text{and}]) are used to indicate that we are looking at the value of some expression, relative to a particular set in which \(M\) represents the model in question, \(w\) is a world, \(t\) is a time, and \(g\) is an assignment of values to variables. If none of these are mentioned, then we assume the expression to be in the current world and at the current time. The \(\neg\) has this effect as well.

While the underlying model of Intensional Logic is different from that of the First Order Predicate Calculus, Montague introduced the same set of operators. Thus, value expressions may contain \(\land, \lor, \leftrightarrow, \rightarrow, \forall\) and \(\exists\), with their expected semantics. However, in addition, Montague introduces \(\hat{\text{F}}, \check{\text{F}}, \text{P}, =, \Box\) and \(\Diamond\), which are used in conjunction with the world and time coordinates to produce different meanings. For example, \(\text{F}\) (future) forces evaluation of the function to occur at times after the current time, while \(\Box\) examines a value relative to all possible worlds. The overall effect is to provide Intensional Logic with a degree of expressiveness greater than that of the First Order Predicate Calculus. However, this expressiveness comes at the cost of decreased user-friendliness.

Unlike truth-conditional representations such as those described above, formalisms such as Conceptual Structures (CS) (Jackendoff, 1983) and Conceptual Dependency (CD) (Schank, 1972) are based on the idea that there exists a number of conceptual primitives that may be arranged in different ways to create complex meaning, but without reference to some world within which their truth may be established. Primitives usually take the form of concepts that are typically assumed to correspond to basic units of knowledge that a human inherently understands. For example, in CD, Schank theorized that there were six different types of concepts: nominals called picture producers (PP),

\[\text{picture producers (PP)}^{1}\], modifiers of picture producers called picture aiders (PA),

---

\[^{1}\text{Schank (1972) provided the name } \text{picture producer for nominals since they tend “to produce a picture of that real world item in the mind of the hearer.”}\]
actions (ACT), modifiers of actions called action aiders (AA), times (T) and locations (LOC). While the PP, PA, AA and LOC classes are potentially open sets, the ACT and T set are closed. For example, Schank (1975) enumerates only eleven ACTs. Of these, PROPEL, MOVE, INGEST, EXPEL, GRASP, and SPEAK have the meanings we might expect, while PTRANS means to physically move something, MTRANS means to mentally transfer something, ATRANS means to change an abstract value, MBUILD means to build a mental representation, and ATTEND means to wilfully pay attention to something.

In the Conceptual Dependency model, concepts are drawn in various relationships on a graph called a conceptual dependency network. The placement of various primitives, in conjunction with their location on the network, represents meaning. For example, Figure 2.3 provides an example of the conceptual dependency network for the sentence “Caesar walked home from the Senate.”

Figure 2.3: An example of the conceptual dependency diagram for the sentence “Caesar walked home from the Senate”.

In this figure, the picture producers are Caesar, Senate and home. Starting at the top right hand corner, Caesar and home are involved in a one-way dependency. The POSS is a possessive modifier, meaning that Caesar possesses the home. This creates the meaning of “Caesar’s home.” The structure involving the Senate, home and Caesar is a directive relationship, which states that Caesar was involved in a directive ACT that originated at the Senate, and ended up at Caesar’s home. The PTRANS states that Caesar moved Caesar in the directive relationship. The o stands for object,
meaning that the object moved in the PTRANS was Caesar. The p stands for past.

The challenge with such conceptual theories is to find some set of primitives upon which complex conceptual definitions can be based. For example, Dunlop (1990) criticizes descriptive limitations of the primitive ACTs, arguing that a description of a social institution is simply not possible. Thus, while CD can describe the physical act of “the policeman gave me a parking ticket,” it cannot truly explain what the act signifies. While some researchers such as Wierzbicka (1996) have attempted to catalogue and categorize a universal set of primitives, ones that would be sufficient to cover all of these required aspects, the completeness of that work has yet to be empirically tested.

One of the major limitations of many formalisms, whether they be based on conceptual primitives or on model theory, is that most do not attempt to provide a structuring mechanism beyond the individual sentence. To fill this gap, a number of different mechanisms have been proposed. For example, Discourse Representation Theory (DRT) by Kamp (1981) is meant to extend the First Order Predicate Calculus model by adding special Discourse Representation Structures (DRS) that capture the dynamic nature of multi-sentence utterances. Each DRS contains a list of referents that exist in the world, along with the list of predicates that make use of those referents. As new sentences are encountered, a new DRS is created to capture the meaning of the utterance. The resultant DRS is merged with any existing ones, and the structures are then simplified\(^2\). For example, Figure 2.4 provides an example of the structures needed to represent the sentences “Brutus stabs Caesar. He dies.”

Figure 2.4(a) contains the first structure created to represent the sentence “Brutus stabs Caesar.” In this first structure, two variables, \(x\) and \(y\) are introduced as the discourse referents. These referents are then linked to the set of conditions which describe them. In this instance, \(x\) is identified by the predicate BRUTUS as being

\(^2\text{This process outlines the newer two-step version of DRT as described by Geurts and Beaver (2008). For the original one-step version, see Kamp (1981) and Kamp and Reyle (1993).}\)
Brutus. Similarly, $y$ is identified by the predicate \textsc{CAESAR} as being Caesar. The predicate \textsc{STAB} relates the fact that Brutus stabs Caesar. In Figure 2.4(b), the second sentence, “He dies” is encountered. The DRS that is created introduces the variable $u$ for the new referent. It is underlined to emphasize the fact that it is not yet related to a concrete entity in the model. The predicate \textsc{DIE} states that the referent $u$ dies. In Figure 2.4(c), the two structures are merged, and an identity is determined for $u$. In Figure 2.4(d), the structure is simplified by removing the extraneous reference to the variable $u$.

Rhetorical Structure Theory (RST) by Mann and Thompson (1988) is another example of a structuring mechanism. The purpose of RST is to provide an outline of the relationships that occur between various portions of non-overlapping text. These are rhetorical in nature, describing how an item termed the \textit{satellite} is related to the core item termed the \textit{nucleus}. The theory uses a set number of schemas along with rhetorical relationships in various configurations to describe the structure of the text. The result is a tree structure in which each portion of text is related to the entire story. While the exact number of relationships available in RST is as yet undetermined, a core set of rhetorical relationships such as \textit{justification}, \textit{background} and \textit{purpose} have been proposed and extensively described by Mann and Thompson. One of the examples they provide is that for an \textit{evidence} relationship, “intended to increase belief of the nuclear material.” (Mann and Thompson, 1988) The example
they provide involves the following clauses:

1. The program as published for calendar year 1980 really works.

2. In only a few minutes, I entered all the figures from my 1980 tax return

3. and got a result which agreed with my hand calculations to the penny.

A graphical depiction of the rhetorical relationship for the clauses is illustrated in Figure 2.5. In this example, the curved arrow indicates the direction of the evidence relationship, while the horizontal lines and their associated numbers indicate the clauses that are involved.

![Figure 2.5: An example of an evidence rhetorical relationship, reproduced from Mann and Thompson (1988).](image)

In conclusion, while many different semantic systems have been proposed, nearly all of them are lacking in mechanisms to deal with some type of linguistic phenomena. As well, usually a formalism must resort to the use of an additional structuring mechanism if it is to represent more than an isolated utterance. The model used in this work is non-truth functional, dealing with concepts similar to those proposed by Jackendoff. Additionally, it is capable of capturing textual structures. For reviews of other formalisms, see Thomas (2009a,b).

### 2.3.2 Required Attributes in a Semantic Formalism

As noted by Donald (2006) and Levison et al. (forthcoming) to be used in a computational environment, a semantic formalism must be practical to implement. One
aspect of practicality is *scalability* in the sense that the formalism must be capable of representing the smallest unit of meaning, and yet at the same time capable of representing sentences, paragraphs, and entire collections of related text while retaining a degree of *human readability*. This second point is important, since in a computational setting, a human being will eventually be required to verify the correctness of semantic expressions during the design and development of any automated process. Second, the formalism must have *wide linguistic coverage*. This means that it must be possible to represent a wide variety of linguistic phenomena in a clear and concise manner. Third, and finally, the formalism should be *language independent*, that is capable of capturing the underlying meaning of an utterance, without specific ties to the languages in which it is to be expressed.

2.4 Creativity and Content

Before we turn to the issue of text planning, it is necessary to introduce two separate, but equally important issues that influence how a text-planning system is structured.

2.4.1 Definitions of Creativity

The first issue concerns a debate centred on the notion of *creativity*. There are many different mechanisms that a text planner may use to structure a story, some of which are more successful than others in producing interesting texts. A brief exploration of the notion of creativity is required to understand why some text planners are structured in particular ways. A definition of creativity also allows for a simple and effective check to determine if the content-generation strategies presented in this research is capable of producing creative output.

As noted by Turing (1950), Boden (1990), and Bringsjord and Ferrucci (2000), the discussions surrounding whether a computer can be creative date as far back as
the 19th century to a paper by Menabrea (1843) on Charles Babbage’s Analytical Engine. In a series of translation notes accompanying the paper, Countess Lovelace makes the statement that “The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths.” In other words, the computer itself is not creative, and can only produce output that it has been specifically programmed to compute. This objection against computer creativity was noted and dismissed by Turing (1950)\(^3\), who found that computers had the ability to surprise the programmer on countless occasions, typically through errors in the program code. Other objections against computational creativity noted by Turing, range from the computer’s lack of a soul in the theological sense, to the notion of extra-sensory perception. In his paper, Turing systematically confronts and dismisses each argument, and also develops a test for basic creativity, in which an interrogator interviews two participants and attempts to determine which is a computer. The participants are hidden from view and are only allowed to respond via electronic methods, in order to remove any bias that may be present against a computer. Turing’s “Imitation Game” is one of the first examples of a mechanism that could be used to evaluate whether creativity was present in a computer program. Presumably, if the computer is creative enough in its responses, the interrogator would not be able to distinguish it from a genuine human.

Unfortunately, Turing’s test does not offer a concrete definition of what creativity actually entails. More recent work on computer creativity was conducted by Boden (1990) in an effort to establish a more formal definition. Following from Turing, Boden strongly believes that computers can at least be programmed to appear to be creative, regardless of whether or not they can in fact originate creative ideas. Boden’s work examines human descriptions of the creative process, and reveals how

\(^{3}\)Turing’s paper discussed *thinking* rather than creativity, however, he provides examples specifically related to literary creativity.
new ideas may be generated from a psychological standpoint. In her work, Boden defines two different types of creativity that humans typically exhibit: p-creativity and h-creativity. To be h-creative means to generate an idea or work that has never been generated before in all of history. (The h in h-creativity stands for historical.) To be p-creative on the other hand, is to only generate an idea or work that is new to the individual who has the thought. (Here, the p stands for psychological.) Boden also makes note of the fact that creativity is not simply the combination of pre-existing ideas, since there is usually some competence required to combine ideas. Competence means understanding the set of constraints, and entails knowledge of a particular field. This type of competence usually guides the creative process, at times in a completely unconscious manner, to produce new and unexpected results.

It is important to note that both Turing and Boden do not limit their definition of creativity to a specific artistic or scientific field. For example, Turing made note of the wide variety of questions that a computer would need to interpret and understand during the Imitation Game. These questions involve aspects of literature, science, mathematics and other areas that a human would have some experience with. Similarly, Boden explores the human creative process in areas such as chemistry, mathematics, fine art and literature. Unfortunately, it is beyond the scope of this thesis to argue whether or not a computer can be truly creative in all of these areas. Instead, we will limit our perspective to the field of literature, specifically, works of fiction. Additionally, following from Boden’s assumptions, an underlying premise of this work is that a computer can be made to appear to be creative with respect to the type of output that it can create. In other words, we use an output-centric definition of creativity.

Following from Boden’s work on creativity, P´erez y P´erez and Sharples (2004) define a new version of creativity called c-creativity, specifically with applications in the literary domain. Similar to p-creativity, the idea behind c-creativity is that a
computational process can exhibit c-creativity if it is capable of generating knowledge that has not been explicitly encoded in its knowledge base. (Here, the c stands for computational.) Pérez y Pérez and Sharples point out that there are two different types of knowledge that a content generation system usually contains: knowledge of the story structure and knowledge of the story content. A lack of creativity in either type of structure leads to predictability, which typically has a negative impact on interest. For example, if the structure of a story can be predicted from the input structures it is given, then the system is said to suffer from structure predictability. This problem is particularly pronounced in stories produced by a story grammar. Similarly, if the content of the story is known at the outset, and the system has no way of generating new content, then the system is said to suffer from content predictability. This difference in predictability types makes it difficult to compare systems, since any evaluation methodology must clarify what type of knowledge a given system is capable of creating.

This output-centric definition and evaluation of creativity is supported by more recent work. For example, Ritchie (2007) develops criteria that can be used for evaluating the creativity of a computational system. Ritchie’s definition of creativity is based on what we would consider creative if a human were to produce it. The notion of how to evaluate creativity is centred on what Ritchie calls “empirically observable factors.” Ritchie’s argument is that with human creativity, there are only a limited number of artifacts in the resultant output of a creative process that we can actually observe. Other aspects of creativity, such as the creator’s emotional state or mental processes, are not directly observable, and hence should not be included in an evaluation of creativity. This line of reasoning bluntly states that any measure of creativity in a computational system should likewise be based on the same artifacts, and should therefore ignore the inner construction of the process.

Within the framework Ritchie proposes, there are 18 different criteria that can be
used to measure the creativity of the output from some computational process. To establish these, Ritchie defines several key items involved in the creation of a work. The first is the basic items set, the set of elements that can be created by the program. For example, in a story, these basic items would be sequences of words. Using these basic items, Ritchie then proposes a mapping based upon three properties: novelty, typicality and quality. Novelty is a measure of how dissimilar a production of basic items is to other works within the given genre. Typicality is a measure of the extent to which a production of basic items is an example of a work within the given genre, based upon some rating scheme. Quality is a measure of how good the production is judged to be in the given genre. Novelty, typicality and quality are mappings from a set of basic items onto the interval $[0, 1]$. For example, a work that scores high for typicality is considered to be a good example of a work within that genre. Ritchie suggests two additional sets that can be formed from the basic items. These are the inspiring set, and result set. These two sets are useful, since some measures of creativity may be based on how dissimilar the result set is when compared to the inspiring set.

Given these definitions and mappings, Ritchie enumerates the criteria for the creativity of an output. Most of the criteria are calculations based upon the novelty, typicality and quality mappings. For example, the first criterion measures how close an output comes to being a typical example within the genre. This is done by taking the average typicality score of each result set that the system produces, and comparing it to some typicality threshold value that exists for the genre. Items that meet or exceed the threshold value are said to be typical members of the set. Similarly, the third criterion measures the average quality score, again comparing it to a quality threshold value. Other criteria take into account various aspects of creativity. For example, criterion seven examines the ratio of low typicality but high quality items within the result set, again measured with respect to a threshold value. It is generally
accepted that a creative output may be untypical within its genre, yet still be of high quality. Yet other criteria take into account measures of novelty. For example criterion eleven examines the value of items in the result set that are not part of the inspiring set. This measure roughly correlates to whether or not the newly introduced items were creative. These criteria, when taken as a whole, examine many different facets of creativity.

The main issue with applying Ritchie’s criteria is that there is no consensus as to whether or not they are the correct criteria to be applying, since a widely accepted definition of “creativity” is difficult to achieve. As Ritchie points out, having a computational process produce something that simply exists in a genre is quite different from having it produce high-value, but completely atypical works. Both tasks are worthwhile for different reasons, however, applying all of the criteria equally to both systems as a quantitative measure of creativity is counterproductive and unwarranted. Some of the criteria are in competition with one another, thus a high score for the output from two systems may be due to very different reasons. Finally, Ritchie admits that there is no known list of items that can be used to produce the rating schemes used to measure typicality and quality. Similarly, concrete threshold values for each of the criteria have not yet been established. While Ritchie suggests ways of establishing these values, a study achieving that has yet to be performed.

On a final note, as observed by Boden, creativity requires competence. For example, it is not enough to randomly strike the keys of a piano and call the resultant output creative. Knowledge of the structure and rules that are required to create music of a certain type is needed before the output can be called creative. However, competence combined with creativity can result in individuals who purposefully break the established structure and rules in an effort to fully explore some domain. When excessive or inspired, this exploration usually results in a paradigmatic shift within the creative area, leading to the formation of new genres.
2.4.2 Elements of a Good Story

The second issue pertains to defining the fundamental elements that are present in *good stories*. Researchers such as Meehan (1976) have suggested that those elements are: *consistency, cohesion, interest, style* and *complexity*. While a detailed examination of these elements is beyond the scope of this thesis, we assume that a text-planning system should adhere to at least some of these basic requirements.

Stories are not simply random collections of facts strung together in an arbitrary order. As seen in the previous section, there must be some guiding principles that are involved in producing a good story. One of the first researchers to attempt to define these elements in a computational setting was Meehan (1976). In his PhD dissertation, he defines five elements that he believes make a good story: *style, complexity, coherence, consistency, and interest*.

The *style* of a story refers to the set of rules that contribute to the structure of a plot. Different sets of rules result in different types of stories. Meehan’s examples include soap operas, episodic television shows, and detective stories. A soap opera is a tale in which solutions to problems cause other problems, ultimately creating a story that has no end. In contrast, episodic television usually focuses on problems and their solutions, in which the state of world affairs is returned to the status quo at the end of the episode. Detective stories on the other hand, usually refrain from telling the reader all the necessary details, such as the identity of the murderer, which are usually revealed at the end. Stories in each of these areas clearly use different ways of structuring their plots, providing a variety of different story styles.

The issue of *complexity* reflects the underlying model of the story world. As Meehan notes, simple models lead to short stories, while more complex models lead to longer stories. The types of information represented in the story model also have an impact on the overall quality of the story. Meehan’s TALE-SPIN worlds cover only the most basic forms of character and world interaction, sufficient to provide
stories of animals seeking to overcome issues such as hunger. More complex models that capture wider forms of interaction and general world knowledge would lead to longer stories.

Meehan defines \textit{coherence} as a story’s ability to maintain and explain causal relationships. Good stories maintain this type of coherence, whereas bad stories do not. To help illustrate his idea of coherence, Meehan provides an example of Joe the Bear attempting to satisfy his hunger by counting to thirty. Strictly speaking, this is not coherent with the way the world model works, in the sense that counting to thirty normally does not satisfy hunger. The story would be more coherent if Joe Bear ate some food instead. At a different level however, counting to thirty to satisfy hunger is coherent in the sense that the words and their ordering make sense to the reader, and are not just a random jumble. Meehan’s definition is meant to reflect the former type of coherence. In some instances, coherence can be tricky to maintain, since many different genres of fiction allow for different types of causal relationships. Meehan revives his example of Joe Bear counting to thirty to satisfy his hunger, which is completely coherent if it is part of a magic spell to satisfy hunger in a storybook world where magic is commonplace.

In addition to coherence, Meehan mentions \textit{consistency}. A good story must be consistent in its adherence to allowable causal relationships. This means that there should be no anomalies as to what is permissible in the world, and what is not. However, according to Meehan, some types of consistency anomalies can sometimes be a deciding factor in what makes a good story, as in the case of a man waking up as a giant cockroach in Kafka’s “Metamorphosis,” a story in which no explanation is given as to why this particular anomaly has occurred. As this example shows, some anomaly is allowable, even encouraged, if good stories are to be told. However, Meehan warns that without a theory as to \textit{why} anomalies make good stories, it is more likely that introducing them will produce the opposite effect.
Unfortunately, having coherence and consistency is not enough to make a story interesting. In fact, researchers such as Bringsjord and Ferrucci (2000) and Pérez y Pérez and Sharples (2004) have demonstrated that completely coherent and consistent stories can be quite boring, typically by providing examples of some of Meehan’s work (see Section 2.5.2.1 for an example). Therefore, in order for a story to be interesting, a reader must be able to relate to it in some way. However, further exploration of this topic by Meehan (1976) shows that interest in terms of relatedness is not a simple concept. At one level, a reader may be able to relate to a story about brushing teeth, since it is a task that everyone must engage in. However, this alone does not make for an interesting story, unless there are elements of a problem that need to be solved in some way. The problem may come in many different forms, but it is usually a difficult problem that cannot be solved easily. In general, the more complex the problem, the more interesting the story. However, while a problem may generate interest, providing too many of the low-level details in the solution can make the story boring.

2.4.3 Story Grammars

Rumelhart (1975) defines a story grammar as a means of specifying the internal structure of a story. Grammars of this sort may be used in story-comprehension tasks, and when properly defined, would be capable of distinguishing sets of sentences which form a story from those that do not. Rumelhart’s research represents some of the earliest work geared toward creating a generalized story grammar, one in which a wide variety of different story types would be accepted and produced. The story grammar makes use of a set of rewrite rules that have both a syntactic component, which specifies how story structures can be built, and a semantic component, which specifies how the built structures are to be interpreted. For example, some of the rules in the grammar are provided in Table 2.1.

The first syntactic rule states that a Story is composed of a Setting and an Episode.
Table 2.1: A listing of the first three story grammar rules in a general story grammar, reproduced from Rumelhart (1977).

The semantic interpretation of the rule makes use of the ALLOW relationship, which states that the Episode is made possible through the Setting. Rumelhart notes that the ALLOW relationship is not a CAUSE: the Setting of the story does not CAUSE the main story Episode to occur. The second syntactic rule states that a Setting is composed of any number of States, which are a set of propositions that describe the conditions of the story. The semantic interpretation of the rule basically states that the States of the story are conjoined. The third syntactic rule describes the Episode, which is composed of some Event in the story world, followed by the Reaction of the various characters. The semantic interpretation of the rule makes use of the INITIATE relationship, which Rumelhart describes as an Event that is followed by some wilful reaction of an entity to that Event.

Every rule in the story grammar that Rumelhart describes is specified in this way, with each syntactic rule having a corresponding semantic interpretation. The end result is that the story grammar can produce a valid structure with a valid semantic interpretation that corresponds to the structure produced. As noted by Andersen and Slator (1990), several different general story grammars emerged around the same time as Rumelhart’s grammar. Most were variants of the grammar, with modifications or improvements made to account for various story properties. Most however, retain the same type of context-free rule system.

Black and Wilensky (1979) propose several criticisms of story grammars from both a linguistic and a psychological perspective. As they note, Rumelhart’s grammar is

\[\text{#} \quad \text{Syntactic Rule} \quad \text{Semantic Rule}\]

<table>
<thead>
<tr>
<th>#</th>
<th>Syntactic Rule</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Story → Setting + Episode</td>
<td>ALLOW(Setting, Episode)</td>
</tr>
<tr>
<td>2</td>
<td>Setting → (States)*</td>
<td>AND(State, State, ...)</td>
</tr>
<tr>
<td>3</td>
<td>Episode → Event + Reaction</td>
<td>INITIATE(Event, Reaction)</td>
</tr>
</tbody>
</table>

4This thesis does not suggest that computational tools should emulate the way a human process would form stories, so the psychological arguments will not be presented here.
an example of a phrase-structure grammar, more specifically, a context-free grammar (CFG), as indicated by the structure of the rewrite rules. The main linguistic argument against a story grammar built in such a fashion is that CFGs cannot properly generate story constituents that are completely discontinuous. However, Mandler and Johnson (1980) responded to this criticism, questioning the validity of Black and Wilensky’s arguments. The examples of stories Black and Wilensky use in their criticism are either embedded within collections of stories, or were invented fragments. While these example stories do contain discontinuity, the question is whether or not discontinuity appears in real stories (i.e. works that are not separate collections put together in a volume). Mandler and Johnson contend that the stories appearing in their research do not have any form of discontinuity that is likely to be found in stories that follow the oral tradition (i.e. folktales and other fairy tales). Therefore, the issue of the expressiveness of CFGs does not apply.

Following Black and Wilensky, several other researchers criticized story grammars for other reasons. As Andersen and Slator (1990) explain in a historical overview of story grammars, many of these criticisms centre on whether or not such grammars could be used for story comprehension in the artificial intelligence (AI) community. They suggest that the use of the story grammar has died out within that community due mostly to these criticisms, despite clear and concise defences put forward by researchers such as Mandler and Johnson. While the use of story grammars in AI has diminished, the use of these grammars in the area of computational linguistics has continued in projects such as GESTER (Pemberton, 1989), DEFACTO (Sgouros, 1999), and many others.
2.5 Text Planning

As explained by McKeown (1985), text-planning systems typically have two problems that must be solved: the system must decide what information it wishes to convey, and how that information should be stated in a natural language. These two distinct phases are usually performed by separate strategic and tactical components respectively. In the strategic phase of content planning, the meaning of a text is assembled, and various structural configurations are used to organize it. The collection and assembly of meaning may involve creative processes, and is thought to operate solely in the semantic layer of language. The tactical component is responsible for creating the language specific linguistic representations needed to convey the story or dialogue to another human, and is believed to be independent of the knowledge domain. However, researchers such as Stone et al. (2003) have concluded that additional steps are required between the content planner and the realization phase in order to better match the semantic representation with a linguistic description. Nevertheless, the main processes concerning the strategic and tactical components are believed to be separable, allowing the study of one in the absence of the others. The process of planning a text has often been referred to as a pipeline, and contains a number of steps that most computational linguists agree must occur (Reiter, 1994). Figure 2.6 demonstrates the pipeline process from content selection to natural language realization that typically occurs in a text-planning system.

Content Planning $\rightarrow$ Micro Planning $\rightarrow$ Natural Language Realization

Figure 2.6: The general pipeline process used to generate text from semantics, adapted from Reiter (1994) and Stone et al. (2003).

As mentioned above, during the content-planning phase, semantic information is generated using some language-independent representation. This phase typically
makes use of a semantic formalism that either has wide coverage, or is applicable to a specific problem domain. All of the details and information surrounding the meaning of the text are either generated or gathered from various databases. During the *micro-planning* phase, content is refined and re-ordered in various ways. For example, specific lexical items may be selected, and micro-sequences of text may be rearranged to make a better fit with the thematic and communicative goals of the document (Hovy, 1987). Intermediate semantic structures may also be formed, to form a tighter fit with language dependent structures used by an NLG (Stede, 1996).

In the *natural language realization* phase, the modified content from the micro-planner is used by an NLG to generate natural-language output. It is the content-planning phase of the pipeline that is of interest in this research.

Different types of text planners are used for different purposes. For example, some systems exist solely to produce information within a very limited domain, such as describing museum exhibits, as in the ILEX system by O’Donnell *et al.* (2001). Other systems exist to answer questions about a particular domain. The TEXT system by McKeown (1985) is an example, where users were able to ask questions regarding the structure of a database. Yet other systems may be used to generate concise paraphrases of various semantic descriptions, as was demonstrated by Stede (1996). For the purposes of this research, we will limit our discussion of text planners to those that are meant to generate works of fiction in the form of folktales, since not all of the elements that go into telling a story may be desired in a different communicative domain. For example, while it may be advantageous to introduce elements of drama in a work of fiction, a text planner embedded in a diagnostic system would not be very well received should it produce elements of foreshadowing or tragedy during a question-answering session.
2.5.1 Classification of Systems

Text-planning systems may be classified based upon their method of story composition. According to Peinado and Gervás (2006), generation methodologies may be divided into transformational or structural mechanisms. Transformational mechanisms generate interesting stories through the simulated behaviour of characters motivated to achieve certain goals, as is the case of TALE-SPIN (Meehan, 1976). Other transformational systems such as MINSTREL (Turner, 1992) may have knowledge of goals the author wants to achieve, such as the dramatic use of foreshadowing and tragedy. The motivation for transformational systems is that the solution to a problem, be it finding something to eat or finding a novel way for a character to die, makes for an interesting story. The steps a problem solver takes to solve these problems typically become the main events that take place in the story.

Structural mechanisms on the other hand, use the previously discussed notion of story grammars to define the main outline of a text. As Pemberton (1989) explains, this particular approach to text generation assumes there is an overarching structure that exists for different genres of text. While portions of these systems may appear identical to the transformational approach taken in transformational systems such as TALE-SPIN (for example, modelling characters behaviours using goals), the main difference is that they make heavy use of known plot structures in order to formulate interesting stories. Thus while transformational systems use character goals and problem solving to develop novel story lines, structural systems rely on the creative generation of novel characters and the combination of different, simple stories within a well defined set of plot and dramatic devices to generate novel texts.
2.5.2 Transformational Systems

2.5.2.1 TALE-SPIN

TALE-SPIN by Meehan (1976), is an example of a transformational system. It was the first text planner to use a problem solver and a set of character-driven goals as a method of producing stories. The system utilizes Schank’s (1972) Conceptual Dependency framework as the semantic formalism to encode information about a simulated world that is populated by a set of characters, objects and locations provided by the user through an interactive process.

The process of generating a story in TALE-SPIN is essentially a problem solving exercise. The problem a character must solve is the fulfilment of a primitive need known as a sigma state, which may be one of sex, hunger, thirst or rest. Throughout the course of generation, characters interact with the environment using plans in order to achieve goals and sub-goals that eventually satisfy the sigma state that the story calls for. Characters may travel through the world using maps that contain various levels of detail, and may interact with other characters based upon a set number of relationship attributes such as trust, affection and familiarity. For example, Henry may be hungry, and may formulate various plans to obtain food to eat in order to satisfy his hunger. Along the way, part of a plan may be to ask Joe for directions to various food sources. Such an interaction would require that both characters be in proximity to one another, so Henry would formulate a plan to move himself closer to Joe. In order for such a journey to take place, Henry would need to consult a map to determine where he would have to travel. Once both characters are together, how the query is made between them is based upon their relationship status. If Henry and Joe are familiar and friendly with each other, the information may be passed without any problems. However, in many other cases, there may be some degree of persuasion required, or the query may not be made at all if the two hate each other.
A construct central to the operation of TALE-SPIN is that of the *plan*. As Meehan describes them, plans are procedures that are used to accomplish goals. At the centre of a plan are a series of *planboxes*, each composed of a set of *pre-conditions* that must exist in the world model, and a set of *actions* that are instantiated if the pre-conditions hold. For example, Meehan details the plan **DELTA-PROX**, the goal of which is for person X to get persons Y and Z together at the same location. The plan **DELTA-PROX** contains 8 different planboxes which are enacted in order until one of them succeeds. The first planbox, Planbox 0, is an explicit check to see if the goal has already been accomplished. In this example, if Y and Z were already together, no action is necessary. If Y and Z are not together, then the next planbox is consulted, where X tries to physically move Y to where Z is located. In this scenario, there are several pre-conditions, such as the fact that X must know where Z is, and must know the directions to Z’s location.

As Meehan explains, many of the original attempts at generating stories resulted in “mis-spun” tales due to missing actions or goals which could not be accomplished. However, repeated experimentation and revision of the system eventually led to fairy tales such as the example below.

Once upon a time, there was a dishonest fox named Henry who lived in a cave, and a vain and trusting crow named Joe who lived in an elm tree. Joe had gotten a piece of cheese and was holding it in his mouth. One day, Henry walked from his cave, across the meadow to the elm tree. He saw Joe Crow and the cheese and became hungry. He decided that he might get the cheese if Joe Crow spoke, so he told Joe that he liked his singing very much and wanted to hear him sing. Joe was very pleased with Henry and began to sing. The cheese fell out of his mouth, down to the ground. Henry picked up the cheese and told Joe Crow that he was stupid. Joe was angry and didn’t trust Henry anymore. Henry returned to his cave. (Meehan, 1976, p. 37)

While TALE-SPIN’s approach yields reasonable output, there are three different points to consider: the issues of creativity, interest, and the lack of semantic detail.
Researchers such as Pérez y Pérez and Sharples (2004) have noted that the stories told by TALE-SPIN are not interesting or creative. The following example from Pérez y Pérez and Sharples demonstrates these points.

John Bear is hungry, John Bear gets some berries, John Bear eats the berries, John Bear is not hungry anymore, the end.

The sample above is generated purely based upon character motivations, and does not take into consideration any storytelling devices, such as foreshadowing and climax, that an author may wish to touch upon to draw the reader to the story. Thus, while the story is coherent and consistent, there is no doubt that it is uninteresting. In terms of creativity, the story does not introduce any new knowledge that was not previously held in the system’s knowledge base. Similarly, the system does not attempt to produce new plans or goals that exist outside of the originally programmed specifications. These two issues lead to a lack of c-creativity in the resultant stories.

Turning now to the issue of semantic detail, it is apparent that the output from TALE-SPIN could benefit from two different types of semantic enrichment. In the first instance, it is possible to elaborate the descriptions of the basic actions themselves, possibly through the use of a knowledge base. For example, consider the sentence “He saw Joe Crow and the cheese and became hungry.” The simple act of seeing Joe Crow could contain a much more complete description of how Henry came to see him. There are many questions regarding the circumstances of the situation that, if answered, could reveal a number of interesting details. How was Joe perched in the tree? What were Joe’s physical features? How high up the tree was Joe sitting? What did the tree look like? If one takes into account these facts, a much more elaborate syntactic construction may be “craning his neck to look up, Henry saw the jet black feathery features of Joe Crow perched precariously on the topmost branch of a sadly drooping elm tree.” However, in order to formulate this description, a richer set of semantic information is required. As noted by Meehan (1981), the problem
has to do with the representation and computation of very detailed meaning: how much information should be created by each event, and how is all of the information organized within the system?

In the second instance of enrichment, it is possible to add detail in between the main points of the story using the process of interpolation, which, as we will see later, is one of the key points of this thesis. For example, consider the first two sentences of the story generated by TALE-SPIN: “Once upon a time, there was a dishonest fox named Henry who lived in a cave, and a vain and trusting crow named Joe who lived in an elm tree. Joe had gotten a piece of cheese and was holding it in his mouth.” The semantic state of the world differs significantly from the first sentence to the second. In particular, the second provides completely new information about the world without providing any kind of back-story as to how it was that Joe came to be holding a piece of cheese. By picking an imaginary point between the first and second sentence, we can insert a short portion of story that would join the two sentences together in a much more coherent fashion. Details of Joe’s cheese exploits may be revealed, such as where he got the cheese from and who he interacted with on his journey. This interpolation process could then be repeated for any two adjacent elements in the plot. The end result would be a much more detailed and creative account of the actions involved in a story.

2.5.2.2 MINSTREL

The MINSTREL system (Turner, 1992) is another example of a transformational implementation. While MINSTREL is similar to TALE-SPIN in that a problem-solver is employed to plan the text, there are several important differences. The system differentiates between character-level goals and author-level goals using different representations of knowledge known as schemas. Author schemas are concerned with the rhetorical nature of the story, and are employed in order to make the story interesting
for the reader. There are four different types of author level goals that MINSTREL implements, relating to theme, drama, presentation and consistency. Theme goals reflect the purpose of the story, and are captured in a series of planning advice themes (PATs). Each PAT relates to an adage such as “the early bird catches the worm,” and contains a structure that MINSTREL follows when creating character events to illustrate the theme. Suspense, tragedy, foreshadowing and characterization are the four drama goals that are used to enhance the quality of the story. Consistency goals focus on ensuring that the flow of events is believable and consistent with the story world, and presentation goals help to organize the final output for the reader by summarizing, omitting or ordering various events. MINSTREL uses combinations of these author-level goals to ensure that the resultant story is interesting to the reader.

Author schemas make use of character schemas, which are responsible for describing the events that occur in the story. MINSTREL addresses issues concerning creativity and uniqueness by making use of episodic memory and heuristic methods called Transform-Recall-Adapt Methods (TRAMs) when creating events that illustrate the theme. The idea behind the use of TRAMs is that the failure of a problem-solver to generate a required event typically requires a new and creative solution. When failures occur, past problems and their solutions recorded in memory can be analyzed and possibly transformed into novel solutions. Turner describes several different types of TRAMs that may be used, the selection of which depends on the type of problem to be solved. Some TRAMs are applicable only to certain classes of problems, while others may be more general in nature. Some TRAMs may also be combined to create a solution, and in cases where more than one set of TRAMs is applicable to the problem, a random selection is made from the list of candidates.

To understand how TRAMs work, consider Turner’s example of a story in which the event “a knight dies” must occur. For this particular example, assume that MINSTREL has no knowledge of previous death events to draw from, but that episodic
memory has the event “a knight is injured by falling from a horse.” This latter event can be transformed into a death event by using two different TRAMs, through the following process. The first TRAM called *TRAM:Similar-Outcomes* is employed to change the problem description of the death event into an event that has a similar outcome. In this case, sustaining an injury is recognized as having a similar outcome to a death event, since both have a negative impact on health. Thus the event “a knight dies” is transformed into “a knight is injured.” MINSTREL searches episodic memory for instances where a knight is injured and finds the example where a knight falls from his horse. The second TRAM called *TRAM:Similar-Outcomes-Partial-Change* is used to adapt the event to match the outcome that is required. In this example, the outcome is changed from an injury to death. This change is allowed, since a partial negative health event can be transformed into a complete negative health event. When the application of the two TRAMs is complete, MINSTREL would have the event “a knight dies by falling from a horse.”

An example of a story generated by MINSTREL appears below.

Once upon a time there was a Lady of the Court named Jennifer. Jennifer loved a knight named Grunfeld. Grunfeld loved Jennifer. Jennifer wanted revenge on a lady of the court named Darlene because she had the berries which she picked in the woods and Jennifer wanted to have the berries. Jennifer wanted to scare Darlene. Jennifer wanted a dragon to move toward Darlene so that Darlene believed it would eat her. Jennifer wanted to appear to be a dragon so that a dragon would move toward Darlene. Jennifer drank a magic potion. Jennifer transformed into a dragon. A dragon moved toward Darlene. A dragon was near Darlene. Grunfeld wanted to impress the king. Grunfeld wanted to move toward the woods so that he could fight a dragon. Grunfeld moved toward the woods. Grunfeld was near the woods. Grunfeld fought a dragon. The dragon died. The dragon was Jennifer. Jennifer wanted to live. Jennifer tried to drink a magic potion but failed. Grunfeld was filled with grief. Jennifer was buried in the woods. Grunfeld became a hermit. MORAL: Deception is a weapon difficult to aim. (Turner, 1992, p. 8)

Despite having new mechanisms to create and organize content, there are several
shortcomings in the system. As Turner himself points out, plans to achieve author-level goals are captured as LISP objects. Unfortunately, these plans are responsible for encoding information relating to story structure, and since MINSTREL has no knowledge of LISP, it is unable to adapt these plans to create new story structures. Thus, while MINSTREL’s use of TRAMs does contribute to c-creativity in the sense that it is capable of producing new knowledge, the number and nature of MINSTREL’s story structures remains fixed. In relation to c-creativity, Pérez y Pérez and Sharples (2004) go on to criticize the nature of some of the TRAMs that Turner employs. They believe that some of the TRAMs are too tailored to solve a specific problem, and feel that they may become unproductive in different circumstances. The example Pérez y Pérez and Sharples use is *TRAM:Similar-Outcomes-Partial-Change* with a knight sewing his socks. Consider if the knight were to prick himself, causing an injury, and that MINSTREL finds that an injury is similar to being killed. The conclusion that MINSTREL may reach is that sewing is a way to kill someone.

Finally, turning to the underlying detail of the stories that MINSTREL produces, the same issues surrounding the need for semantic enrichment are apparent. For example, consider the sentence “Jennifer drank a magic potion.” There is no mention of how Jennifer came to possess a magic potion with the ability to turn the one who drinks it into a dragon. Interpolation between this sentence and the one before it would provide us these details. Additionally, there is a lack of description surrounding the Jennifer-to-dragon transformation process. Again, elaboration is necessary in order to produce these details.

### 2.5.2.3 MEXICA

The MEXICA system (Pérez y Pérez, 1999; Pérez y Pérez and Sharples, 2001) is a different type of transformational system, meant to reproduce the cyclic pattern of *engagement* and *reflection* that a human uses to write stories. In MEXICA, sequences
of primitive story elements known as *story-actions* form stories. Each story-action has a set of pre-conditions and post-conditions that dictate what emotional links and tensions must occur, and a series of texts that are used to realize the action (one text is selected at random during realization if more than one text is supplied for the action). During generation, story-actions are chosen according to various contextual and rhetorical constraints. Contextual constraints occur through the use of *story-world contexts* associated with the characters in a story, and represent the relationships that a character has with other objects in the world. MEXICA models three types of relationships: emotional links, tension, and location. Rhetorical constraints work to shape the set of next possible actions based upon a set of guidelines obtained through reflection. For example, a valid rhetorical constraint would be to select actions that result in heightened tension.

The production of a story begins with the engagement stage and some initial set of story-world contexts. Given this initial situation, a story action is selected by the system. For example, if two characters are in love with the same person, then an action that builds tension may result. In this way, the set of actions that the system may choose from is constrained by the story-world contexts currently in effect. The set of actions is also filtered so that it only contains actions that meet certain rhetorical guidelines set by the reflection stage. Each time an action in the story occurs, the story-world contexts for each character is updated, and a new set of possible actions are chosen and filtered. The system chooses one of the next possible actions at random from the pool, and the process repeats. Engagement ends when no further actions are possible.

During reflection, the coherence of a story is verified by checking the pre- and post-conditions of an action. If the story-world contexts do not support the requirements of the action, then an appropriate action is chosen to satisfy them, and the new action is inserted into the story. Guidelines are set based upon a collection of previous stories.
Essentially, MEXICA can check the current story against a set of previously generated stories, and create filters that will guide the selection of a next story action in a novel direction, or in directions that create tension. Reflection ends when coherence is validated and guidelines are set. The system may then enter another engagement state. A portion of a story generated using MEXICA is provided below. The example illustrates the emotional relationships involved between characters.

Jaguar_Knight was an inhabitant of the Great Tenochtitlan. Princess was an inhabitant of The Great Tenochtitlan. Tlaloc – the god of the rain – was angry and sent a storm. The heavy rain damaged the old wooden bridge. When Jaguar_Knight tried to cross the river the bridge collapsed injuring badly Jaguar_Knight’s head. Princess knew that Jaguar_Knight could die and that Princess had to do something about it. Princess had heard that the Tepescohuitl was an effective curative plant. So, Princess prepared a plasma and applied it to Jaguar_Knight’s wounds. It worked and Jaguar_Knight started to recuperate! Jaguar_Knight realised that Princess’s determination had saved Jaguar_Knight’s life. (Pérez y Pérez, 1999, p. 103)

This approach differs from other transformational systems in that MEXICA functions without using explicitly coded goal states, such as those dictated by TALE-SPIN. In other words, rather than have a character achieve a goal, the system attempts to produce actions that result in increases and decreases in tension, with an overall model to build tension to a climax, and then gradually diminish it. Additionally, MEXICA keeps a collection of previous stories in memory, and by doing so, can perform a direct comparison to ensure that the story being produced is novel when compared to other instances it has on record.

There are some limitations to this approach. First, the syntactic structures produced by the system are templates, with various character names filled in along the way. Thus, while the functions can be directly modified to create rich descriptions, the resultant output is highly language dependent, and cannot be modified during generation to scale its description of an action. Second, the system does not model
the meaning of a story action beyond stating what emotional impacts the action has on characters, and the type of tension that results. While this is sufficient to form a chain of actions from one instant in time to the next, it limits the applicability of the approach to other genres.

### 2.5.3 Structural Systems

#### 2.5.3.1 DEFACTO

The DEFACTO system by Sgouros (1999) is an example of a structural system, although it uses aspects of transformational systems such as character goals and simulation. In DEFACTO, the user assumes the role of the protagonist, and interacts with other characters within a story. Each of these entities is generated automatically, and is assigned goals. Through the use of a central plot generator, the DEFACTO system creates actions for the various actors based on their goals in a generation phase. Then, the system evaluates the list of possible character actions to determine if they would fit well within an Aristotelian type plot, eventually reaching a resolution phase where all the plot threads are resolved. In this way, DEFACTO merges simulated characters with an overall plot structure, thus providing a measure of coherence to a simulated world, while maintaining a dramatic flow of events. The main purpose of DEFACTO is to produce interactive, dynamic stories that can be incorporated into games, or can stand alone as examples of interactively generated stories.

As Sgouros explains, the DEFACTO system makes use of Aristotle’s theory of how theatrical dramas are structured to drive generation. Loosely speaking, the structure of such plays is as follows: an initial situation builds into a series of character conflicts of action and counter-action, culminating in some form of climax, whereby the main plot is unambiguously resolved. This type of structure can be seen as a simple story grammar. The DEFACTO system adheres to this general plot flow by making an
evaluation at each stage of generation to determine whether or not a character action would support the current stage of the plot. For example, the evaluation portion of the plot generator will determine whether various character behaviours have some form of dramatic value such as *irony, rising complications, reversal of fortunes* or an impact on a *lifeline*. Using these items as a guide, along with user-initiated behaviour, the system decides which dramatic actions fit in with the next stage of the plot development. Additionally, each action is checked to see if it fits in with the overall *storyline goal* - the character goal that caused the initial conflict. The system only considers those actions which support the further development of the storyline goal within the framework of the general plot structure. Actions which do not support the flow are simply not developed. The system also detects character goals that are in conflict with one another, and prevents them from occurring to avert an infinite set of action and counter-action exchanges.

The result of all of these mechanisms is a series of character actions that results in a simple story line. An example of a story generated using the DEFACTO system is given below.

The protagonist enters Corinthos and seeks to worship Poseidon. For this reason, s/he decides to perform a sacrifice to Poseidon. Eumeneas, the king, is an enemy of the user, therefore he seeks to block the user from sacrificing to the gods. Eumeneas decides to forbid the protagonist to perform this sacrifice. The protagonist reacts and seeks to confront Eumeneas. This behavior violates the law that demands obedience to the royal orders. As a result, Dikosthenis, the judge, seeks to punish the protagonist. In reaction to Dikosthenis the user tries to hide. Anacleoussa, the priestess, notices that when Eumeneas forbids the user from worshipping Poseidon he opposes the religious law that allows the mortals to freely worship the gods. As a result, Anacleoussa seeks the help of Poseidon to stop the king. In reaction, the king seeks to send the priest into exile. (Sgouros, 1999, p. 50)

Clearly the same arguments discussed in the previous section on transformational systems apply to the output generated by DEFACTO. Specifically, there is room for
the creation of more elaborate descriptions of some of the events that are occurring in the story. For example, the descriptions of various confrontations, the prescribed punishment, and the help that Poseidon could provide Anacleoussa are all aspects that could be improved. This system, like the others previously discussed, provides only the most basic amounts of information to the user. In terms of creativity, the system does not appear to suffer from the type of structure predictability that many story grammars fall victim to. However, this assumes that the user does not repeat the same set of actions on each generation attempt. Given the same initial conditions, and the same sets of actions, it is possible to generate the same story.

2.5.3.2 GESTER

The GESTER system (Pemberton, 1989), which generates stories of the Epic genre utilizing a story grammar, introduces two novel elements. The first is the distinction of several layers of organization in a given text, in this case a medieval French epic novel. Pemberton theorized that there are four levels of expression that ultimately contribute to a story: narrative structure, story line, discourse and textual representation. At the highest, most abstract level there is the narrative structure, where functions and roles describe the structure of the story without any details about the actual content. For example, at the narrative level, a function might be “cause”, and a role may be “subject”. Here we could imagine a subject causing an event. The narrative structure is then instantiated with actors and events that fulfil the demands of the abstract functions. Thus, at the level of the story line, actors such as “Brutus”, “Caesar” and “Brutus stabbed Caesar” would become the main form of information, and would replace the abstract functions that the narrative structure called for. The discourse level then selectively chooses information from the story line that should become the actual story. Depending on the genre of story, some information may be left unexpressed, while other information may be explicitly mentioned. Pemberton
describes the difference between the story line and discourse through the use of the
detective-story genre. While the story line may start with information about a crime
that was committed, the discourse might instead start with the discovery of the crime,
and may selectively leave out portions of the story line purposefully to be revealed at
the end of the novel. Finally, the textual level is the realized natural-language output
generated from the discourse level.

The second novel element introduced by Pemberton is a distinction between a
very general story grammar and the genre-specific information that would be needed
to tell a story that conforms to a particular style. This separation allows the system
to be deployed in a series of separable knowledge modules. Thus creating a story in a
different genre, say that of folktales, would only involve encoding folktale elements into
a suitable knowledge base. Additionally, the model could be extended to incorporate
other sources of information, such as knowledge of the audience, knowledge of the
author and cultural cues. Each of these sources of knowledge would have a distinct
impact on the generated output.

While the GESTER system was unable to map from the discourse level to the
textual output, the rest of the model described above was implemented. An example
of output from the GESTER system is provided below.

Charles lacked a city. As a result of hearing of Narbonne, Charles wanted
Narbonne. Then Aymeri agreed to help Charles. Then Charles and
Aymeri rode to Narbonne. Then, Charles attacked the walls of Nar-onne, currently controlled by Baufumez, helped by Aymeri. Thibaut
and Clarion threw burning pitch down on Charles and Aymeri. Charles
and Aymeri retreated. Then, Charles attacked the walls of Narbonne, cur-
rently controlled by Baufumez, helped by Aymeri. Thibaut and Clarion
threw stones down on Charles and Aymeri. Charles and Aymeri broke into
Narbonne. As a result of seeing Blancheflor Charles wanted Blancheflor.
Charles succeeded in getting Narbonne. Charles praised God. Charles for-
got to reward Aymeri. Charles threw Thibaut into prison. Then Charles
planned to obtain Blancheflor for Charles. Then Aymeri refused to help
Charles because he was not rewarded. Then Bertrand agreed to help
Charles. Charles abducted Blancheflor, currently controlled by Thibaut
helped by Bertrand. Because Thibaut was in prison he did not oppose
Charles and Bertrand. Clarion opposed Charles and Bertrand in getting
Blancheflor. Charles succeeded in getting Blancheflor. Charles praised
God. Charles rewarded Bertrand. (Pemberton, 1989, p. 223)

Several shortcomings of the system were mentioned by Pemberton, notably the lack
of greater detail and enriched statements, as well as a restriction to a single genre.
However, Pemberton introduced two different narrative motifs to the story structure
to allow for enrichments: tied narrative motifs and free narrative motifs. Tied narrat-
ive motifs allow for a step by step breakdown of a single action into multiple actions.
Free narrative motifs on the other hand, are free in the sense that they may be allowed
to enumerate various forms of background information, such as exploring historical
settings or illustrating character traits.

### 2.5.3.3 STORYBOOK and AUTHOR

The STORYBOOK system by Callaway and Lester (2002) is another example of a
structural system based on a story grammar. However, while the plot-generation
mechanism it uses is simple, Callaway and Lester’s research is important for several
other reasons. First, they argue that the sequence of events as told to the reader
does not necessarily follow the same sequence of semantic content created during the
content-planning phase of generation. To distinguish between the two, they borrow
two terms from Russian formalists: fabula and suzjet. The fabula of a story refers
to the total semantic content that is available at the semantic layer of the story.
For example, the basic semantic statements such as “Caesar stabbed Brutus,” and
“Caesar was a dictator” are contained within the fabula. The suzjet refers to some
linear order that is ultimately presented to the reader. Not all of the fabula may
be referenced by the suzjet, and the exact ordering dictated by the suzjet may jump

---

5The punctuation provided in the quotation is that generated by the system. Additionally, while Pemberton provides the statements that are deleted from the narrative, they are not supplied here.
backwards and forwards through the set of story events. This distinction is important, since many systems provide a narrative stream that directly reflects the order in which content was generated.

Second, Callaway and Lester approach story generation from a holistic point of view, noting that start-to-finish generation was lacking in many other systems. Their system attempts to provide a polished natural language output in the form of a story, rather than semantic expressions or hand-crafted linguistic examples that relate to the semantic content generated by other systems. In order to provide this type of output, Callaway and Lester created a modular system that worked as a pipeline. A narrative planner would take the initial requirements of a story, such as the characters, plot and other parameters, and create a set of fabula and a narrative stream. These would subsequently become inputs to a narrative organizer, that would segment the expressions into sentences and paragraphs, scan the discourse history for areas where pronouns or definite/indefinite articles might be applied, and make choices of how specific lexical items should be instantiated from semantic expressions based on usage histories to avoid repetitiveness and monotony. The output of the narrative organizer would then become input to a sentence planner and revision component. The sentence planner would transform the semantic structures into functional descriptions (FDs). As Callaway and Lester explain, FDs are a hybrid structure, maintaining some form of semantic information while specifying necessary linguistic information such as grammatical subject, object, gender and number. From here, the FDs could, in principle, be sent directly to a natural-language realization system to produce natural-language output. However, Callaway and Lester suggest instead sending them to an additional revision component, the purpose of which is to reorder and simplify the FD before sending it to the realization system. For example, multiple sentences may be aggregated into a single sentence. The final step in the process is natural-language realization, where the FDs from the revision component are transformed into natural
language.

Third, Callaway and Lester were able to empirically test the effects of portions of the system. Of the subsystems described above, the discourse-history analyzer and lexical chooser in the narrative organizer, and the revision subsystem are classified as *optional* components. Thus, their use in the system may be turned on or off. This is important, since it allowed Callaway and Lester to directly test their impact on the output generated by the system. To perform the evaluation, Callaway and Lester created the fabula relating to two individual stories, both variants of Little Red Riding Hood, and used them for each run of the system. Although there were eight possible comparisons to be made, only five variants were tested: all components on, revision off, lexical choice off, discourse history off, and all optional components off. The evaluation spanned many facets of the text ranging from grammatical acceptability, to style, to overall story believability. The results showed that generation with discourse history and all optional components turned off had the lowest scores. Interestingly, the highest scores were attributed to runs where all the optional components were on, or where the lexical choice was turned off. Callaway and Lester suggest two possible reasons for this result, the first being that the lexical variation created by the lexical choice subsystem was judged to be too difficult for a young reader to understand, the second being that the lexical choice system itself lacked sophistication that would have resulted in a higher score. Regardless of these reasons, they believe the results from the study provide sufficient proof that the structure of the system is warranted. An example of a portion of a story generated by STORYBOOK is provided below.

Once upon a time, there was a woodman and his wife who lived in a pretty cottage on the borders of a great forest. They had one little daughter, a sweet child, who was a favorite with everyone. She was the joy of her mother’s heart, and to please her, the good woman made her a little scarlet cloak and hood. She looked so pretty in it that everyone called her Little Red Riding Hood. (Callaway and Lester, 2002, p. 247)
While the output is much improved compared to previous systems, it is unclear as to how much detail is being produced by the underlying narrative planner. That being said, it is clear that more complex stories may be possible if more semantic detail is added to the underlying representations.

2.6 Summary

This chapter provided the background information and an overview of early work on text-generation systems. The theoretical concepts of language stratification and semantic formalisms were discussed. The characteristics of a good story and an output-centric definition of creativity (c-creativity) were defined. The concept of a story grammar and story primitives were introduced. The relative merits and limitations of generating story content using transformational text-planning systems such as TALE-SPIN (Meehan, 1976) and MINSTREL (Turner, 1992), and structural text-planning systems such as GESTER (Pemberton, 1989), DEFACTO (Sgouros, 1999) and STORYBOOK (Callaway and Lester, 2002) were reviewed.
Chapter 3

The Generation Framework

A semantic formalism that has wide linguistic coverage, is precise, is human readable, and is scalable, is needed to generate large, creative stories. While other researchers have made use of formalisms such as Conceptual Dependency, First Order Predicate Calculus, and other more exotic representations, we have shown in Chapter 2 and in other work, that each of these formalisms fails to fulfil one or more of our fundamental criteria. More importantly, no previously defined formalism has demonstrated the capability of dealing with the structure of text beyond the sentence level, without incorporating some other structuring mechanism such as Rhetorical Structure Theory (Mann and Thompson, 1988).

This thesis makes use of a formalism created by Levison and Lessard (2004), extended and explored by Donald (2006), practically demonstrated by Thomas (2008), and formalized in Levison et al. (forthcoming). This representation meets all of our criteria, with the added benefit of being able to deal with linguistic structures as small as a single lexical item and as large as a multi-paragraph text. The purpose of this chapter is to describe how this semantic formalism works, as well as to provide examples of how content may be generated in a system that makes use of it.
3.1 Semantic Expressions

Meaning is captured by a series of semantic expressions that take the form of a *function application*. Each function application returns a basic *meaning*. In this research, a meaning is similar to the idea of the concept used in Jackendoff’s (1983) Conceptual Structures and Schank’s (1972) Conceptual Dependency. For the sake of simplicity, very simple English words will be used to represent meanings. The various structural combinations of function applications allow complex meaning to be represented. Additionally, each function application may accept zero or more argument parameters which are themselves function applications. The design of these function applications is meant to be applied as a type of *semantic programming language*. Its design is inspired by features similar to those found in functional programming languages such as Haskell (Bird, 1998). It is important to note that the list of topics that follows is meant only to serve as a general introduction to the framework, and is not meant to be taken as an exhaustive examination of its capabilities and coverage.

3.1.1 Basic Functions

Each function application is essentially a *semantic function* which has a *name*, a *semantic type*, a set of zero or more *formal parameters*, and a *meaning*. A function name is a simple label that is meant to identify the function. Names carry no special significance, and are simply combinations of letters and digits that form useful mnemonics for the programmer\(^1\). Five different semantic types are defined: *completion, entity, action, qualifier* and *circumstance*. Completions are meant to reflect fully formed thoughts, and are roughly analogous to the idea of a sentence. Entities and actions represent real world objects and events that may occur. Qualifiers are meant to provide additional details surrounding entities, and circumstances provide details

---

\(^1\)The term *programmer* is meant to refer to the individual who will specify the set of semantic expressions that will be used in the generation system.
regarding actions. The formal parameters for a function are listed in the function’s 
*type signature*, and correspond to specific semantic types. Any function passed as a 
formal parameter must have a type that matches the one listed in the type signature. 
Finally, each function has a basic meaning associated with it that is expressed in 
plain English. Each function may also contain other information that will be defined 
throughout the course of this thesis. The combination of all of these elements is 
known as a *function declaration*. Groups of declarations are usually contained within 
a semantic *lexicon*, which contains collections of related functions.

### 3.1.2 Entities

The practical use of these concepts can be best demonstrated with some simple exam-
iples. The listings that follow use a set of conventions that are based on a combination 
of the syntax found in the Haskell programming language, and partly by the authors 
of Levison *et al.* (forthcoming). In the latter part of this chapter, and throughout the 
remainder of this thesis, the conventions are further extended to allow for the illus-
tration of more complex concepts. Consider the definition of the individual Marcus 
Junius Brutus, provided in Listing 3.1.

---

**Listing 3.1** The definition of a zero-valent function `brutus`, which is meant to re-
present the entity Marcus Junius Brutus.

```haskell
brutus :: entity
brutus
"Marcus Junius Brutus"
```

---

The first line of the listing provides the type signature of the function. The signature 
declares that the function named `brutus` returns a semantic type of `entity`. Had the 
function allowed for any formal parameters, they would have been specified on this 
line as well. The second line represents the *composition* of the function, and allows 
for a series of *semantic transformations* in the form of expansions to take place (see
Section 3.3 below). Finally, the third line is where we express the meaning of the semantic function informally by means of a set of simple English statements.

3.1.3 Actions

Turning to more complex examples, consider the instance where we would like to create a function called stab, that combines two individuals in a relation where the first stabs the second. This definition is provided in Listing 3.2.

Listing 3.2 The definition of the functions stab, brutus and caesar.

```
stab :: (entity, entity) -> action
stab(villain, victim)
"villain stabs victim"

brutus :: entity
brutus
"Marcus Junius Brutus"

caesar :: entity
caesar
"Gaius Julius Caesar"
```

In this definition, the function name stab is defined to take two parameters, both entities. The second line provides labels for each of the formal parameter types. In this example, the first entity will be labelled villain, and the second will be labelled victim. These labels carry no semantic value, and are used so that the programmer may associate a formal parameter with an identifier for use during expansion and to calculate semantic meaning. For example, if we were to supply the zero-valent functions brutus and caesar as formal parameters in the expression stab(brutus, caesar), then the identifier villain would be replaced by brutus, and victim by caesar. The meaning after evaluation of the function may be expressed using informal English as “Marcus Junius Brutus stabs Gaius Julius Caesar.”
3.1.4 Qualifiers and Circumstances

The previous examples demonstrated the definitions of simple entities and actions. Both entities and actions can be modified so that they have particular properties associated with them. For this to work, we require the use of a function called \texttt{qual}, which applies a qualifier to an entity and returns a qualified entity. The definition of \texttt{qual} is provided in Listing 3.3.

\begin{quote}
\textbf{Listing 3.3} The definition of the entity modifying function \texttt{qual}.
\end{quote}

\begin{verbatim}
qual :: (entity, qualifier) -> entity
qual(object, desc)
"desc object"

knife :: entity
knife
"a knife"

blue :: qualifier
blue
"blue"
\end{verbatim}

We are now able to formulate expressions such as \texttt{qual(knife, blue)}. Given suitable declarations for \texttt{knife} and \texttt{blue}, we would have “a blue knife.” The function could also be applied repetitively, resulting in expressions such as \texttt{qual(qual(knife, blue), sharp)}, and its corresponding English realization would be “a sharp blue knife.”

Circumstances can be used to modify actions. A new function called \texttt{circ} takes an action and a circumstance, and returns an action. The definition of \texttt{circ} is provided in Listing 3.4. In this example, the previously defined \texttt{stab} function is used to create the expression \texttt{circ(stab(brutus, caesar), vicious)}, which would become “Brutus viciously stabs Caesar.”

This definition for \texttt{stab} is not the only way we could have designed our functions. We could have created a function called \texttt{transitive_action}, and parameterized the action. This alternate definition is provided in Listing 3.5.
**Listing 3.4** The definition of the action modifying function \texttt{circ}.

\begin{verbatim}
circ :: (action, circumstance) -> action
circ(act, desc)
"act is performed in a desc manner"

vicious :: circumstance
vicious
"vicious"
\end{verbatim}

**Listing 3.5** An alternate way of specifying how a stabbing may take place using the function \texttt{transitive_action}.

\begin{verbatim}
transitive_action :: (action, entity, entity) -> completion
transitive_action(act, villain, victim)
"villain performs the action act on victim"

stab1 :: action
stab1
"stab"

brutus :: entity
brutus
"Marcus Junius Brutus"

caesar :: entity
caesar
"Gaius Julius Caesar"
\end{verbatim}
With this alternate definition, the same meaning for the expression \texttt{stab(brutus, caesar)} would be accomplished by the expression \texttt{transitive\_action(stab1, brutus, caesar)}. Here the function \texttt{stab1} is designed such that it has no formal parameters. Instead, the function definition of \texttt{transitive\_action} is responsible for providing the relationship between the action \texttt{stab1} and the entities \texttt{brutus} and \texttt{caesar}.

It is important to note that changing the number of formal parameters that a function accepts results in a completely new and different function. For example, the function called \texttt{stab\_instr} in Listing 3.6 is distinct and different from the \texttt{stab} function in Listing 3.2.

\begin{verbatim}
Listing 3.6 A different declaration for the stab function, which takes an instrument as a formal parameter.

\begin{verbatim}
stab\_instr :: (entity, entity, entity) -> action
stab\_instr(villain, victim, instrument)
"villain stabs victim with instrument"
\end{verbatim}
\end{verbatim}

In the simple case of the function \texttt{stab}, the instrument is not specified. With the new function \texttt{stab\_instr} however, a place is left open for an instrument. Thus, the expression \texttt{stab\_instr(brutus, caesar, knife)} would become “Brutus stabs Caesar with a knife.” In this instance, we passed the function \texttt{knife} as the \texttt{instrument} parameter.

\subsection{3.1.5 Adjustments}

Adjustments give us the opportunity to make subtle tweaks to the meaning of an expression without having to use cumbersome combinations of other functions to achieve the same result. In the framework, adjustments are contained within brackets and follow the function they are applied to. For example, in the previous section, the function \texttt{knife} could have the adjustment \texttt{definite} attached to it. The
resultant adjusted function \texttt{knife[definite]} would have a definite meaning associated with it, in English, would be “the knife.” In a more complex example, we could define a series of adjustments that apply to actions, and in doing so, can reproduce grammatical tense. For example, the expression \texttt{stab[past](brutus, caesar)} would be “Brutus stabbed Caesar.” On the other hand, the expression \texttt{stab[past, complete](brutus, caesar)} would become “Brutus had stabbed Caesar.”

3.1.6 Unspecified Arguments

A special, multi-typed function called \texttt{unspec} is also permitted in the framework. The \texttt{unspec} function states that the argument is to be left \textit{unspecified}. Thus an expression of \texttt{stab.instr[past](brutus, caesar, unspec)} would become “Brutus stabbed Caesar with an unspecified object,” or perhaps more succinctly, “Brutus stabbed Caesar.” The \texttt{unspec} function could also appear in other argument places. Thus, \texttt{stab.instr[past](brutus, unspec, knife)} would become “Brutus stabbed someone / something with a knife.” The exact specification of “someone / something” would rely on the natural-language generator. Similarly, the expression \texttt{stab.instr(unspec, caesar, unspec)} would become “someone / something stabbed Caesar.”

3.1.7 Coercion

In the framework, it is understood that functions of various types may be coerced into different types where necessary. Coercion has a wide range of uses, for example, we may want to reuse the qualifier \texttt{strange} in the expression \texttt{is(brutus, strange)} - “Brutus is strange,” and as a circumstance in the expression \texttt{circ((sing, brutus), strange)} - “Brutus sang strangely.” Coercion allows us to perform this type of semantic type cast automatically, reducing the need for duplicating a large number of qualifiers as circumstances.
In some instances, it is advantageous to collapse a large number of actions, entities, qualifiers, completions and circumstances into lists, as in the case of complex descriptions. For example, \( \text{qual(qual(qual(knife[indefinite], sharp), blue), steel)} \) would be “a sharp, blue, steel knife.” The repeated application of the \text{qual} function makes it difficult to understand the purpose of the expression. In order to improve the clarity of this statement, we can make use of a semantic type known as a \text{list}. This type is identified by enclosing the basic semantic type \text{qualifier} in square brackets. Thus a list of qualifiers will be declared as \([\text{qualifier}]\). The other basic semantic types also have corresponding list types, resulting in the types \([\text{completion}], [\text{action}], [\text{entity}]\) and \([\text{circumstance}]\). It is important to note that any list can be coerced into a single instance of the type, thus a \([\text{qualifier}]\) can be coerced into a \text{qualifier}.

Lists are constructed similarly to those in the Haskell language. For example, we assume that for each semantic type, there is a concatenation operator which produces a list from a function of the correct type, and an existing list. To use this definition however, we must also make use of the type \text{NIL}, an empty list of any given type. A new list can be constructed by concatenating a type with an empty list. For example, a list of qualifiers can be created by using the \text{qcat} function in \text{qcat(sharp, NIL)}. In this case, a list containing one qualifier - \text{sharp} - is the result. By repeating the use of \text{qcat}, additional qualifiers can be added. For example, \text{qcat(sharp, qcat(blue, qcat(steel, NIL)))} would result in the qualifier list of \text{sharp}, \text{blue} and \text{steel}. Intuitively, this can be scaled up to lists of arbitrary length.

To improve the readability of qualifier lists, we use the function \text{qlist}. This function is simply a notational variation of \text{qcat}, allowing us to reduce the creation of a list into a single function. In its basic form, the \text{qlist} function takes an arbitrary number of formal parameters. For example, the expression \text{qlist(blue)} has
only the parameter, but the expression \texttt{qlist(sharp, blue, steel)} is also allowed. Each \texttt{qlist} is transformed into a \texttt{qcat} expression. For example, the expression \texttt{qlist(blue)} is transformed into \texttt{qcat(blue, NIL)}, while the more complex expression \texttt{qlist(sharp, blue, steel)} is transformed into \texttt{qcat(sharp, qlist(blue, steel))}, which is again transformed into \texttt{qcat(sharp, qcat(blue, qlist(steel)))}, which is finally transformed into \texttt{qcat(sharp, qcat(blue, qcat(steel, NIL)))}.

There are corresponding concatenation and list functions available for the other semantic types as well. Thus, the functions \texttt{alist}, \texttt{elist}, and \texttt{clist} exist for the types action, entity and circumstance, that make use of the concatenation functions \texttt{acat}, \texttt{ecat} and \texttt{ccat}. For example, a group of people can be referred to as \texttt{elist(mary, john, peter)}, and a list of actions such as \texttt{alist(sit, stand, walk)}. Various functions can be defined to make use of lists, giving rise to expressions such as \texttt{eat[past, complete](john, elist(sandwich, apple))} - “John had eaten a sandwich and an apple.” Special considerations must be given to lists and adjustments, as well as to completions. Both of these issues will be discussed with more complex examples below.

### 3.1.9 Sequences

Adjusting a list of a given semantic type comes with some caveats. In the previous section, we assumed that the specification of items in a list was not sensitive to the order in which they appeared in the concatenation function. However, there are instances where the ordering matters, for example, if we have a list of completions that we want to occur in a specific order. In this case, a special type of list called a \textit{sequence} is required. The function \texttt{seq} is defined simply as a function that takes two completions and returns a completion, and assumes that each of the completions occurs in a chronological order. In the same way that we defined a \texttt{qlist} to be a variant of the \texttt{qcat}, we can apply the same logic and allow an \texttt{slist} to be a variant of \texttt{seq}. This
lets us tell a story as one large sequence as in $\text{slist[past]}(\text{go(caesar, senate)}, \text{stab(brutus, caesar)}, \text{die(caesar)})$, which can be realized as “Caesar went to the senate. Brutus stabbed Caesar. Caesar died.”

3.1.10 List Adjustments

Care must be taken when applying adjustments to other lists. In each of the list types we have specified so far, such as $\text{alist}$, $\text{qlist}$, $\text{clist}$ and $\text{elist}$, we have assumed that the concatenation function works in a manner similar to an $\text{and}$ operation. However, we can apply other adjustments that modify the lists in various ways. For example, we may want to create a disjunctive list instead, which can be accomplished with the $\text{or}$ adjustment. Thus, $\text{qual(knife indefnite, qlist or(red, blue))}$ becomes “a red or blue knife.” By using various nested forms, we can create more complex lists, such as $\text{elist or(elist(john, peter), elist(bob, mark))}$, realized as “John and Peter or Bob and Mark.”

3.1.11 Different Circumstances

Depending on where we place a circumstance, we can get two different readings for an expression. For example, assume we have defined the actions $\text{sing}$ and $\text{intransitive action}$, and the circumstance $\text{strange}$. We could create the expression $\text{intransitive action(circ(sing past, strange), brutus)}$ with the informal reading “Brutus sang strangely.” However, if we move the $\text{circ}$ so that it modifies the function $\text{intransitive action}$ in the expression $\text{circ(intransitive action(sing past, brutus), strange)}$ we would have the reading of “Strangely, brutus sang.” This difference in placement allows us to add circumstances to a specific action, or to the entire proposition itself.
3.1.12 Quantifiers

Quantifiers are typically used to specify the amount of a particular entity, and can be captured with a set of adjustments. For example, conspirators[all], conspirators[many], and conspirators[few] each refer to different numbers of conspirators, namely “all conspirators”, “many conspirators” and “few conspirators.” While we have only provided three different quantifiers here as adjustments, one may imagine an entire set of quantifiers arranged on a scale from “none” to “all.” Appropriate quantifiers can then be selected from the scale, depending on the quantity needed for the statement in question. It is important to note that while useful natural language terms such as “many” and “few” were selected in these examples as quantifier names, the actual quantifier names on the scale might have simply ranged from “1” to “100.”

While the quantifiers discussed above apply nicely to entities that can easily be counted such as groups of people, there are a number of other quantifiers which apply to sets of objects which are not easily divisible, known as mass entities, such as “gold”, “water”, “sand”, etc. For such entities, we can develop a fractional quantifier scale, allowing us to specify “half of”, “some of”, etc.

3.1.13 Relative Clauses

Relative clauses occur when we need to create a fully formed thought as a qualifier. For example, the fragment “the house where Craig lives” is a relative clause that becomes part of the larger sentence “the house where Craig lives is painted yellow.” Here, the term “house” acts both as an entity to be qualified and as part of the completion that makes up the relative clause. In order to allow relative clauses, we use the function rqual, which takes an entity and a completion and returns a qualified entity. The function works similarly to qual; however, we also use the constant term REL, which is used to refer to the entity that will become part of the relative clause. For example, to represent the meaning of “the house where Craig lives,” we would
have the expression rqual(house, inhabit(craig, REL)).

3.1.14 Interrogatives

The examples provided thus far are all declarative in nature. There are instances however, where it may be beneficial to ask questions. To do so, the framework contains a multi-typed constant ?. The purpose of the ? constant is to ask what the value of a function should be. For example, consider the statement stab_instr(brutus, caesar, ?). The meaning of this particular statement would be expressed in English as “what instrument did Brutus use to stab Caesar?” Depending on where the constant appears, different interrogatives can be formulated. For example stab_instr(brutus, ?, knife) would become “who / what did Brutus stab with a knife?”, and stab_instr(?, caesar, knife) would be “who / what stabbed Caesar with a knife?”

Additional adjustments allow us to ask different sorts of questions. The qu adjustment lets us ask whether the entity in question performed the function it is involved with. For example, stab_instr(brutus, caesar, knife[qu]) would become “did Brutus use a knife to stab Caesar?”, while stab_instr(brutus, caesar[qu], knife) would become “was it Caesar that Brutus stabbed with a knife?” In a similar way, the wh adjustment lets us ask “wh” type questions, such as stab_instr(brutus, caesar, knife[wh]) - “which knife did Brutus use to stab Caesar?”

3.1.15 Speech Acts

Representing what is said during a speech act requires an additional function which introduces the notion of a speaker - the entity or entities performing the speech act, and the listener - the entity or entities that the speech act is directed at. The content of the message will be a semantic expression. It is important to note that the content of the expression may not actually have occurred, it is meant only to represent
an idea that the speaker is attempting to communicate to the listener. To facilitate speech acts, a function called say is defined, which takes three formal parameters - say(speaker, listener, content). For example, say(caesar, brutus, purpose[wh](stab(brutus, caesar))) represents Caesar asking Brutus “why did you stab me?”

3.2 Reproducing Stories

Given the mechanisms we have available in our generation framework, it is possible to reproduce stories by specifying the semantic expressions that underlie their natural language statements. An example of the well known folktale “The Story of the Three Bears”\(^2\) is provided in Listing 3.7. In this example, we will assume that there is a lexicon of semantic functions that is opened prior to reading this story, and that the semantic lexicon defines many of the functions that are needed to tell the details of the story. For example, we will assume that the function bear is defined as a zero-valent function, and that it contains a meaning that would befit a bear.

The set of semantic expressions used to capture the meaning of the story makes use of many of the mechanisms described throughout this section. Comments are contained within C style comment blocks, indicated by /* and */, and in this story, are used to demonstrate the prose that the semantic expressions are meant to convey. The line slist[past] indicates the beginning of the story. Here, the temporal modifier past is used to indicate that all the expressions will be interpreted as having taken place at some point in the past. The first lines following it are included to define the entity constants that will be used throughout the scope of the story. Note that they are declared here, but are not evaluated. Their actual values will not appear as narrative in the resultant story until called by some other set of functions.

The declaration of the function porridge_pot is meant as a convenience. The

\(^2\)The version of the story used here appears in Griffith and Frey (1987).
Listing 3.7 An example of “The Story of the Three Bears”, captured in a series of semantic expressions.

/* The start of the story as a sequence of completions */
slist[past]
(
  /* Constant declarations */
  little_bear, middle_bear, large_bear :: entity,
  little_bear = qual(bear, small),
  middle_bear = qual(bear, medium),
  large_bear = qual(bear, big),

  /* A declaration for the three bears together */
  three_bears :: entity,
  three_bears = elist(little_bear, middle_bear, large_bear),

  /* A function used to specify the owner and purpose of a porridge pot */
  porridge_pot :: (entity, qualifier) -> entity,
  porridge_pot(owner, description) =
    rqual(pot, qlist(description, belong_to(REL, owner),
               used_for(REL, hold(porridge))),

  /* The definition of the porridge pots */
  little_pot, middle_pot :: entity,
  little_pot = porridge_pot(little_bear, little),
  middle_pot = porridge_pot(middle_bear, medium_size),

  /* Once upon a time there were three bears */
  exist(three_bears),

  /* who lived together in a house of their own, in a wood. */
  live_in(three_bears, rqual(house, qlist(belong_to(REL, three_bears),
                               in(REL, woods)))),

  /* One of them was a Little, Small, Wee Bear; and one was a Middle-sized */
  /* Bear, and the other was a Great, Huge Bear. */
  description_of(little_bear),
  description_of(middle_bear),
  description_of(large_bear),

  /* They each had a pot for their porridge, a little pot for the */
  /* Little, Small, Wee Bear, and a middle-sized pot for the Middle */
  /* Bear, and a great pot for the Great, Huge Bear. */
  exist(little_pot),
  description_of(little_pot),

  exist(middle_pot),
  description_of(middle_pot),
  ...
)
function takes two arguments as its parameters. The first is an entity who is to be the owner of the pot, and the second is meant to be a description of the pot. The result of the function is the definition of a unique pot entity who belongs to the owner, is used for holding porridge, and is qualified as having the description that is passed into the function. The line - exists(three_bears) - begins the story, which states that the three bears existed. It is a matter of style as to how the natural language realization system would render these statements in English, or any other natural language. The live_in expression relates the fact that the three bears live in the same house. Note that the use of the REL keyword is used in the relative clause to refer to the item being qualified, in this case, the house.

The next set of functions, each of which involves a description_of, are used to describe the three different bears. It is assumed that the function has been defined such that it will enumerate and describe any qualifiers that are attached a particular entity. In the case of the expression description_of(little_bear), the qualifiers attached to the little_bear entity would be examined and described. The resulting natural language output would be similar to “a little, small, wee bear.” The issue of representing small as “little, small, wee” is an issue left to the realization process. Presumably, it would produce elements of style, perform lexical selection, etc., according to some definition appropriate for the genre.

Both little_pot and middle_pot are declared by making use of the porridge_pot function, which defines a unique pot that belongs to its owner and is used for holding porridge, along with a description. For the little_pot, the pot that is created is rqual(pot, qlist(little, belong_tolittle_bear), used_for(REL, hold(porridge)). A valid natural language realization of this expression would be “a little porridge pot that belongs to the little bear.” The lines exist(little_pot) and description_ of(little_pot) provide descriptions similarly to how they functioned above.
3.3 Expansion

While the previous section demonstrated how a story could be reproduced by specifying a set of semantic expressions for each natural-language statement, there is an alternate way to create stories, based on the idea of expansion. Semantic expressions are able to undergo various forms of transformation, which may result in the preservation of the original semantic meaning with a new form. The primary mechanism that is used to transform semantic expressions is expansion - a process which replaces a given expression with a new, possibly empty set of expressions. The expansions that are applicable to a given expression are defined within the composition field of a function declaration. It is at this point where the work of this thesis begins, since we need to define some new mechanisms within the semantic framework that are not included in Levison et al. (forthcoming).

In the simplest case, functions contain an empty set of expressions in their composition field, and evaluation simply returns the basic semantic meaning contained within the body of the function declaration. Examples include functions such as stab and sing, seen previously. There are however, three additional types of expansions that may occur: mandated choices, random selections and guarded choices. With mandated choices, the original expression is replaced by a different semantic expression. For example, Listing 3.8 contains an example of a function with a mandated choice. In this example, the only possibility for expansion of the expression villainy(villain, victim) is to become the expression kidnap(villain, victim). Note that the argument values of villain and victim get passed on from the villainy function to the kidnap function. For example, if a call was

---

3These expansion types are inspired by similarly named components that operate on syntactic elements in the VINCI system by Levison and Lessard (1992). However, this work redefines the operation of these components with respect to semantic content.

4In this example, and the others following it in this section, only the signature and the expansion choices for the function are provided in the listing. The reader is reminded that other fields such as the English meaning are possible for semantic functions, but are not included here for the sake of brevity.
made to \texttt{villainy(witch, princess)}, then the function \texttt{kidnap} would be called as \texttt{kidnap(witch, princess)}. These named arguments provide a simple way of passing information from a higher-level expression down to a lower-level expression, as we will see when we examine semantic trees in Section 3.3.1.

\textbf{Listing 3.8} An example listing of a function that has a mandated choice.

\begin{verbatim}
villainy :: (entity, entity) -> action
villainy(villain, victim) =
    kidnap(villain, victim)
\end{verbatim}

In a more complex case, the function may have attached to it a number of different possible expansions, one of which is chosen at random. For example, Listing 3.9 contains an example of this. In this example, \texttt{villain_pursuit} function may expand into one of three alternatives, chosen at random. The \texttt{random} keyword tells us that we have a random choice available, while the \texttt{>} character is used to delimit the various possibilities.

\textbf{Listing 3.9} An example listing of a function that has a random choice.

\begin{verbatim}
villain_pursuit :: (entity, entity) -> action
villain_pursuit(villain, hero) =
    random
    > slist(change_into(villain, fish), chase(villain, hero))
    > slist(change_into(villain, wolf), chase(villain, hero))
    > slist(change_into(villain, rooster), chase(villain, hero))
end
\end{verbatim}

The final type of transformation is known as a \textit{guarded choice}, and makes use of a query mechanism that is able to check the surrounding environment. However, before we can examine the operation of the guarded choice, we must first define a \textit{story} and a \textit{semantic tree}.
3.3.1 Semantic Trees

The functions we have seen so far may be represented graphically using a *semantic tree*. The root node of the tree will be the function name, while each of the branches represents one of the formal arguments. For example, assume that we have a function defined as in Listing 3.10.

**Listing 3.10** An example listing of the *kidnap* function.

```haskell
kidnap :: (entity, entity) -> action
kidnap(villain, victim)
"villain kidnaps victim"
```

Given this simple definition, assume that we are evaluating the function as `kidnap(witch, princess)`. The tree representation of this particular function is represented in Figure 3.1.

![Figure 3.1: The *kidnap* function, with labelled branches.](image)

In this figure, the root of the tree is the function *kidnap*. Here, we see that the first formal parameter *villain* is to be the function *witch*, while the second formal parameter *victim* is to be the function *princess*. Any semantic function, no matter how complex, may be represented using this type of tree layout. It is important to note that the picture of the semantic trees shown here differs slightly from those normally drawn by computer scientists. These trees have a left to right, top to bottom orientation. In other words, the root appears at the left of the figure, and the oldest sibling is shown at the top.
To represent function expansion, we will define a second version of a semantic tree. Consider the definition of the \textit{villainy} function we saw previously in Listing 3.8. This example demonstrated a simple mandated choice, namely that the \textit{villainy} function must be rewritten as a \textit{kidnap} function. In instances such as this, the unexpanded function will become the root of a tree, while the expanded function becomes its child. This tree version shows how \textit{concepts} are replaced on the tree. Figure 3.2 demonstrates what this type of semantic tree looks like.

Figure 3.2: A tree depicting a function expansion. Here, one concept is replaced with another.

It is important to note however, that each of the nodes on the tree represents two fully formed functions, both of which can be drawn as we have done previously. In essence, when we use a tree to depict function expansion, we choose not to express the formal parameters of each of the functions. However, these formal parameters remain, and can be thought of as being contained within the body of each node. Figure 3.3 demonstrates this fact.

Figure 3.3: A tree depicting function expansion. Within each node is a tree that represents the formal argument structure of each function.

In this figure, the node on the left represents the \textit{villainy} function, while the node on the right represents its replacement, the \textit{kidnap} function. Embedded within
each node are the semantic trees which represent their formal argument structures. Throughout the course of this dissertation, we intermix these two versions of the tree where it is convenient to do so. For example, since the kidnap function has no further expansions that would result in its concept being replaced on the tree, we will draw its formal parameters. We do this because it is possible that the formal parameters witch and princess may undergo their own separate expansions. Thus, we end up with a hybrid tree like the one depicted in Figure 3.4.

Figure 3.4: A hybrid tree depicting function expansion.

3.3.2 Stories

A story is a collection of semantic expressions, which can be represented as a hybrid semantic tree. The steps taken to form a story are simple. Figure 3.5 provides a graphical depiction of the story building process, in which the function villainy(witch, princess) is expanded. In this example, we begin in Figure 3.5(a) with a single function called villainy that takes two arguments. The first argument is the villain which has the value of witch, and the second is the victim which has the value princess. The arguments will only be evaluated once the function has been expanded. In this example, the definition of the villainy function is consulted in the semantic lexicon, where it is found to be a mandated choice resulting in kidnap. The result of this expansion is provided in Figure 3.5(b). Since the kidnap function offers no further expansion, the arguments to the function, villain and victim, are drawn on the tree and are expanded. The values for these arguments are passed
down from the root node of the tree, and the result is provided in Figure 3.5(c). Like the kidnap function, the zero-valent functions witch and princess undergo no additional expansions themselves upon evaluation, and thus generation ends.

![Semantic Tree Diagram](image)

(a) The semantic tree before expansion of the root node.  
(b) The semantic tree after expansion of the root node.  
(c) The semantic tree after expansion of the arguments villain and victim.

Figure 3.5: The process of building a semantic tree.

To tell entire stories, we can imagine a function called tale that expands into a sequential list of other functions that make up the basis of a story. Figure 3.6 demonstrates this. For the sake of simplicity, in subsequent semantic trees throughout this thesis, we will dispense with representing functions like slist on the tree, and will instead simply draw the resultant list underneath its parent function. For example, when comparing the tree in Figure 3.7 to the tree provided in Figure 3.6, the only difference is that the slist node has been moved up to the tale node.

Semantic trees provide nice graphical representations of the total sum of knowledge contained within a story. It is worthwhile to note however, that on some occasions, the tree may omit information that is considered to be implicitly known or understood. It is left up to the reader to supply reasonable detail as to how something came to
Figure 3.6: A simple semantic tree relating to a story.

Figure 3.7: A simple semantic tree relating to a story, provided without drawing the slist node.
be, as long as such omissions do not result in continuity errors (see Chapter 7). Also noteworthy is the fact that the order of events in a story told to the reader may differ from the order in which they were generated (see Section 3.5).

### 3.3.3 Topoi

In the previous section, we saw that some of the nodes on the semantic tree are used for structuring purposes. For example, the function `tale` acts as a meta-statement responsible for the overarching structure of the story. Similarly, the functions `villainy`, `heroic_act` and `conclusion` may also be labelled meta-statements, since they are capable of providing their own story sub-structures and summaries of story content. Borrowing from literary parlance, we may call these functions *topoi*. As Lancashire (1990) explains, a topos is a *thematic topic* that does not correspond to a specific syntactic manifestation, but rather is a purely semantic idea that can be further developed. In other words, a topos is a summary of a particular idea that can then be elaborated and realized in a concrete fashion. For example, consider the topos “an innocent person is accused of murder” translated from French into English from the SATORBASE website (see Sinclair and SATOR (2009)). This informally stated English string can be formalized with the function `innocent_accused_of_murder`, which in turn, can be realized as a set semantic effect functions in a variety of ways. An example of a very simple expansion is provided in Listing 3.11. Alternatively, the topos may be realized by more complex expansions that involve one or several `slists`, which give rise to entire sets of functions or other topoi that may go on to explain how the innocent was accused.

There are two important items to note regarding topoi. First, the actual composition of a topos is no different from any other semantic expression. The term topos is just a convenient literary label that is used to denote the fact that a function may be further expanded into many other semantic expressions, and that it may provide
Listing 3.11 An example listing of the topos innocent_accused_of_murder.

innocent_accused_of_murder :: (entity, entity) -> action
innocent_accused_of_murder(x, y) =
  slist(
    qual(y, innocent),
    say(x, y, believe(x, murder(y, unspec)))
  )

its own internal structure. Second, each topos has a high-level summary attached to
it so that if the topos is not expanded, the high level summary may be used in its
place. For example, while innocent_accused_of_murder may have many different
and detailed expansion possibilities, the unexpanded expression will have a summary
attached to it that will describe the “gist” of the motif. In this case, a likely summary
might be accuse(x, qual(y, innocent), murder(y, unspec)).

3.4 Tree Queries and Guarded Choices

Coming back to function expansion, we can now examine the notion of a guarded
choice. This type of expansion is separated into two parts: a query of the functions
on the semantic tree to examine some context, and a choice based upon the query
results. Listing 3.12 contains an example of a query and resulting set of choices based
upon the results of the query.

Listing 3.12 An example listing of a function that has a query and a guarded choice.

murder :: (entity, entity) -> action
murder(villain, victim) =
  query item1 : has(villain, *)
    > item1 == knife: slist(stab(villain, victim), die(victim))
    > item1 == hemlock: slist(poison(villain, victim), die(victim))
    > default: kill(villain, victim)
  end

The first part of the process used to expand a guarded choice involves the query
keyword, which is used to set up a search. The query also introduces a variable,
which is used to hold the results of the query. In this instance, the variable item1 will contain whatever semantic function the * is found to match. With this query, we are looking on the semantic tree to see what item the argument villain has in his possession. In other words, if murder(ogre, prince) is called, the query checks to see if the expression has(ogre, *) exists on the semantic tree. Whatever value appearing in the position of the * is returned by the query, and becomes the value of item1. Thus if has(ogre, knife) appears, then item1 will have the value of knife. If has(ogre, qual(hair, red)) appears, then item1 will have the value qual(hair, red).

Once the query is complete, the guarded choices are examined. Each guard is in the form \( > \text{expression}: \text{transformation} \) where expression is a list of valid boolean expressions and transformation is a semantic function that should replace the current function. In the instance that no matches are found, the default rule, indicated by \( > \text{default} \) will be selected. For example, if the function murder(ogre, prince) is called, and if has(ogre, knife) appears on the semantic tree, then the value of item1 will be knife and the function will expand to be slist(stab(ogre, prince), die(prince)). Similarly, if has(ogre, hemlock) appears on the semantic tree, then poison(ogre, prince) will result. If both appear on the tree, then the expression that is closest to the function will be selected. If the query does not return any value, or if the value of the query does not match any of the guards, then the default function of kill(ogre, prince) will be selected.

While guarded choices allow us a degree of control over the expansion of a semantic expression, the query mechanism as described here is a rather blunt instrument. The problem is that we lack a way of restricting the scope of a query so that only a portion of the tree is searched for any given context. Without such a restriction, we risk basing our expansion choice on the wrong context. For example, consider the semantic tree provided in Figure 3.8. Using this tree, if we were to issue the query
item1 : has(villain, *) without specifying the scope of the search or more of the semantics of the query function, we could not be sure which function the query will return. In order to provide an account of how the query function should work, we need to understand how the tree may be traversed using semantic threads.

![Diagram of a story involving Brutus and Caesar](image)

Figure 3.8: A fragment of a simple story involving Brutus and Caesar.

### 3.5 Threads and Reader Chronologies

A *thread* is a means of linking together semantic functions on the tree, where the order is important. They are represented by a series of pointers, each referred to by a specific name. For example, thus far we have only been concerned with how various semantic expressions are placed on the tree, and have ignored how the information may be presented to a reader. While the exact process used to realize the story in a natural language is clearly beyond the scope of this thesis, at least some thought needs to be given to the *narrative sequence* that a reader will experience. This sequence of functions can be specified purely at the semantic level, and may be drastically different
from the order in which they are generated. The order in which our semantic functions are presented to the reader will be called the *reader chronology*, represented on the tree with a thread called *story thread*. Figure 3.9 presents a simple semantic tree that has been threaded with a reader chronology.

![Figure 3.9: A simple semantic tree with a story thread.](image.png)

The order in which the thread progresses through the tree in a reader chronology is used to indicate the way in which the story would be told. In the case of the thread in Figure 3.9, the semantic function *kidnap* would be told first, followed by *rescue* and then ending with *livehappily*. This results in the following story:

> Once upon a time, a princess was kidnapped by a witch. The hero rescued the princess. The hero and princess lived happily ever after.

While in this instance, the story thread follows the order of the semantic functions, it is not always the case that we would want to do so. For example, we could tell the story from the ending and then move forward. We would have something more along the lines of:
Although the hero and princess lived happily every after, this was not always the case. Once upon a time, before the princess and hero were married...

In this instance, we would have a different story thread that runs through the tree, starting at the last node on the tree, and then jumping back to the first few nodes. Notice that we rely on the natural-language generator to add appropriate syntactic structures to explain this jump through time. An example of the thread that occurs with this simple story is provided in Figure 3.10.

![Figure 3.10: A simple semantic tree with a story thread that starts at the end of the tale.](image)

In fact, given the capability of topoi to produce summaries of events, we could ignore the low-level details altogether, and instead tell the story only at the level of the topoi. This would result in a very high-level summary. This type of thread is provided in Figure 3.11. The story that we would get from this thread would be something like:

Once upon a time a villainy occurred. A hero set out to correct the villainy. The end.
This type of summary may be quite useful in instances where only the most basic understanding of the story is needed. For example, one may be able to summarize Romeo and Juliet in this way, breaking it down into:

Two families are engaged in a feud. Boy from one feuding family meets girl in the other. Boy and girl fall in love. Love between boy and girl is denied...

In fact, one may imagine a system where the reader is presented with a story and a mechanism to choose the level of detail. Users could skim the entire summary, or choose to expand a specific topic when they wish to get more detail.

3.5.1 Other Threads

Reader chronologies are not the only type of threading possible on the tree. One possible alternative is a set of character reference threads. As implied by the name, each character in the story would have its own thread, where the thread simply joins together any node (in the order they were generated) that makes reference to
the character. In this way, if we ever need to check a character’s status, it is a simple matter of traversing their respective thread, rather than searching through the entirety of a semantic tree. For example, perhaps we want to know if a Prince came into contact with a merchant. Rather than search the entire tree, we could instead move along the character reference thread for the Prince to determine if he met, or had any interaction with a merchant.

As a final note, the use of such threads is not limited to characters. Any semantic object, location, or idea could also have a semantic thread. In fact, any semantic dimension may have its own thread. For example, threads can be used to keep track of instances of anger, or specific groups of actions. These types of threads become important when searching for continuity errors in a generated story (see Chapter 7).

3.5.2 Limiting Tree Queries

Turning back to tree queries, we now can develop a restriction mechanism that can limit the query to a specific context on the tree. This type of query limitation is known as a thread restriction, since it restricts the query such that it may only examine the set of contexts along a particular named thread. With this type of restriction, both the thread name and a search direction relative to the current node may be specified. For example, consider the threaded semantic tree provided in Figure 3.12.

In this semantic tree, we have a thread called brutus that runs throughout the plot line. Consider the case where we want to know what implement Brutus has in his possession before the murder node is expanded. Recall that our original query was simply item1 : has(villain, *). With this particular query, there is no restriction as to where on the semantic tree we should look for a matching context. To do so, we need to augment the query such that a specific thread is mentioned, and specify a directionality for the search relative to the current node. An example of this new query is provided in Listing 3.13.
Figure 3.12: An example of a semantic tree with a character thread.

Listing 3.13 An example listing of a function with a restricted query.

murder :: (entity, entity) -> action
murder(villain, victim) =
    query item1 : has(villain, *) in thread villain before current
    > item1 == knife: slist(stab(villain, victim), die(victim))
    > item1 == hemlock: slist(poison(villain, victim), die(victim))
    > default: kill(villain, victim)
end
When the **murder** expression is to be expanded, the sequence of events would then unfold as follows. The argument **villain** would be replaced by the actual function that was passed to the **murder** function. In this example, the argument **villain** would become **brutus**, turning our query string into `item1 : has(brutus, *) in thread brutus before current`. The query would then move backwards along the **brutus thread** starting from the **current** node - **murder** - looking for the first matching context for `has(brutus, *)`. According to the semantic tree provided in the figure, the first matching context before **murder** along the **brutus thread** is `has(brutus, knife)`, thus `item1` would become **knife** and the **murder** expression would be transformed into `slist(stab(brutus, caesar), die(caesar))`.

Additionally, more than one variable can be established by a query. For example, perhaps a query needs to check on two different requirements within the semantic tree. For example, the **murder** function may need to check what type of poison Brutus has, whether it be a type that needs to be slipped into a drink, or perhaps one that works on a morsel of food. Listing 3.14 provides an example of this type of query. Here, there are two items that are being checked on the semantic tree. The first is the actual item that the **villain** has, whether it be a **knife** or a **poison**. Then, the type of the item is checked. In the instance of the **poison**, a second query checks to see if it is a **liquid** or a **food**. Depending on the type results returned, the **villain** will either **stab** the **victim**, or will **poison** the **victim** by slipping it in his drink or in his food.

Finally, it is worthwhile to note that we can use a distance measure to limit our tree queries. For example, perhaps we wish to check on a **villain** that occurs along a semantic thread, but we do not want to return results that are further away then nine nodes along the thread. Using the keyword **distance** as a part of our query, we can impose this limitation, as demonstrated in Listing 3.15.

---

5We also could have used the keyword **after** to perform a search through the nodes that come after the current node.
Listing 3.14 An example listing of a function with two queries to the semantic tree.

```plaintext
m Pers :: (entity, entity) -> action
m Pers(villain, victim) =
  query item1 : has(villain, *) in thread villain before current
  query type1 : qual(poison, *) in thread villain before current
    > item1 == knife:
      slist(stab(villain, victim), die(victim))
    > item1 == poison && type1 == liquid
      slist(
        pour(villain, poison, qual(cup, belong_to(REL, victim))),
        drink(victim, from(qual(cup, belong_to(REL, victim)))),
        die(victim)
      )
    > item1 == poison && type1 == food
      slist(
        insert(villain, poison, qual(food, belong_to(REL, victim))),
        eat(victim, qual(food, belong_to(REL, victim))),
        die(victim)
      )
    > default:
      kill(villain, victim)
  end
```

Listing 3.15 An example listing of a function with two queries to the semantic tree.

```plaintext
m Pers :: (entity, entity) -> action
m Pers(villain, victim) =
  query item1 : has(villain, *) in thread villain before current distance 4
  query type1 : qual(poison, *) in thread villain before current distance 5
    > item1 == knife:
      ...
```
3.6 Summary

A description of the semantic framework defined by Levison et al. (forthcoming) was provided. Complex meaning can be achieved by combining several functions in function applications to produce semantic expressions. A method of reproducing the meaning of a simple fairy tale through the sequential ordering of sets of semantic expressions was described. An alternate method of generating stories using the process of expansion was explained. The concept of semantic threads was introduced to explain how different reader chronologies may be defined. A set of syntactic conventions in the form of code listings was introduced to illustrate the concepts of the semantic formalism.
Chapter 4

Generating Proppian Folktales

Now that the organization and development of text-generation systems have been explored, as well as the details pertaining to the generation of stories using our new system of function expansion, the next problem is understanding how various types of stories are structured. While the study of a wide range of literature is beyond the scope of this research, we will focus here on an in-depth analysis of folktales produced by Propp (1928). Using this analysis as the basis of our functions, we will demonstrate how functional expansion can generate interesting content. Additionally, this type of simple demonstration will serve to reveal inadequacies in the current generation methodology, and provide a road-map of the additions that are required to generate more complex stories.

4.1 Propp’s Morphological Analysis

Propp’s (1968) work was undertaken to gain insight into the structure of the folktale. Folklorists at the time were attempting to make comparisons of folktales across different cultures, without detailed knowledge of how tales were structured. Propp

---

1This thesis makes use of the 1968 second edition English translation of Propp’s original work written in Russian.
proposed that without a clear and concise way of classifying and examining the inner components of a tale, no useful and scientifically motivated conclusions could be drawn about the origin of a tale or how it might manifest itself in other cultures. While systems of classification existed at the time, they were insufficient or contained over-generalizations that made their use problematic. For example, early classification schemes divided tales into groupings such as stories containing animals or stories containing elements of everyday life. These attempts were deemed insufficient, since a tale may fall into more than one of these different categories, or worse, may not fall under any category at all.

As Propp explains, historical work done by Veselóvskij (1913)\(^2\) attempted to develop such a definition by isolating the meaning of a *motif* and a *theme* within the folktale genre. According to Propp, Veselóvskij’s motifs were elements that were involved in the realization of themes, and were believed to be non-decomposable units. For example, according to Propp, a motif in a tale may be “a stepdaughter leaving home.” Unfortunately, while this two level scheme of classification was a step in the right direction, it turned out that motifs could be decomposed into other more basic *elements*, which themselves could be combined in many different ways. Using Propp’s example to illustrate this point, there may be a motif where “a dragon kidnaps a tsar’s daughter.” Each component of the motif, the dragon, the tsar’s daughter, and the kidnapping, can be replaced with other components that convey the same motif, which in this instance describes a disappearance. Propp suggests that the main reason why Veselóvskij’s work fell out of favour was due to the lack of a solid definition of a motif. Propp’s investigation makes mention of further work done by Bédier (1893), who suggested that these basic units could be sorted into both *constant* and *variable* elements. According to Propp, Bédier’s work attempted to produce a formal notation that would capture the relationships between these constant and variable elements.

\(^2\)Note that references to Veselovskij (1913) and Bédier (1893) are being included for the sake of completeness. The author does not have access to these works.
However, this work was not successful due to issues surrounding the definition of constant elements.

The novel contribution by Propp was the way in which he approached the division of units. He noticed that in many stories, while the characters known as *dramatis personae* are different, the actions and functions they perform are virtually identical. On that basis, Propp proceeded to classify the functions that the characters in a story may perform. While his investigation covered only a portion of the literature available, his reasoning for doing so was based on his observation that the set of functions was, in actuality, very limited. He believed that the progression of the elements within the folktale followed a set pattern, and proceeded to outline a general structure of how folktales are formed.

4.1.1 Functions

Propp was able to classify all of the actions that occur within a folktale into different types of functions. For example, he defines function \( A \) as a *Villainy*, and assigns it a high level description of “the villain causes harm or injury to a member of the family.” Contained within this function are a number of sub-functions, each of which represents a different kind of villainy. In the case of the *Villainy* function, there are 19 different sub-functions that may occur, ranging from abductions, to bodily injuries, to incitements of war. Propp goes on to provide numerous examples of how each sub-function could be manifested in actual tales. A simple listing of each of the upper level functions with their descriptions is provided in Table 4.1.

4.1.2 Moves

Propp defined a folktale to be a story which has one or more *moves* that may be intertwined with one another. According to his definition, a move is a sequence of functions that starts out with a *Villainy* (\( A \)) or *Lack* (\( a \)), progresses through a
<table>
<thead>
<tr>
<th>Designation</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>Initial situation</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>Absentation</td>
<td>A family member absents himself from home</td>
</tr>
<tr>
<td>γ</td>
<td>Interdiction</td>
<td>An interdiction is addressed to the hero</td>
</tr>
<tr>
<td>δ</td>
<td>Violation</td>
<td>The interdiction is violated</td>
</tr>
<tr>
<td>ε</td>
<td>Reconnaissance</td>
<td>The villain makes an attempt at reconnaissance</td>
</tr>
<tr>
<td>ζ</td>
<td>Delivery</td>
<td>The villain receives information about his victim</td>
</tr>
<tr>
<td>η</td>
<td>Trickery</td>
<td>The villain attempts to deceive his victim in order to take possession of him or his belongings</td>
</tr>
<tr>
<td>θ</td>
<td>Complicity</td>
<td>The victim submits to deception and thereby unwittingly helps his enemy</td>
</tr>
<tr>
<td>A</td>
<td>Villainy</td>
<td>The villain causes harm to a member of a family</td>
</tr>
<tr>
<td>a</td>
<td>Lack</td>
<td>One member of a family either lacks something or desires to have something</td>
</tr>
<tr>
<td>B</td>
<td>Mediation</td>
<td>Misfortune or lack is made known; the hero is approached with a request or command; he is allowed to go or is dispatched</td>
</tr>
<tr>
<td>C</td>
<td>Counteraction</td>
<td>The seeker agrees to or decides upon counteraction</td>
</tr>
<tr>
<td>†</td>
<td>Departure</td>
<td>The hero leaves home</td>
</tr>
<tr>
<td>D</td>
<td>First Donor Function</td>
<td>The hero is tested, which prepares the way for his receiving either a magical agent or helper</td>
</tr>
<tr>
<td>E</td>
<td>Hero’s Reaction</td>
<td>The hero reacts to the actions of the future donor</td>
</tr>
<tr>
<td>F</td>
<td>Provision</td>
<td>The hero acquires the use of a magical agent</td>
</tr>
<tr>
<td>G</td>
<td>Guidance</td>
<td>The hero is transferred, delivered, or led to the whereabouts of an object of search</td>
</tr>
<tr>
<td>H</td>
<td>Struggle</td>
<td>The hero and villain join in direct combat</td>
</tr>
<tr>
<td>J</td>
<td>Branding</td>
<td>The hero is branded</td>
</tr>
<tr>
<td>I</td>
<td>Victory</td>
<td>The villain is defeated</td>
</tr>
<tr>
<td>K</td>
<td>Liquidation</td>
<td>The initial lack or misfortune is liquidated</td>
</tr>
<tr>
<td>↓</td>
<td>Return</td>
<td>The hero returns</td>
</tr>
<tr>
<td>Pr</td>
<td>Pursuit</td>
<td>The hero is pursued</td>
</tr>
<tr>
<td>Rs</td>
<td>Rescue</td>
<td>Rescue of the hero from pursuit</td>
</tr>
<tr>
<td>o</td>
<td>Unrecognized Arrival</td>
<td>The hero, unrecognized, arrives home or in another country</td>
</tr>
<tr>
<td>L</td>
<td>Claims</td>
<td>A false hero presents unfounded claims</td>
</tr>
<tr>
<td>M</td>
<td>Task</td>
<td>A difficult task is proposed to the hero</td>
</tr>
<tr>
<td>N</td>
<td>Solution</td>
<td>The task is resolved</td>
</tr>
<tr>
<td>Q</td>
<td>Recognition</td>
<td>The hero is recognized</td>
</tr>
<tr>
<td>Ex</td>
<td>Exposure</td>
<td>The false hero or villain is exposed</td>
</tr>
<tr>
<td>T</td>
<td>Transfiguration</td>
<td>The hero is given a new appearance</td>
</tr>
<tr>
<td>U</td>
<td>Punishment</td>
<td>The villain is punished</td>
</tr>
<tr>
<td>W</td>
<td>Wedding</td>
<td>The hero is married and ascends the throne</td>
</tr>
</tbody>
</table>

Table 4.1: A listing of the functions that occur in folktales, reproduced from Propp (1968).
number of intermediate functions, and eventually terminates with a *Wedding* (*W*), or the resolution of one or more major plot complications. The combination of various moves can be quite complex. Propp identified six different ways in which moves may be intertwined. The first case is depicted in Figure 4.1.

I. A________W

II. A________W

Figure 4.1: The first move combination, reproduced from Propp (1968).

In this figure, the progression of the story is understood to occur from left to right. Each line of the figure represents a different move. Here, a fully completed move (I) is followed by a second, fully completed move (II). In this figure, and the simple figures that follow it, the A________W indicates that the move starts with the function *Villainy* (*A*), and ends with the function *Wedding* (*W*). The line separating the functions *A* and *W* is used to indicate that other functions appear in between.

In the second case of move combination, a second move (II) interrupts the progression of the first (I). The second move then completes, and the first interrupted move is resumed until it reaches its own completion. In Figure 4.2, the set of dots (····) indicates that progression along the first move is suspended until the second move has been resolved.

I. A________G  ··········· K________W

II. a________K

Figure 4.2: The second move combination, reproduced from Propp (1968).

In the third case of move combination, a second move (II) interrupts the first (I). However, a third move (III) interrupts the second, creating a complex story line. As in the previous combination, the third move will reach its completion, followed by either the completion of the first, or the second move. This move combination is represented in Figure 4.3.
In the fourth case of move combination, two moves begin simultaneously by the introduction of two villainies. With such combinations, usually the first move (I) reaches its completion in the form of a Liquidation, followed by the second move (II) reaching its completion. This move combination is represented in Figure 4.4.

In the fourth case of move combination, two moves begin simultaneously by the introduction of two villainies. With such combinations, usually the first move (I) reaches its completion in the form of a Liquidation, followed by the second move (II) reaching its completion. This move combination is represented in Figure 4.4.

The fifth move combination involves a second move (II) interrupting the first move (I). However, the resolution to both moves is the same, and is shared between them. The first move is suspended until the resolution begins, at which point both moves have a common ending. This move combination is represented in Figure 4.5.

The sixth and final move combination involves tales where there are two heroes or seekers on a quest. In such instances, the first move typically branches into two other moves, which are then completed separately, and are usually resolved with the same set of functions. The divergence of the two seekers is indicated by <, and an additional function is defined by Propp to account for the fact that one seeker usually transfers an object to the other. This move combination is represented in Figure 4.6.
4.1.3 The Structure of a Folktale

Propp’s work also outlines the general structure of a folktale. First, he examined the structure of an individual move. As discussed above, a move begins from a Villainy (A) or Lack (a) and ends with a Wedding (W) or conclusion of a plot complication. Propp defined the progression of these functions by means of an equation, provided in Figure 4.7. In the progression, we see that all moves start out with functions A, B, C, ↑, D, E, F and then G. At this point, the move may branch to follow functions H-L, or functions L-Rs. Regardless of the path, it will then end with the progression Q, Ex, T, U, and W. When realized in a folktale, this sequence of functions may be interrupted by other moves in the set of combinations discussed in the previous section.

\[
\begin{align*}
A & \quad B & \quad C & \quad \uparrow & \quad D & \quad E & \quad F & \quad G & \quad H & \quad I & \quad K & \quad L & \quad M & \quad J & \quad N & \quad K & \quad Q & \quad Ex & \quad T & \quad U & \quad W
\end{align*}
\]

Figure 4.7: The progression of functions within a move, reproduced from Propp (1968).

Turning from moves to complete folktales, in its simplest form a folktale consists of a single move. Thus, in general, folktales of the Proppian type will take on the general structure indicated above. However, it is important to note that some functions may be omitted from the general progression. These omissions may be made as long as the function being omitted is not linked to another function. For example, the Hero’s Reaction (F), where the hero responds to a test, cannot be omitted when there is a First Donor Function (D) in which the test is put to the hero. With this in mind,
an example of a valid folktale may be $A^{ii}$, $D^3$, $E^3$, $F^{iv}$, $M$, $N$, $W^*$, which corresponds to tale 100 in the collection that Propp was examining. This specific instance of a folktale progression is one of 46 different schemas that Propp explicitly defines, and is used to demonstrate how different varieties of tales occur from the same generic progression.

In addition to function omission, there are other instances where the general progression of a folktale may differ. First, some sequences of functions may be repeated. For example, if the hero responds negatively to a test, a different test may be put to him or her. This repetition is known as *trebling*, and occurs in many different folktales. Second, some sequences of functions may be inverted. Propp’s example is that of the hero leaving home - *Departure* (↑) - before the usual sequence of events - *Villainy* ($A$), *Mediation* ($B$), *Counteraction* ($C$), and then *Departure* (↑). In other words, the progression ↑$ABC$ is encountered, instead of the usual $ABC$↑.

## 4.2 Proppian Generation

While Propp’s original work was focused on analyzing the structure of the folktale, we can begin to see how the morphology can be used for generation purposes. A collection of previous work already does so, for example, see Klein (1975), Grasbon and Braun (2001), Fairclough and Cunningham (2003), Levison and Lessard (2004) and Peinado and Gervás (2006). Each of the functions that Propp has defined can be captured as a series of semantic expressions. For example, in the *Villainy* function, Propp enumerated 19 different possibilities that could occur. Using the generation framework, we can apply our notation to define a function called *villainy*³, and endow it with expressions that capture each of the 19 different possibilities. The function *villainy* would be designed to accept two formal parameters, a *villain* and

³We will depart from using Propp’s function names such as $A$, $B$, $C$, etc. and instead will use his descriptive names *villainy, mediation, beginning_counteraction*, etc.
a *victim*. The values of these parameters would be passed to the expressions for each of the expansion possibilities. An example of how the *villainy* function might look is provided in Listing 4.1.

**Listing 4.1** An example of the *villainy* function.

```plaintext
villainy :: (entity, entity, entity) -> action
villainy(villain, victim) =
    random
    > kidnap(villain, victim)
    > steal(villain, sunlight)
    > pillage(villain, crops)
    > stab(villain, victim)
    > threaten(villain, eat(villain, victim))
end
```

While this gives us a way to represent each Proppian function, we have yet to determine how the *structure* of the story is produced. There are two separate approaches that can be used. In the first instance, we can make use of the 46 different schemas that Propp identified, encoding them directly as a specific set of rules in the framework. For example, one story schema, mentioned in Section 4.1.3, is the progression A\textsuperscript{ii}, D\textsuperscript{3}, E\textsuperscript{3}, F\textsuperscript{iv}, M, N, W*. The numbers associated with each of the functions indicate the expansion route that the function should take. For example, D\textsuperscript{3} indicates that the third expansion possibility for function D is the one used for the tale. Propp defines this sub-function as “A dying or deceased person requests the rendering of a service.” Using this structural method, we can create 46 unique stories in the framework, and simply choose one at random when asked to tell a story, populating it with the dramatis personae as needed.

The second approach to generating structure makes use of the progression that Propp identified. Instead of relying directly on a set of hard-coded schemas to develop stories, we can use the progression and encode it into a type of generative grammar. Propp himself suggested that this could be done to experimentally verify the correctness of his analysis.
Both of these approaches for generating folktales will be explored in the following sections. They echo the work of others such as Klein (1975) and Fairclough and Cunningham (2003), and are meant only to demonstrate some of the current shortcomings of the stories produced in this semantic framework.

4.2.1 Schema Expansion

4.2.1.1 Function Hierarchy

In order to tell a folktale using the schemas that Propp identified, we must first start with the definition of the most basic functions that will become the events and characters of the story itself. For example, many of the different stories have interactions with secondary characters such as beggars, townspeople or shoemakers. In addition to these characters, there are locations that characters visit, such as mountains, forests, towns and other foreign lands. Within these locations, there are other story items such as berries, rabbits and trees. All of these characters, locations and objects need to be defined in the semantic lexicon before generation can occur. Listing 4.2 contains an example of some of the entities that need to be defined within our framework. In previous function declarations, we used newline characters to separate the items involved in a function declaration. Here a vertical bar | is used to delimit the various fields. Recall from Chapter 3 that the first field indicates the signature, the second field indicates the function’s composition, and the third field indicates the English description of the meaning. Note that these functions do not constitute an exhaustive list of what is needed to generate a folktale. The ... is used to indicate that there may be many more declarations.

Once the basic functions have been defined, it is necessary to create the functions that correspond to the set of actions that a dramatis personae may take in the course of the story. These were explored and aggregated in Table 4.1. However, given the
Listing 4.2 The definition of some of the basic functions needed to populate a folktale world.

```c
/* Basic entities */
river :: entity|river|"river"
townspeople :: entity|townspeople|"townspeople"
blacksmith :: entity|blacksmith|"blacksmith"
goldsmith :: entity|goldsmith|"goldsmith"
shoemaker :: entity|shoemaker|"shoemaker"
cook :: entity|cook|"cook"
food :: entity|food|"food"
beggar :: entity|beggar|"beggar"
mountains :: entity|mountains|"mountains"
forest :: entity|forest|"forest"
lake :: entity|lake|"lake"
...
```

fact that the schema makes use of very specific instances of a villainy, we need to look closer at each function to understand what needs to be encoded. Each of the functions that appears in the table has several different sub-functions. For example, based on Propp’s analysis, the third form of a villainy is “The villain pillages or spoils the crops.” This requires that we create a function called `pillage` that captures this meaning. Additionally, we will explicitly create a function called `villainy3` that then has the `pillage` function as the only choice for its expansion. In effect, both the function `villainy` and `villainy3` become topoi. Our motivation as to why we are structuring our functions in this way will become evident when we examine the alternate method of generating story structures. For now, it is sufficient to know that every sub-function that occurs in the `villainy` function will have its own unique function declaration, and in turn, these sub-function declarations make use of other functions. An example of the functions that would need to be declared is provided in Listing 4.3.

Now that the basic functions of the story have been defined, it is a simple matter to create a function that represents a particular schema. In order to do this, we will make use of the different schemas that Propp defines in his work. For example, tale
Listing 4.3 The definition of some of the basic functions needed to populate a folktale world.

/* Basic functions */
kidnap :: (entity, entity) -> action
kidnap(en1, en2)
"en1 kidnaps en2"

pillage :: (entity, entity, entity) -> action
pillage(en1, en2, en3)
"en1 pillages en3 from en2"

steal :: (entity, entity, entity) -> action
steal(en1, en2, en3)
"en1 steals en3 from en2"

...  

/* Proppian Functions and Sub-Functions */
villainy1 :: (entity, entity) -> action
villainy1(villain, victim) = kidnap(villain, victim)

villainy2 :: (entity, entity) -> action
villainy2(villain, object) = steal(villain, unspec, object)

villainy3 :: (entity, entity, entity) -> action
villainy3(villain, object, victim) = pillage(villain, victim, object)

villainy4 :: (entity) -> action
villainy4(villain) = steal(villain, unspec, sunlight)

...
131 has the following progression:

\[ A^1 \ B^1 \ C \uparrow H^1 \ I^1 \ K^4 \downarrow W_0 \]

This structure can be reproduced by creating a topos called \texttt{tale131} and encoding the list of functions that occurs within it in an \texttt{slist}. An example of how this is done is provided in Listing 4.4.

**Listing 4.4 The definition for schema \texttt{tale131}.**

```haskell
/* Schema Definitions */
tale131 :: (entity, entity, entity) -> [completion]
tale131(hero, villain, victim) =
  slist(
    villainy1(villain, victim),
    mediation(hero, victim),
    counteraction(hero),
    departure(hero),
    struggle1(hero, villain),
    victory1(hero, villain),
    liquidation4(hero, villain),
    return(hero),
    wedding0(hero, victim)
  )
```

This same process can be repeated for the other story schemas that Propp captures in his research. Thus, we can imagine a declaration for semantic function \texttt{tale100}, \texttt{tale106}, etc. With the schema functions in place, all that remains is to define a function called \texttt{tale} that will accept a set of formal parameters that correspond to the set of dramatis personae that will take place in the folktale, and will expand into one of our folktale schemas at random. The definition of this function is provided in Listing 4.5.

As can be seen in the listing, the formal parameters are labelled \texttt{hero}, \texttt{villain}, \texttt{victim}, \texttt{donor} and \texttt{magical\_object}. The functional values that are passed to each of these parameters will become those respective actors in the expanded tale. Additionally, as is evident in the listing, not all of these formal parameters are used in
**Listing 4.5** The definition of the `tale` function.

```haskell
/* The tale meta-function */
tale :: (entity, entity, entity, entity, entity) -> [completion]
tale(hero, villain, victim, donor, magical_object) =
    random
    > tale106(hero, villain, victim, donor)
    > tale108(hero, villain, victim, donor)
    > tale114(hero, villain, victim, donor, magical_object)
    > tale131(hero, villain, victim)
    ...
```

every single schema. For example, the schema corresponding to `tale131` does not make use of the `donor` and `magical_object` parameters.

### 4.2.1.2 Dramatis Personae

At this point, we have yet to define the set of dramatis personae that may be involved in a folktale. A simple solution is to define a number of different characters in our semantic lexicon, and define a set of functions which choose at random a hero, villain, victim, donor, etc. We can then pass the values that these functions create into the `tale` function. An example of the function that chooses a hero at random is provided in Listing 4.6. The function simply makes a random selection from a pre-defined set of hero type characters.

**Listing 4.6** The definition of the `random_hero` function.

```haskell
random_hero :: entity
random_hero =
    random
    > prince_montague
    > prince_robert
    > prince_william
end
```

This type of random character selection will suffice for the moment, however, there are complications when choosing various characters. For example, consider a folktale in which Princess Olivia is kidnapped, and her father seeks out a hero to
rescue her. We need to make sure that Princess Olivia actually has a father, and was not orphaned as a child. This type of checking must be done to ensure that the continuity of the story is free from errors. Chapter 7 examines these types of issues more closely. For the moment however, we are only interested in the structure of the story, and will assume that any randomly selected character for a specific type of dramatis persona fits the requirements of the role.

4.2.1.3 A Generation Example

With the fundamentals of schema expansion explained, we can turn to a generation example. To begin, we assume that an interactive environment exists in which a user can issue commands to load various semantic lexicons into memory, define constants, and evaluate the results of a particular semantic expression. Given this environment, a user would need only to create a list of entity constants for the story, and simply ask the system to evaluate the expression tale. The list of constants and the statement for evaluating the tale expression is provided in Listing 4.7.

Listing 4.7 The list of semantic expressions needed to generate a folktale.

```plaintext
hero, vil, vic, donor, mag_obj :: entity
hero = random_hero
vil = random_villain
vic = random_victim
donor = random_donor
mag_obj = random_magical_object

tale(hero, vil, vic, donor, mag_obj)
```

In this listing, the first line contains a multi-constant declaration. Essentially, each of hero, vil, vic, donor and mag_obj is declared to be of type entity. The lines following assign a constant value to each of these entities. For example, the random_hero function will pick one of prince_montague, prince_robert or prince_william as a function. For the purposes of this generation example, we will assume that random_
hero selects \texttt{prince}_\texttt{montague} as the expansion choice. Thus, the constant hero will always return \texttt{prince}_\texttt{montague}. Once the generation of each of the dramatis personae is complete, the expression \texttt{tale(hero, vil, vic, donor, mag_obj)} starts the expansion process. The semantic tree will begin with a single node as indicated in Figure 4.8.

![tale](image)

Figure 4.8: The semantic tree at the start of the generation process.

The first part of the evaluation of the \texttt{tale} expression will result in a selection, at random, of one of the pre-defined set of schemas. For this example, we will assume that \texttt{tale131} was selected. The result of this expansion is provided in Figure 4.9.

![tale](image)

Figure 4.9: The semantic tree after \texttt{tale} is expanded.

Once the schema has been selected, it will be expanded into its mandated choice \texttt{slist}. The \texttt{slist} contains the rest of the functions that the schema will call. The result of this expansion is provided in Figure 4.10. Once \texttt{tale131} has been expanded, we begin to expand each of the sub-functions involved in the \texttt{slist}, starting with \texttt{villainy1}. With the Proppian morphology, the function \texttt{villainy1} is defined as the expression \texttt{kidnap(villain, victim)}. Thus, this new semantic expression replaces the \texttt{villainy1} expression. Since there are no further replacements that exist in the definition of the \texttt{kidnap} function, the sub-functions passed to \texttt{kidnap} are expanded. These are the dramatis personae that correspond to \texttt{vil} and \texttt{vic}. For the purposes of this example, we will assume that \texttt{witch} is the \texttt{vil} and \texttt{princess_olivia} was the \texttt{vic}. The result of this set of replacements is provided in Figure 4.11.
Figure 4.10: The semantic tree after \texttt{tale131} is expanded.

Figure 4.11: The semantic tree after \texttt{villainy1} is expanded.
The process of expansion will continue in the same fashion for the expression \texttt{mediation}, and then \texttt{counteraction} and so on, until all of the expressions have been fully expanded. It is important to note however, that something slightly different happens with the functions \texttt{counteraction} and \texttt{liquidation4}. In both cases, the choice of the expansion relies on information that is found on the semantic tree that was placed there by previous expansions. For example, \texttt{liquidation4} is described by Propp as “The object of a quest is obtained as a direct result of preceding actions.” In fact, the whole family of liquidation functions rely on what villainy occurred at the start of the story. In this particular folktale, the “object” referred to here is the victim of the story. Thus the proper expansion of the function \texttt{liquidation4} must rely on a tree query, such as the one provided in Listing 4.8.

\begin{quote}
\textbf{Listing 4.8} The tree query involved with the \texttt{liquidation4} function.

\begin{verbatim}
liquidation4 :: (entity, entity, entity) -> action
liquidation4(hero, villain, victim) =
    query action1 : *(villain, victim)
        in thread villain before current
        where parent like villainy
        > action1 == kidnap : free(hero, victim)
    ...
\end{verbatim}
\end{quote}

In this listing, we need to check to the left of the current node to see what villainy actually occurred at the start of the story. Additionally, since the wild-card in \texttt{*(villain, victim)} is much too general, the query needs to be constrained to look only at the type of villainy that occurred. The query operator has thus been updated to allow for a \texttt{where} clause, the purpose of which is to provide an additional restriction. In this case, the \texttt{like} keyword tells the query to look only at expressions who have a parent node that is a type of \texttt{villainy}. This same sort of query is issued for the \texttt{counteraction} expression, since the function needs to know what exactly the hero is agreeing to do.

Once all of the expansions are complete, generation ends. The result is a set
returned home. The father of Princess Olivia gave Prince Montague lots of money.

4.2.2 A Proppian Story Grammar

4.2.2.1 Function Hierarchy

An alternative method to using hard-coded schemas is to make use of the general progression that Propp identified. In order to do so, we can reuse many of the elements that we created for use in the schema-expansion methodology. In particular, we can use the set of entities that were previously defined, as well as the basic functions relating to the actions such as kidnap, steal, etc. We can also make use of the Proppian sub-functions that were defined, examples of which include villainy1, villainy2, etc. The main difference in strategies is that the story-grammar approach makes use of an extra layer of organization. We will collect each Proppian sub-function into a corresponding parent function. For example, we had the sub-functions villainy1, villainy2, etc., separately defined. We will now create a new function called villainy, and define it to be a random selection of one of these sub-functions. This process continues with each of the other sets of sub-functions that were previously defined. An example of the villainy function is provided in Listing 4.10.

Listing 4.10 The definition of the villainy function.

```plaintext
villainy :: (entity, entity) -> action
villainy(villain, victim) =
  random
  > villainy1(villain, victim)
  > villainy2(villain)
  > villainy3(villain)
  > villainy4(villain, victim)
  ...
```

Once all of the Proppian functions have been defined, it is necessary to create a series of functions capable of modelling various folktale move combinations. The
first type of move function we can define is called `full_move`. The function randomly expands into one of several different possibilities. In a single, fully-formed move, the folktale follows the general progression, starting at function `A`, moving through functions `H-L` or functions `L-Rs`, and ending at function `W`. This results in two different schemas that need to be captured for `full_move`, one that follows the `H-L` progression, and one that follows the `L-Rs` progression. Since both possibilities start with the progression `Villainy (A)` through to `Guidance (G)`, we will create a convenience function called `start_move` that has this as its composition. We can reuse this function for other move combinations as well. An example of the `full_move` and `start_move` functions is provided in Listing 4.11. Here the `...` is used to indicate that additional functions would be inserted according to the Proppian progression. These are left out here for space considerations.

An additional type of move function that we need to model begins with a `Villainy (A)` or `Lack (a)`, and ends with a `Liquidation (K)`. We will label this type of move `liquidation_move` and encode these two possibilities as its composition. This particular function is needed to model stories that have moves that interrupt other moves. An example of its composition is provided in Listing 4.12. We can now create a function called `interrupted_move`, that captures an interwoven move. It begins by using the function `start_move`, continues with `liquidation_move`, and then moves from function `Liquidation (K)` through to `Wedding (W)`. This definition is provided in Listing 4.13.

We now have the tools necessary to account for the various move combinations that can occur within a folktale. Given these various moves, we can create a function called `tale` that will expand into various move combinations. In its simplest form, a folktale contains a single move, thus the first expansion possibility in `tale` is simply `full_move`. Turning to more complicated tales, a full move may follow the completion of a full move, leading to the second expansion possibility. In the third expansion
Listing 4.11 The definition of the start\_move and full\_move functions.

\[
\text{start\_move} :: (\text{entity, entity, entity, entity, entity}) \to [\text{completion}]
\]
\[
\text{start\_move}(\text{hero, villain, victim, donor, magical\_object}) =
\]
\[
\begin{aligned}
\text{slist(}
\quad & \text{villainy(}\text{villain, victim, magical\_object}), \\
\quad & \text{mediation(}\text{hero, victim}), \\
\quad & \text{counteraction(}\text{hero}), \\
\quad & \text{...} \\
\quad & \text{guidance(}\text{hero, donor})
\end{aligned}
\]

\[
\text{full\_move} :: (\text{entity, entity, entity, entity, entity}) \to [\text{completion}]
\]
\[
\text{full\_move}(\text{hero, villain, victim, donor, magical\_object}) =
\]
\[
\begin{aligned}
\text{random} \ > \ & \text{slist(}
\quad & \text{start\_move}(\text{hero, villain, victim, donor, magical\_object}), \\
\quad & \text{struggle(}\text{hero, villain}), \\
\quad & \text{branding(}\text{hero}), \\
\quad & \text{...} \\
\quad & \text{wedding(}\text{hero, victim})
\end{aligned}
\]
\[
\begin{aligned}
\text{random} \ > \ & \text{slist(}
\quad & \text{start\_move}(\text{hero, villain, victim, donor, magical\_object}), \\
\quad & \text{claims(}\text{hero, villain}), \\
\quad & \text{task(}\text{hero, villain}), \\
\quad & \text{...} \\
\quad & \text{wedding(}\text{hero, victim})
\end{aligned}
\]

end

Listing 4.12 The definition of the liquidation\_move function.

\[
\text{liquidation\_move} :: (\text{entity, entity, entity, entity, entity}) \to [\text{completion}]
\]
\[
\text{liquidation\_move}(\text{hero, villain, victim, donor, magical\_object}) =
\]
\[
\begin{aligned}
\text{random} \ > \ & \text{slist(}
\quad & \text{villainy(}\text{villain, victim}), \\
\quad & \text{mediation(}\text{hero, victim}), \\
\quad & \text{counteraction(}\text{hero}), \\
\quad & \text{...} \\
\quad & \text{liquidation(}\text{hero, villain, victim})
\end{aligned}
\]
\[
\begin{aligned}
\text{random} \ > \ & \text{slist(}
\quad & \text{lack(}\text{villain, victim}), \\
\quad & \text{mediation(}\text{hero, victim}), \\
\quad & \text{counteraction(}\text{hero}), \\
\quad & \text{...} \\
\quad & \text{liquidation(}\text{hero, villain, victim})
\end{aligned}
\]

end
Listing 4.13 The definition of the interrupted_move function.

interrupted_move :: (entity, entity, entity, entity, entity) -> [completion]
interrupted_move(hero, villain, victim, donor, magical_object) =
  slist(
    start_move(hero, villain, victim, donor, magical_object),
    liquidation_move(hero, victim, donor, magical_object),
    liquidation(hero, villain, victim),
    ...
    wedding(hero, victim)
  )

possibility, a tale may contain a move that is interrupted. Finally, a tale may contain a
move that is interrupted with another move, which is in turn interrupted by another
move. All of these possibilities can be encoded in the tale function, provided in

Listing 4.14 The definition of the tale function.

tale :: (entity, entity, entity, entity, entity) -> [completion]
tale(hero, villain, victim, donor, magical_object) =
  random
    > full_move(hero, villain, victim, donor, magical_object)
    > slist(
      full_move(hero, villain, victim, donor, magical_object),
      full_move(hero, villain, victim, donor, magical_object)
    )
    > interrupted_move(hero, villain, victim, donor, magical_object)
    > slist(
      start_move(hero, villain, victim, donor, magical_object),
      interrupted_move(hero, villain, victim, donor, magical_object),
      resume_move(hero, villain, victim, donor, magical_object)
    )
  end

4.2.2.2 Dramatis Personae

We will re-use the same method for generating the dramatis personae that was ex-
plained in Section 4.2.1.2. Each of the characters will be generated randomly, and
then passed into the main tale function that we defined above.
4.2.2.3 A Note on the Model

The model created here only partially captures the types of folktales that Propp outlined in his morphology. Specifically, we have only captured three of the six move combinations listed in Section 4.1.2. In addition, we have not allowed for functions to be inverted, as explained in Section 4.1.3, nor have we modelled the omission of some of the functions within a move. Both of these phenomena can be modelled using our framework by adding additional choices to the appropriate function. For example, we could create an additional random selection choice to the `full_move` function which omits various functions.

A second problem with the model described above is encountered with tales interrupting tales. We will assume that each of the Proppian functions we have designed has query mechanisms built in to check the surrounding environment to make sure that the same villainy is not enacted in the space of an interruption. For example, if a princess is kidnapped, then the interruption of the tale should not contain a second instance of the princess being kidnapped. We do however, allow this to happen in instances where one move is allowed to fully complete before starting a different move. Again, proper use of a query tool will be able to check the surrounding environment to make sure such issues are not a problem.

4.2.2.4 A Partial Generation Example

Given the functions we have just defined, we now have the ability to create a story through functional expansion. The start of expansion is the same as seen in the previous section. The same sets of commands issued in Listing 4.7 can be issued to a generation environment loaded with the functions we defined in the previous sections. The start of expansion will result in a single node on the tree. Following that, one of the move structures will be selected at random. For this example, we will assume that `interrupted_move` is selected. The resulting tree structure from this expansion
is provided in Figure 4.12.

Figure 4.12: The semantic tree after tale is expanded.

Once the interrupted_move function has been chosen, it expands into the mandated choice that is in the body of its composition. This particular expansion made use of the start of a move with the start_move function, followed by liquidation_move, and then the resumption of the move that was interrupted. The state of the semantic tree after the interrupted_move is expanded is provided in Figure 4.13.

Figure 4.13: The semantic tree after interrupted_move is expanded.

Following the expansion of interrupted_move, the generation process turns to the content of start_move. The contents of this function are dictated by a single, mandated choice, which is an slist that contains the first sequence of moves that may occur in any given movement. The state of the semantic tree after the expansion of the start_move function is provided in Figure 4.14.
Figure 4.14: The semantic tree after \texttt{start\_move} is expanded.
Once \texttt{start.move} has been expanded into its \texttt{slist}, we turn to the \texttt{villainy} function. This function makes a random selection from each of the various villainies that Propp defined. In the case of this example, we will choose \texttt{villainy1}. In turn, this villainy function is expanded into its single, mandated choice, which is the familiar \texttt{kidnap} function, provided in Figure 4.15.

We are now in familiar territory concerning how the rest of the tree is developed. The \texttt{mediation} function will be expanded, looking back at the villainy that occurred to determine a correct expansion choice. From here, the \texttt{counteraction} function follows, and so on, until we have completely expanded each of the functions that occurred as a part of the \texttt{start.move} function’s \texttt{slist}. Generation then moves to expand the \texttt{liquidation.move} function, which works in the same way as the expansion of the \texttt{start.move} expression. The end result of generation is a much more complex tree than the simple example provided in the schema-expansion section. However, the basic mechanisms involved in both processes are exactly the same.

\section{Discussion}

The production of the simple folktale in Section 4.2.1.3 demonstrates how the Proppian morphology can serve as a source of functions and for the structure of folktales, and provides proof that our method of function expansion is capable of generating folktales. There are, however, a number of shortcomings that are attached to these processes as they currently stand.

\subsection{Lack of Preparation}

Propp gives special attention to two sections of the folktale known as the \emph{Initial Situation} ($\alpha$) and the \emph{preparatory section}, which includes the functions \textit{Absention} ($\beta$), \textit{Interdiction} ($\gamma$), \textit{Violation} ($\delta$), \textit{Reconnaissance} ($\varepsilon$), \textit{Delivery} ($\zeta$), \textit{Trickery} ($\eta$).
Figure 4.15: The semantic tree after villainy is expanded.
and *Complicity* ($\theta$). In these parts of the story, background information relating to various characters is introduced, and the events leading up to the main villainy are described. However, during the analysis of any given tale, Propp does not include these functions as part of the story, noting that several of the functions result in the same set of circumstances. His example is that of the conditions for a villainy to take place. If a kidnapping is to occur, it can come about if the hero violates an *Interdiction*, or if the hero complies with a villain’s demands (*Complicity*). The consequences of having both of these conditions present in a story were not well understood, and Propp’s suggestion is that more study is needed to determine how these situations behave across a wider number of stories. The result is that the stories we generate lack background information on the characters, as well as how certain circumstances came about (i.e. how did the victim let herself be kidnapped?).

### 4.3.2 High-Level Descriptions

The functions used for Proppian story generation are only very high-level descriptions of what is occurring in the actual story. While the human-level descriptions that have been written thus far provide a realization of the content of the story, the information that is conveyed by the semantic expressions themselves does not provide the level of detail that would make for a very interesting story. For example, while a semantic expression in our system may be \texttt{rescue(hero, princess)}, it is sorely lacking in the type of descriptive information that would be interesting for a reader. The description itself lacks vital information that occurs during the course of the rescue. What are the circumstances surrounding the princess’s incarceration? What does the hero do to rescue the princess? Does anyone try to stop him? Clearly the level of detail provided in this single semantic statement does not give a creative account of what actually occurred. Given that we wish to have a much more detailed account of certain actions in a story, more advanced methodologies of content generation are needed during the
expansion process.

4.3.3 Structure and Content Predictability

As we have noted, Pérez y Pérez and Sharples (2004) claim that a text generation system lacks c-creativity if both the content and the structure of the story can be predicted. In terms of content predictability, with the current generation methodology, the dramatis personae are chosen at random from a pool of available characters. Similarly, the exact semantic expressions that are produced during the expansion phase are chosen from a limited list of transformation options. Since both the cast of characters and the resultant actions they may engage in is known to the story designer, it is fair to state that the content of the story will be highly predictable. In terms of structure predictability, the very essence of the Proppian morphology, and story grammars in general, is based on the idea that the structure of the folktale is well-defined and predictable. Thus, the structure of the set of stories that will be produced using this methodology will be highly predictable.

4.3.4 Dramatis Personae and Random Expansion

An additional difficulty occurs with the selection of the dramatis personae. As mentioned above, the current practice is to generate the characters at random, and then insert them into the functional argument slots as necessary during the process of expansion. In this respect, each of the character identities (as well as some necessary donor objects, such as magical swords and ducks) is said to be stored within the root node of the semantic tree. Unfortunately, this practice can lead to some strange inconsistencies. For example, as previously noted, it would be unsuitable to select at random a princess who was an orphan all of her life, and insert her into a story where she suddenly lives in a castle with her father the king. Such events would confuse the reader, especially if no explanation is given as to how these particular circumstances
came into being.

A solution to this problem would be to wait until the end of the generation process, check the story content, and then choose a suitable princess from the list of available dramatis personae. However, this particular approach is still not good enough, since the generation process itself may create inconsistencies in the story that prevent any character from being selected (see Chapter 7 for an example called the dead brother scenario). One solution to this problem is to introduce more control mechanisms to the generation process to prevent these types of conflicts from occurring in the first place.

4.4 Summary

A review of Propp’s (1968) morphological analysis was provided. Although the work was not intended to be interpreted as a generative device, it does provide the formal structure of the elements in a folktale, which can be used for generation. Two different methods were proposed to generate folktales using Propp’s analysis. Both methods make use of a set of story objects and actions that are codified as semantic functions. The first generation methodology captures the story schemas that Propp uses as the basis of his analysis. When asked to generate a folktale, a selection is made at random from one of these schemas, and each function is successively expanded until a story results. The second generation methodology encodes the general progression of functions as a collection of different moves. Stories were generated by selecting various sets of move combinations. Limitations to both these approaches were discussed.
Chapter 5

Elaboration

The simple generation attempts in the previous chapter provided an illustration of how Propp’s (1968) morphology can be turned into a generative device that can be used within our semantic framework. However, generation of these simple stories demonstrated a number of problems with the formation of content. In particular, the lack of background information leading up to a set of story events, coupled with very high-level descriptions of our semantic functions results in unsatisfying stories. The purpose of this chapter is to examine the details of the problem more closely and propose a new strategy - elaboration - which can be used to enrich the existing semantic content of a story.

5.1 Elaboration Problems in Generation

Consider the instance where the generation environment has created a simple semantic tree, provided in Figure 5.1. A very simple English version of the story would be the following.

Once upon a time, a witch kidnapped a princess. A hero rescued the princess. The hero and the princess got married.
While the functions \textit{kidnap}, \textit{rescue} and \textit{marry} form the actual content of the story, they present very basic explanations of the actions that take place. For example, the \textit{kidnap} function where the villain kidnaps the princess lacks more descriptive narrative. This narrative, while not critical to our understanding of the story, provides commentary that makes the story more interesting. Without it, we are left asking questions surrounding the makeup of the actual events. As a human, we can effortlessly create the circumstances and easily add detail to make the story more interesting. One possible descriptive replacement for \textit{kidnap(witch, princess)} would be the following:

The princess lived in a castle, and cared for many different pets. One of her favourite pets was a pet rabbit that she kept in a cage in her bedroom. One day, while the princess was out riding a horse, the witch sneaked into the castle and let loose the rabbit. The witch chased the rabbit out of the castle and into the woods. When the princess returned to the castle, she was distressed to find her rabbit missing. The princess searched the castle from top to bottom, and when she could not find her beloved rabbit, she ran out into the woods to continue her search. The witch was waiting in a dark group of trees, near the path through the forest. When the princess came near the witch, she leaped out and wrapped the princess up in a bag. The witch ran back to her swamp with the princess in the bag.

This narrative sequence is but one way we could imagine this story unfolding. There are many others. Our goal is to capture how this type of semantic information
can be added to the tree. We will call this process *elaboration*.

## 5.2 A Brute-Force Solution

A simple way to implement elaboration is to endow each function with every single descriptive possibility that could occur. For example, the above description of the kidnapping could be represented as a set of semantic expressions. This set of expressions would form one choice as an expansion possibility in the `kidnap` function. All of the other possible descriptions could also be placed in the `kidnap` function, each one becoming a new expansion possibility. An example of how the `kidnap` function would look using this brute force solution is given in Listing 5.1. Here the `>` ... is used to indicate the fact that other sets of semantic expressions would be possible, but are not listed due to space considerations.

Adding these details to the `kidnap` function has an effect on the semantic tree, as demonstrated in Figure 5.2. When compared to Figure 5.1, we see that the tree was sparse before, but is now bushy, since many more expressions are involved in the expansion of a single function. This example provides a simple visual demonstration of the effect of adding additional detail to the functions in the lexicon. Trees of this type are exactly what we want to achieve during the course of generation. Had we created elaboration possibilities for the `rescue` and `marry` functions, then the tree would contain a proportionally larger number of semantic functions on it.

However, there are some issues regarding the process used to achieve this result. The solution is *brute force* in the sense that every single possible expansion under any possible context would need to be recorded in the `kidnap` function. While this certainly provides us a mechanism to add detail to a semantic expression, it has a few problems. First, since the set of contexts that exist in the world is a potentially open set, it would be impossible to specify every possible expansion for a given function.
Listing 5.1 An example listing of the `kidnap` function with a detailed expansion possibility.

```
kidnap :: (entity, entity) -> action
kidnap(villain, victim) =
  random
    > slist(
        live(victim, in(castle)),
        care_for(victim, pets[many]),
        love(victim, rabbit),
        keep(victim, rqual(rabbit, in(in(cage, REL), bedroom))),
        while(
          slist(
            go(victim, outside),
            ride(victim, horse)
          ),
          slist(
            sneak(villain, in(castle)),
            free(villain, rabbit)
          )
        )
    )
  return(victim, castle),
  cause(
    find(victim, qual(cage, empty)),
    feel(victim, distress)
  ),
  search(victim, qual(castle, all)),
  cause(
    find[neg](victim, rabbit),
    go(victim, forest)
  ),
  wait(
    villain,
    in(
      qual(
        trees,
        dark,
        near(rqual(path, in(REL, forest)))
      )
    ),
  ),
  cause(
    move(victim, near(villain)),
    slist(
      leap(villain, out),
      wrap(villain, victim, bag),
      run_to(villain, dwelling_of(villain))
    )
  )
  > ...
```
Figure 5.2: The simple story with an elaborated kidnap.
Even if we provide a limit on the number of contexts, for example, 100 different possibilities, the declarations of our semantic functions would become unwieldy. Clearly a more elegant method of achieving this type of detail is required.

Second, the elaboration of a particular function may violate the well-formed constraint of a story grammar. In essence, our brute force mechanism is not just supplying what happens during a kidnap action, it is also supplying the pre-conditions that need to be in place in order for the kidnapping to occur. We have previously stated that one of the purposes of the story grammar is to control where various sets of semantic expressions are introduced. This is exactly the case with simple Proppian stories. The purpose of the preparatory section of a folktale is to provide these details as a setup to the actual villainy that will occur. In the best case, it may be that the pre-conditions necessary for the kidnap function already appear on the tree. In such instances, we don’t want to add the full set of semantic expressions to the tree, we only want to add those statements that are necessary to explain the details of the kidnapping. However, if the necessary pre-conditions don’t already exist, we want the option of being able to add them to the tree in the correct locations as dictated by the story grammar. For these reasons, a better separation of the pre-conditions from the main body of the elaboration is required.

Third, and finally, in order for the expanded possibility to make sense within the confines of the current story, we need to make use of the query mechanism, so that we understand what is going on in the semantic world. Without this, random expansion may result in a meaningless plot line. For example, the story would not make sense if the semantic expressions specified that the witch was to hide out in the woods, while the introduction stated that the whole story takes place in the middle of a desert.
5.3 Fine Tuning

Is it possible to fine tune the brute force solution to remove some of the difficulties noted above? In essence, yes. Specifically, by making use of tree queries and separating out the pre-conditions from the actual body of the function composition, we can improve the composition of the functions. When we elaborate a function, we will only provide the details of what happens if a very specific set of circumstances already exist on the semantic tree. This removes the need for very large function bodies, since the necessary background details giving rise to a function are already assumed to have been provided elsewhere on the semantic tree.

5.3.1 Pre-conditions

As stated above, the first task is to separate the elements of the plot that give rise to a situation from the actions that may occur because of their presence. Consider the example of the witch, princess, and kidnapping given above. There are two distinct narrative intentions that are captured by the prose. The first is the establishment of the pre-conditions that give rise to the kidnapping situation:

The princess lived in a castle, and cared for many different pets. One of her favourite pets was a pet rabbit that she kept in a cage in her bedroom. One day, while the princess was out riding a horse, the witch sneaked into the castle and let loose the rabbit. The witch chased the rabbit out of the castle and into the woods. When the princess returned to the castle, she was distressed to find her rabbit missing. The princess searched the castle from top to bottom, and when she could not find her beloved rabbit, she ran out into the woods to continue her search. The witch was waiting in a dark group of trees, near the path through the forest.

The second narrative intention is to describe the actual effects of the kidnap function given our initial situation:

When the princess came near the witch, she leaped out and wrapped the
princess up in a bag. The witch ran back to her swamp with the princess in the bag.

Separating the pre-conditions from the event description has simplified what we need to specify in the \texttt{kidnap} function. It is no longer responsible for describing the complete set of conditions that give rise to its ability to perform the act. Instead, the expansion can simply query the existing semantic tree to see what conditions exist before producing a resulting effect. Using our witch and princess example, we can see that there are several pre-conditions that must be in place for this particular expansion of the \texttt{kidnap} function to take place:

- The witch is hidden.
- The princess is alone.
- The witch and the princess are in close proximity to one another.

\subsection*{5.3.1.1 Necessary Pre-conditions and Context}

It is important to note that the three conditions specified in this example do not necessarily reflect the full and necessary set of requirements for the \texttt{kidnap} function in every possible semantic context. This particular function may only be suitable for use in a very limited set of contexts, for example in a folktale. There may be a whole library of other kidnap-like functions that are appropriate for different types of stories, each with its own possibly unique sets of pre-conditions. For example, if the programmer wanted to make a version of the \texttt{kidnap} function with an instrument, one of the pre-conditions would be that the kidnapper must have had the instrument in their possession.
5.3.1.2 Chains of Pre-conditions

It may be tempting to argue that other pre-conditions are necessary, beyond the three listed above. For example: that the princess owns a rabbit, that the witch sets the rabbit loose, etc. While they are necessary for the story in its entirety, these pre-conditions do not belong on the body of the `kidnap` function. Rather, they belong to the set of functions needed to satisfy the pre-condition “the princess is alone.” In effect, a single pre-condition may in turn require other pre-conditions to be in place, forming a *chain of events*. Thus, in order for the princess to be alone, she must have left the castle by herself. She left the castle by herself to go searching for her pet rabbit. She decided to search in the woods for her pet rabbit because she could not find it in the castle. She searched the castle for her rabbit because she could not find it in its usual cage. She could not find her pet rabbit in its usual cage because it was freed by the witch. The witch freed the pet rabbit when the princess was out riding. Clearly these other semantic statements are necessary to satisfy pre-conditions of pre-conditions, and do not need to be stated in the body of the `kidnap` function itself.

5.3.1.3 Pre-condition Syntax

The form of our pre-conditions is indicated by the use of the `pre` keyword, an example of which is provided in Listing 5.2. Essentially, `pre` signals the generation system that the semantic tree needs to be checked for a particular expression. If found, then expansion may occur. If not, then the system will raise an error\(^1\). The first `pre` in the listing checks to make sure that the `villain` is hidden, the second checks to ensure that the `victim` is alone, and the third is a check to make sure that the `villain` is close to the location of the `victim`. If all three of these pre-conditions are in place,

\[^1\]Recovery from a pre-condition error is possible by adding the necessary requirements to the tree. This is discussed in Section 5.3.4.
then the function may be expanded into the specified slist.

**Listing 5.2** An example listing of the `kidnap` function that queries the semantic tree looking for specific pre-conditions.

```plaintext
kidnap :: (entity, entity) -> action
kidnap(villain, victim) =
    pre qual(villain, hidden)
    pre qual(victim, alone)
    pre near(villain, victim)
    slist(
        wrap(villain, victim, bag),
        run_to(villain, dwelling_of(villain))
    )
```

An arbitrary number of pre-conditions may be listed for a given function. In addition, we can specify which threads to look in, a distance from the current function, and any other limiters that are needed in order for the statement to make sense. In this example, we have left these restrictions off, simply for the sake of clarity. However, it is obvious that these limiters are needed so that only the relevant locations of the tree are searched. In this example, the pre-condition search could be restricted to include only the functions nearest to the `kidnap` function concerning the `villain` thread.

### 5.3.1.4 Guarded Choices as Pre-conditions

Guarded choices are also a form of pre-condition. The queries that are performed as part of a guarded choice relate to the conditions that must be in place in order for an expansion selection to be made. For example, assume that the witch is using magic to kidnap the princess. Listing 5.3 provides a modified version of the `kidnap` function that queries the semantic tree and selects an appropriate expansion based upon the spell that was cast.

In this listing, the `kidnap` function now requires that one condition is in place: that the `villain` casts a spell on the `victim`. This query is essentially a pre-condition
Listing 5.3 An example listing of the `kidnap` function that uses a guarded choice as a form of pre-condition.

```plaintext
kidnap :: (entity, entity) -> action
kidnap(villain, victim) =
  query spell1 : cast(villain, victim, *) in thread villain before current
    > spell1 == transport_spell:
      slist(
        disappear(victim),
        appear(victim, in(dwelling_of(villain)))
      )
    > spell1 == sleepwalk_spell:
      slist(
        sleep[vbegin](victim),
        walk(victim, to(dwelling_of(villain)))
      )
  end
```

for the expansion of the function, since the two expansion choices depend on the type of spell that was cast. If the witch casts a `transport_spell`, then the `victim` is immediately transported from her current location, to the house of the `villain`. If it is a `sleepwalk_spell`, then the `victim` falls asleep and begins to walk to the house of the `villain`. If the `villain` did not cast a spell at all, then expansion cannot occur. Since no default choice is provided, the system would raise an error.

5.3.1.5 Multiple Contexts

There is one aspect regarding the use of pre-conditions that has been omitted thus far. Specifying a set of pre-conditions is similar to establishing the fact that a very specific context exists on the semantic tree. However, there may be some instances where a function has a wide variety of expansion choices, each requiring a completely different set of pre-conditions and guarded choices. For example, recall that our original specification of the `kidnap` function contained three pre-conditions – the witch is hidden, the princess is alone, and the witch is near the princess. A later specification involved a guarded choice that defined a different pre-condition – the witch casts a spell on the princess. Clearly, the pre-conditions listed for the first
context are completely different from the pre-conditions required for the second. To accommodate this possibility, we need to make a change to our function compositions to allow for different sets of contexts. This new syntax is demonstrated in Listing 5.4.

**Listing 5.4** An example listing of a function that makes use of multiple contexts.

```plaintext
text

In this listing, the keyword `context` is used to indicate that the expansion of the `kidnap` function will be based on a variety of different contexts, each one identified by a different label. Here, `context 1` refers to the context where the witch must be hidden, the princess must be alone and the witch must be near the princess. The only expansion choice is where these three conditions are met. If the pre-conditions associated with this particular context are not supported by the environment, then the next context is examined. In `context 2`, if the witch cast a `transport_spell`,
then the victim disappears and reappears in the witch’s dwelling. If the witch casts a sleepwalk spell, then the princess falls asleep and creeps silently to the dwelling of the witch. If neither of these match, then the default context is selected, in which case the semantic expression results in a generic capture event.

Context names are simply labels. In the previous example, the labels 1 and 2 name the different contexts. However, any combination of numbers and letters are permitted for use as a context label. Thus, we could easily substitute the label physical_captur e for the first context, and magical_captur e for the second. It is important to note, however, that these context names carry no semantic significance.

5.3.1.6 The Default Context

There is one special context that may be attached to any semantic function called the default context. Within this context are a series of semantic expressions that can be used as the basis of the functional expansion that do not rely on any pre-existing environmental states. Consider Listing 5.5, which defines the function murder. In this listing, we see that there are \( n \) different contexts available. They are currently left unspecified, as indicated with ..., but are meant to contain a list of guarded choices, pre-conditions and semantic expressions. The context of interest is the context default, which specifies that a villain plans to kill someone, and then carries out the actual killing act.

During the course of generation, the default context is meant to provide a neutral expansion option. In other words, if none of the other contexts on the semantic tree are appropriate, the generation system may always use this default context as the expansion option, since it does not rely on any other information on the semantic tree. It is left up to the programmer to ensure that they provide a suitable set of expressions that can be used in the default instance.
Listing 5.5 An example listing of the murder function that contains $n$ different contexts, along with a default context.

murder :: (entity, entity) -> action
murder(villain, victim) =
    context 1:
        ...
    end
context 2:
        ...
    end
context n:
        ...
    end
context default:
    the_murder :: action
    the_murder = kill(villain, victim)
    plan(villain, the_murder)
    the_murder
end

5.3.2 Post-conditions

Related to the pre-condition is the notion of the post-condition. When any semantic function is evaluated, its result has an impact on the remainder of the story. Post-conditions come in the form of semantic expressions. They are not explicitly placed on the semantic tree, but rather are defined in the body of a function declaration, and are understood to implicitly contribute their semantic effects when the function is evaluated. Listing 5.6 provides a simple example of the give function, with the post-conditions it entails.

Listing 5.6 An example listing of the give function, with its set of post-conditions.

give :: (entity, entity, entity) -> action
give(donor, object, recipient) =
    pre possess(donor, object)
    post possess(recipient, object)
    post possess[neg](donor, object)
"donor gives object to recipient"
This example defines a function in which a donor gives an object to a recipient. The pre-condition identified by the pre keyword states that the donor must be in possession of the object. The post keyword is used to introduce the post-conditions. In this declaration, two statements are established as post-conditions: that the recipient is now in possession of the object, and that the donor no longer has possession of the object. These post-condition statements are understood to take effect whenever the give function is evaluated.

Post-conditions give us the ability to say more about the state of the world after a particular expression has been evaluated. For example, the function buy states that one person buys an object from another person. However, there is more that occurs within that function that affects the state of the world: the buyer will now have the object in his or her possession. Additionally, the buyer will have given some form of payment to the original owner in exchange for the object. All of this is common knowledge that humans have about a specific function, that does not get explicitly expressed on the tree, but needs to be captured in the function. Post-conditions are essentially optional statements of fact that are associated with a function.

5.3.2.1 Satisfying a Pre-condition with a Post-condition

Rather than querying the tree for the explicit appearance of a semantic expression, we can check the post-conditions on each function to see if it is supplied implicitly. This is interesting, since it allows us to use functions that indirectly create the semantic context that we are looking for. Consider the definition of two related functions provided in Listing 5.7.

In this example, there are two different functions defined, give and take. These functions are similar to one another, the difference being that in give, the donor

\footnote{Post-conditions behave like implications.}

\footnote{There are subtle issues in deriving the post-conditions when the function has certain adjustments, however, these will not be discussed here.}
Listing 5.7 An example listing of the **give** and **take** functions.

```haskell
give :: (entity, entity, entity) -> action
give(donor, object, recipient) =
    pre possess(donor, object)
    post possess(recipient, object)
    post possess[neg](donor, object)
"donor gives object to recipient"

take :: (entity, entity, entity) -> action
take(recipient, object, donor) =
    pre possess(donor, object)
    post possess(recipient, object)
    post possess[neg](donor, object)
"recipient takes object from donor"
```

initiates the action, where in **take** the **recipient** initiates the action. In both cases, the same set of post-conditions occur: the **recipient** has the **object**, and the **donor** does not.

When performing a check for a pre-condition, in addition to looking directly for the expression `possess(prince, magic_sword)`, we can look for functions that provide this expression as a post-condition. Thus, the functions `give(donor, magic_sword, prince)` and `take(prince, magic_sword, donor)` would be acceptable. With post-conditions, we are no longer required to have `possess` be explicitly expressed on the tree.

### 5.3.3 The Pre-Condition Problem

Section 5.3 outlined the need for a separation between a set of semantic statements and the pre-conditions that must be in place in order for an expansion choice to be made. During expansion, the generation process now looks to the semantic tree for a specific semantic context, and will add appropriate amounts of detail as a replacement to a function, depending on what it finds. In our example, while this particular implementation has removed some of the creative burden from the **kidnap** function, it appears that we have simply shifted this problem to a different part of the generation
process. In order for elaboration to work, we now require that there be enough detail on the semantic tree to give rise to the specific contexts that the *kidnap* function requires. How this content comes into existence is a problem that we will call the *pre-condition problem*.

At one end of the spectrum, we could assume that this creative information is always available on the semantic tree. This is a valid option, as it is possible that the story grammar will have added a number of topoi to the tree which are responsible for generating introduction and backstories for our characters and events. However, it may also be the case that very few of the required pre-conditions for a function appear on the semantic tree, leading to stories which are no better than the types we are currently capable of producing without the elaboration mechanism.

An alternative solution would be to supply these necessary pre-conditions as parameters to the generation process. During the course of generation, they could be inserted into the semantic tree where appropriate. However, this solution is also rather unsatisfying, since this type of parametrization would result in a predictable story. The programmer, in effect, would be in complete control of which semantic expansion they wish to appear.

5.3.4 Tree Modifications

The solution to the pre-condition problem is to modify the semantic tree so that it holds the necessary contexts to allow for elaboration. In essence, this strategy amounts to *making up* the set of descriptions that will eventually fulfil the requirements of a pre-condition only when we encounter a function that requires them. This strategy can be realized by making use of two different processes. The first involves checking the tree for locations where additional detail can be added or changed to accommodate the required context. These locations are known as *hooks*, since we will use them as anchor points where we will *hang* additional content. The second process
involves adding the necessary set of expressions to the tree at the current location.

### 5.3.4.1 Searching for Hooks

Hooks come in two different forms: unspecified arguments, or unexpanded functions on the tree. An example will clarify this\(^4\). Let us assume that the generation process has created a tree such as that shown in Figure 5.3, which relates to the simple story where a witch kidnapa princess, and the hero rescues her. However, in this example, we are assuming that the generation process has not yet expanded the conclusion function.

![Figure 5.3: A simple semantic tree relating to a story where conclusion has not yet been expanded.](image)

The conclusion function is defined with a number of pre-conditions that must be satisfied before an expansion choice may be made. Listing 5.8 provides an example of the possible expansions. In this example, we assume that there is more than one context available in the query list; however, we have chosen not to list any of the others for the sake of clarity. The context that we are interested in assures that the hero marries the princess and inherits the kingdom, as shown in context 1.

The set of semantic expressions in context 1 essentially says that the hero will marry the victim, and inherit the kingdom. Clearly, context 1 has a pre-condition that the king of the kingdom must already be dead. While we could certainly add

\(^4\)While the kidnap function provides a simple introduction to the elaboration problem, demonstrating the need for hooks and tree additions requires a separate, more complicated example.
Listing 5.8 An example listing of the conclusion function with a large number of detailed expansion possibilities.

class conclusion :: (entity, entity) -> action

conclusion(hero, victim) =
  context 1:
    query status1 : *(father_of(victim))
      > status1 == die || status1 == die[past]
        slist(
          marry(hero, victim),
          inherit(hero, belong_to(land, father_of(victim)))
        )
    end
  end
...

die(father_of(princess))\(^5\) as a branch in the semantic tree prior to encountering this context, this is not a particularly creative solution, since it lacks the type of descriptive content that would explain how the king came to be dead.

There are two hook-based processes that we can use to correct this problem. The first works by searching the semantic tree for areas where the function we require already exists but uses an unspecified argument. In this particular example, we are looking for the function die(unspec) to appear on the tree explicitly, or as a post-condition to a function that appears on the tree. If it exists, it may be possible to change the unspec entity to father_of(princess) resulting in die(father_of(princess)), which is the required pre-condition for context 1 to occur. However, there are consequences associated with this strategy. The semantic tree must be checked for continuity after making this change. This is done to ensure that the function die(father_of(princess)) is consistent and makes sense in that area of the story. If not, then the generative process must apply a continuity fix as discussed in Chapter 7, to make sure the events of the story make sense.

The second process involves checking the semantic tree for unexpanded topoi. Recall from Section 3.3.3 that every topos has a summary attached to it that the

\(^5\)In our story, the formal parameter victim will be replaced with princess.
generation process may use instead of expanding the entire function. Under these circumstances, it is possible that the die function is on the tree, but is not expressed in the story since the generation process chose not to expand the topos that contains it. This unexpanded topos becomes a potential hook for our required content. For example, assume that one of the expansions of the function kidnap is the following:

> Early one morning, the princess and her father went out for a walk in the forest. The witch, hiding in some nearby bushes, cast a death spell on the father of the princess, causing him to die. The princess, trying to revive her father, was unaware of the witch. The witch cast a sleeping spell on the princess, and carried her away to her lair.

This particular expansion of the kidnap function contains the function we are looking for, namely \text{die(father\_of(princess))}. If the kidnap function was originally left in its unexpanded form on the tree, then the generation process can go back and expand it, thereby expressing the required set of functions.

5.3.4.2 Tree Additions

If there are no suitable hooks available on the tree, then a second method, tree additions, can be used to add the required context to the tree at the location where the information is required. To see how this second method works, consider a simple example of a topos called family\_tragedy, whose purpose is to describe an accidental or volitional killing of a sibling, and how this effects a set of family relationships. Figure 5.4 demonstrates the expansion variations involved in the family\_tragedy topos. In this figure, each box contains the composition of a function declaration. The arrows connecting each box demonstrate the expansions that may occur. Dotted lines indicate that a choice exists between one or more functions, meaning that only the one branch will be selected during expansion.

At the top of the tree, the function family\_tragedy is defined as a random selection between the topos a\_kill\_family\_grow, in which a sibling accidentally
Figure 5.4: Possible expansions for the topos `family_tragedy`.
kills his brother causing their entire family to grow closer together, or the topos \texttt{v_kill_family_feud}, in which a sibling purposefully kills his brother, causing the family to hate each other. The topos \texttt{a_kill_family_grow} contains a cause statement, which states that an \texttt{accidental_kill} causes \texttt{family_grow_closer}. The \texttt{slist} in the \texttt{accidental_kill} represents the following narrative:

Two brothers, \texttt{x} and \texttt{y}, are out hunting on horseback in the woods one day. Deciding that they need a break, the brothers stop at a river to rest their horses. \texttt{y} dismounts from his horse, and takes a drink from the river. A ferocious boar runs out of the forest toward \texttt{y}. \texttt{x} quickly readies his bow and nocks an arrow. \texttt{x} aims at the boar. \texttt{x}'s horse rears causing \texttt{x} to shoot \texttt{y}. \texttt{y} dies.

Recall that our purpose is to modify the semantic tree such that \texttt{die(father_of(princess))} is expressed. Clearly the expansion of the \texttt{family_tragedy} function contains the statement we are looking for - \texttt{die(y)} - as long as we are able to substitute \texttt{father_of(princess)} for the parameter \texttt{y}. Assuming this is possible (see Section 5.3.4.3), we can place \texttt{family_tragedy} on the tree before the expansions belonging to the \texttt{conclusion} function, as depicted in Figure 5.5.

Figure 5.5: A simple semantic tree relating to a story where \texttt{family_tragedy} has been inserted before \texttt{conclusion} to satisfy pre-conditions.

In this example, we have created an \texttt{slist} in which the \texttt{family_tragedy} function appears first, followed by the \texttt{conclusion} function. The two \texttt{unspec} entities relate to
the formal parameters $x$ and $z$ in the `family_tragedy` function, are left as unspecified individuals that can be used later on in the generative process as potential hooks, if needed. The `father_of(princess)` function relates to the formal parameter $y$ in the `family_tragedy` function, and is the entity who is eventually killed. With this new addition to the tree, the generation process expands `family_tragedy` until the required expression - `die(father_of(princess))` - is on the tree. This is shown in Figure 5.6.

Now that the desired pre-condition exists on the semantic tree, the `conclusion` function can be expanded. Recall from Listing 5.8 that the context we desired is `context 1`, which requires that the father of the princess be dead. Given that this set of conditions now exists on the tree, we may use that context as our expansion choice, and add the fact that the prince marries the princess and inherits the kingdom to the semantic tree. This is shown in Figure 5.7.
Figure 5.6: A semantic tree relating to a story where family_tragedy has been inserted before conclusion to satisfy preconditions.
Figure 5.7: The semantic tree after the expansion of \textit{conclusion}.
5.3.4.3 Issues When Using Hooks and Tree Additions

When searching for unexpanded topoi, or when performing tree additions, we omitted two important details. First, we must be able to exert some control as to how a function is expanded. For example, when we are examining the \texttt{kidnap} function, we need to ensure that the choice containing \texttt{die(father\_of(princess))} is actually placed on the tree, rather than some other context which does not contain this expression. In other words, we need to \textit{guide} the expansion process to make sure this set of functions is actually expressed on the tree. Second, and perhaps more importantly, we need to know what set of functions may appear in a specific topos. For example, if we need to express \texttt{die(father\_of(princess))}, we need to know which topoi actually contain that particular function, otherwise we will not know which topos to choose from, or which unexpanded topoi on the tree are candidates for our expansion process. The details of both of these issues are discussed in the next section.

5.3.5 Pre-computing Topoi Expansions

One way to determine the list of functions a topos may contain is to pre-compute every single derivation that can occur in the expansion of a topos and then migrate the list of semantic expressions to the root of the topos tree. For example, Figure 5.8 contains the list of all the semantic expressions that occur in the expansion of the topos \texttt{family\_tragedy}. Each function has a section labelled \textit{contents}, which is a complete collection of semantic expressions found in the current function, as well as any expressions that may occur as a result of its expansion.

While other methods of determining function inclusion are possible, moving functions to the root of a topos tree has several advantages. First, we can easily search through the \textit{contents} section of a topos if we are looking for a particular function. For example, if we are looking for the set of all topoi that contain a \texttt{die} function, we simply search through our library of functions and check to see if the \textit{contents}
Figure 5.8: The tree of semantic expressions that can be built from the topos family_tragedy. Each function is labelled with a complete list of semantic expressions that it may possibly give rise to.
section contains what we are looking for. Second, we can use the functional annotation generated during migration to guide us in the expansion process. For example, let us assume that we were looking for the statement `forgive(john, bob)`. We would search the root of the topos `family_tragedy` to see if that expression occurs, which it does: `forgive(z, x)`. Now, having found it, we need to expand the tree in such a way that we guarantee that `forgive(z, x)` will occur. Starting at the root of `family_tragedy`, we see that we have a choice between `a_kill_family_grow` and `v_kill_family_feud`. We check the root of both functions to see which one still contains the semantic effect `forgive(z, x)`. In this case, we see that only `a_kill_family_grow` still contains the effect, so we choose that one to expand. We continue with this process until the function we are looking for has been expressed.

This guided process becomes more difficult when attempting to accommodate the requirements of more than one expression. For example, assume that there are two statements that need to be expressed: `forgive(john, bob)` and `stab(bob, fred)`. We are looking for a single function in which these two statements appear. Both of the functions `forgive(z, x)` and `stab(x, y)` are listed as appearing within the topos `family_tragedy`. Assuming we can replace `z` with `john`, `x` with `bob`, and `y` with `fred`, this topos appears to be a match. However, when we attempt to expand it following the method described above, we run into a problem. If the generation process chooses `a_kill_family_grow`, then the statement `stab(x, y)` will not be expressed. However, if the generation process chooses `v_kill_family_feud`, then the statement `forgive(z, x)` will not be expressed. Thus, the generation system will conclude that the topos cannot express the required pre-conditions, and will select an alternate one from the list of candidates. If there are no other suitable topoi, the system will raise an error.

While adding a `contents` section and pre-computing each possible expansion of a topos provides us with an ability to search, the resulting functions are not elegant.
All of the semantic expressions of a topos end up migrating to the root of the topos. If we view the possible set of expansions as a tree, similar to what we did in Figure 5.8, we have in effect flattened the tree. Worse yet, if we allow for circular topos inclusion (i.e. topos $x$ includes $y$, $y$ includes $z$ and $z$ includes $x$), we literally cannot compute every single derivation, since we could end up with an infinite number of possible expansions. We therefore need another method of determining what functions a topos may express.

### 5.3.6 Tagging Topoi

Rather than pre-compute all of the derivations for each topos, we can instead tag our semantic functions with the list of topoi they appear in. Consider the example of *family_tragedy* from above. Assume that we want to know which sets of functions *forgive* occurs in. We would attach a list of pointers to the *forgive* function, where each pointer would point to a function that *forgive* appears in. In this case, *forgive* would have a pointer to *family_grow_close*. It would also have a pointer to *a_kill_family_grow*, since *family_grow_close* appears as one of its expansion choices, and it can expand to express the function *forgive*. In turn, *forgive* would also have a pointer to *family_tragedy*, since it has an expansion path that could give rise to the *forgive* function. We would do this same type of topos tagging with every other function in the semantic lexicon. To understand this process better, consider how we read in the contents of the lexicon:

1. When reading in functions from the semantic lexicon, we start with the most basic functions first. These are functions that do not have any other functions as part of their composition, such as semantic individuals like *juliet* or simple functions like *forgive*. 

143
2. Once we finish processing these basic functions, we can start processing functions which are composed or expand into sets of other, previously defined functions. For example, after reading the functions `forgive` and `love`, we could then read in the function `family_grow_closer`.

3. When we read the function `family_grow_closer`, we know that it can expand to give rise to `forgive(z, x)`. Therefore, the function `forgive` would have a pointer to the function `family_grow_closer`, since it can appear inside of it.

4. When we read the function `a_kill_family_grow`, we see that part of its composition includes the function `family_grow_closer`. Upon learning this, we would search all of the lexical entries that have pointers to `family_grow_closer` and update them so that they also include a pointer to `a_kill_family_grow`. Since `forgive` is a part of `family_grow_closer`, it would receive another pointer to `a_kill_family_grow`.

5. We repeat this process until we have read in all of our semantic functions.

When we have finished reading in the lexicon, each function will be annotated with the list of topoi it appears in. Figure 5.9 provides a graphical representation of how the function `forgive` would look, with pointers to its relevant topoi.

While there may be other ways of identifying functions in topoi, this particular tagging method has several advantages. Once the functions are tagged, finding a topos with the expression that we want is trivial. Rather than searching amongst all of our topoi, we instead consult the function that we are looking for, and make an appropriate selection from its list of topos pointers. For example, if we need a list of topoi that contain the function `forgive`, we simply consult the topos list that is attached to `forgive`. In this instance, we would get back the topos `family_tragedy`, `a_kill_family_grow` and `family_grow_closer`. This same tagging method can be used when a search needs to find two or more functions that appear in the same
Figure 5.9: An example how semantic functions are tagged with topoi. After reading the lexicon, the function **forgive** has been tagged as appearing in the functions **family_tragedy**, **a_kill_family_grow** and **family_grow_closer**.

topos. For example, assume we need to find a topos which has both **forgive** and **hunt**. We consult the lists attached to both **forgive** and **hunt** and make note of the topoi that are common to both. In this case, the list of applicable topoi will be **a_kill_family_grow** and **family_tragedy**.

### 5.3.6.1 Some Additional Consequences of Tagging Topoi

Tagging functions with topoi gives rise to a number of interesting side effects. As we saw before, our search for appropriate topoi is reduced to a trivial operation. We simply consult the function that we need to have appear in our semantic tree, and then select at random one of the topoi from its list. More interesting however, is the fact that we now have a new way of relating semantic functions to one another. For example, consider the topos **volitional_kill** in the example above. Our high level summary of this function is simply **kill(x, y)**. Additionally, there are two ways this topos can be expanded: **murder_stab** and **murder_poison**. The effects of these functions boil down to **stab(x, y)** and **poison(x, y)**. Since both functions
stab and poison will be tagged as appearing in volitional_kill, we now have a rudimentary association between these functions and the function kill. Even better is that we get this information “for free” as a byproduct of the tagging process. Section 8.2 discusses some of the possibilities relating to the exploration of these semantic relationships.

5.4 Controlling Expansion

Clearly, we don’t want to continue to elaborate ad infinitum. At the same time, we want to make sure that some of the expressions on the semantic tree are elaborated to a much richer extent than was previously possible. To allow for this difference in requirements, we need to provide some general elaboration controls: elaboration attributes, context selection, maximum tree depth specifiers, optional pre-conditions, and an interactive generation environment.

5.4.1 Elaboration Attribute

One way of controlling expansion is to have an elaboration attribute attached to each expression. The value of the attribute would be either true or false. A value of true would tell the generation system to elaborate the expression, thus putting into motion the series of events discussed throughout this chapter. A value of false on the other hand, would tell the generation system not to elaborate the expression. In such instances, rather than make use of the set of expansions possible, the system would instead use the high-level summary (see Section 3.3.3).

The elaboration attribute may come to be set in two different ways. During the generation of a story, the default value for the attribute will be set to true for all expressions on the tree. This means that any expression on the tree is a candidate to undergo elaboration. Alternately, the programmer may wish to designate
some expressions that may have elaborations, and some that may not. For example, consider Listing 5.9 where \texttt{context 1} contains a list of expressions that detail the events of a murder. The programmer has used an elaboration attribute, indicated by \texttt{elaboration=false} attached to the \texttt{kill} function, to prevent further expansion.

\textbf{Listing 5.9} An example listing of the \texttt{murder} function, where elaboration attributes are used to control further expansions of the set of functions in \texttt{context 1}.

\begin{verbatim}
murder :: (entity, entity) -> action
murder(villain, victim) =
  context 1:
    query occupation : is(victim, *)
      > occupation == king
      slist(
        say(villain, victim, hate(villain, victim)),
        kill(villain, victim) : elaboration=false
      )
  end
end
...
\end{verbatim}

\section{5.4.2 Context Selection}

A second method used to control elaboration is a \textit{context selector}. The point of the selector is to allow the programmer some form of control over the context that appears during the course of generation, rather than letting the system choose one at random. To do so, the programmer simply names the context after the expression. For example, Listing 5.10 contains the definition of a topos where contexts for various expressions have been specified.

This example demonstrates a simple function called \texttt{tale}, which is expanded into three different functions: \texttt{kidnap}, \texttt{rescue} and \texttt{livehappily}. The \texttt{context=physical\_capture} attached to the \texttt{kidnap} function tells the generation process to use the context \texttt{physical\_capture} as the context of choice when performing an elaboration. Similarly, \texttt{context=challenge\_villain} and \texttt{context=inherit\_kingdom} refer
**Listing 5.10** An example listing of a function `tale`, which uses context selection to guide the generation process.

```haskell
tale :: (entity, entity, entity) -> [completion]
tale(villain, victim, hero) = slist(
    kidnap(villain, victim) : context=physical_capture,
    rescue(hero, victim, villain) : context=challenge_villain,
    livehappily(hero, victim) : context=inherit_kingdom
)
```

to contexts that should be used during the elaborations of `rescue` and `livehappily`, respectively.

There are some additional details surrounding context selection. First, if a specified context does not exist for a given function, then the generation system will use the default context instead. This is to prevent a random selection from one of the other available contexts, all of which may be unsuitable for what the programmer intends. Second, we can develop a method of specifying lists of contexts so that a programmer may specify more than one appropriate context. To construct a list, the programmer need only use a `+` to combine contexts. For example, `context = spell_cast + physical_capture` states that these two contexts are valid choices. When provided with more than one context, the generation system will choose one at random.

### 5.4.3 Depth of Expansion

A third method of controlling elaboration comes in the form of restricting how many elaborations may occur during the expansion of a specific function. To understand how this works, consider the semantic tree provided in Figure 5.10. With this tree, we have a simple folktale that has a `villainy`, a `heroic_act` and a `conclusion`.

Each of these three functions is labelled with a depth restriction: a number that specifies how deep elaboration may proceed. When elaboration occurs, a new depth
Figure 5.10: A simple semantic tree relating to a story where the depth of elaboration is controlled.

restriction is calculated (current depth minus one), and is passed on to the children of the elaborated expression. If the depth associated with an expression reaches zero, then the expression is not allowed to undergo further elaboration. For example, assume that the 

villainy function above expands into a 

kidnap function, which in turn expands into a 

cast_spell function and a number of other functions which we will leave unspecified. This is depicted in Figure 5.11. Here we see the effects of the elaboration depth being passed down from each level of the tree. When the tree expands to have the 

cast_spell function, the depth has reached zero. This means that the 

cast_spell function cannot undergo elaboration, and instead must make use of its high level summary (see Section 3.3.3). In this way, we can limit the amount of information that is available at each node.

5.4.4 Optional Pre-conditions

It can be desirable to leave the details of some pre-conditions unspecified, as long as the surrounding context does not cause a conflict with the set of information they would entail. For example, consider the definition of the function 

provision_of_mobj given in Listing 5.11.

The body of this function makes no queries to the semantic tree, and simply
Figure 5.11: A simple semantic tree relating to a story where the depth of elaboration is propagated through the tree.

**Listing 5.11** An example listing of the `provision_of_mobj` function.

```
provision_of_mobj :: (entity, entity, entity) -> action
provision_of_mobj(donor, hero, magical_object) =
    give(donor, hero, magical_object)
```

results in an expansion in which the donor gives the magical_object to the hero.

What is left unsaid, but assumed to be true, is that the donor actually has the magical_object in his immediate possession. This particular condition is obviously necessary for the function to make sense. However, as long as the surrounding context does not contain anything that would be contrary to it, we do not have to explicitly say anything about it when we expand the function. In essence, we are allowed to omit the details of the pre-condition.

There is however, still an obligation to list the pre-condition as being necessary for the function to exist, regardless of whether or not specific details are optional. To do so, we need to introduce a new attribute to our pre-condition syntax to indicate that a pre-condition is attached to a specific function, but does not necessarily need to be explained in detail as part of an elaboration. The example provided in Listing 5.12 makes use of this new mechanism.

The attachment of the `optional` keyword states that the generation process does not need to create an explanation as to how this particular condition came to be.
Listing 5.12  An example listing of the provision_of_mobj function.

provision_of_mobj :: (entity, entity, entity) -> action
provision_of_mobj(donor, hero, magical_object) =
    pre possess(donor, magical_object) : optional
    give(donor, hero, magical_object)

Leaving the optional tag off of the statement would tell the generation system that it needs to create an explanation for the pre-condition. It would use the techniques discussed throughout this section to do so.

5.4.5 Interactive Elaboration

A final method of controlling elaboration comes from the idea of an interactive generation environment, where the user has direct input as to which expressions should undergo the elaboration process. This type of interactivity in telling stories was first seen in systems such as TALE-SPIN by Meehan (1976). In its simplest form the generation process would simply prompt the user before expanding each expression, and would provide an interface for the user to control whether or not elaboration is to occur. In this way, a user would have complete control and freedom to expand the details of the expressions he or she wished to know more about.

5.5 Discussion

Given the elaboration mechanisms just discussed, we are now able to produce more complex stories that provide a great amount of detail. However, some important points require consideration.
5.5.1 Differences from MEXICA

In some ways, the elaboration mechanism appears to function similarly to the coherence check in MEXICA’s (Pérez y Pérez, 1999) reflection stage. In this work and in MEXICA, pre- and post-conditions are used to ensure that the logical progression of events in a story are maintained and are explainable. Should an expression have a set of pre-conditions that need to be satisfied but do not yet exist in the story context, then a function is selected from a list of logical candidates, and is inserted into the story as an explanation. There are, however, fundamental differences that make this work unique.

First, pre-conditions and post-conditions in MEXICA are necessarily restricted to a limited set of effects relating to emotion, location, and tension. In this work, pre-conditions and post-conditions have no such restrictions, allowing for a much richer set of presuppositions and implications to be expressed.

Second, MEXICA inserts the action that creates the required pre-condition directly before the action that requires it. While this work proposes a similar mechanism, it also provides an alternative. Hooks can be used to introduce necessary content to a location defined elsewhere on the semantic tree, thereby weaving the requirements of a particular pre-condition into already established portions of the story.

Third, the MEXICA system was meant to operate in an unrestricted environment with broad rhetorical guidelines, using post-conditions to direct content generation to shape the actions a character may perform. In comparison, this work focuses specifically on how content may be added within the confines of a rigidly defined structure, and provides methods to add content that conform to the demands of that structure.

Fourth, and finally, the actions defined in MEXICA cannot be decomposed or combined to create other actions. There is no mechanism to define actions that are
composed of other actions. The framework used in this work is based on scalability, essentially allowing complex functions to be composed of other sets of functions. This has an important implication in terms of the satisfaction of pre- and post-conditions, and requires more care to be taken in the selection of a valid function. In effect, since we wish to add enriched details to a story, it is not enough to search for a function that has a composition that explicitly matches the required pre- or post-conditions. For example, we do not want to simply select \texttt{die(king)} when we want to describe how the King died. High-level functions provide us this enriched detail, however, their composition must be dissected, and in turn, each sub-function must be searched to see if it provides the desired effect. This work proposes mechanisms that are capable of performing this type of search. To ease the burden of searching through function compositions, each function contains a list of the topoi it is contained in (see Section 5.3.6), making selection of a high-level function easier. Directed generation (see Section 5.3.4.3) then ensures that the chosen high-level function is expanded such that it produces the desired effects.

### 5.5.2 Tagging Pre- and Post-conditions

We can extend tagging to encompass pre- and post-conditions. This becomes useful during the process of interpolation, where we need to search for functions that contain certain pre- and post-conditions. Recall that we have already tagged each function with a list of topoi it appears in. We can create two additional lists for each function, one that contains the list of topoi that it appears in as a pre-condition, and another that contains the list of topoi that it appears in as a post-condition. Listing 5.13 provides an example of the function \texttt{forgive} with its associated lists.

In this example, the function \texttt{forgive} is tagged as appearing in the topoi \texttt{family_tragedy}, \texttt{a_kill_family_grow} and \texttt{family_grow_closer}. The keyword \texttt{pre_list} is used to identify all of the functions that \texttt{forgive} is a pre-condition in. With this
Listing 5.13  An example listing of the `forgive` function, with its topoi, pre- and post-condition lists.

```plaintext
forgive :: (entity, entity, action) -> action
forgive(person1, person2, act) =
  "person1 forgives person2 for performing act"
topoi_list: family_tragedy, a_kill_family_grow, family_grow_closer
pre_list: forget, love
post_list: family_grow_closer
```

example, the function `forgive` is a pre-condition for the functions `forget` and `love`. Finally, the `post_list` is used to identify all of the functions that `forgive` is a post-condition of. In this case, `family_grow_closer` is the only one.

### 5.5.3 Elaboration is Expansion

To be clear, the core process of elaboration is nothing more than semantic expansion. The main difference that elaboration introduces is the ability to add new detail to the semantic tree in order to fit the needs of certain pre-conditions. Whereas in the simple Proppian folktale generation we were restricted to only running queries, the elaboration mechanism allows us to update the tree and place the desired background information directly into the story. In effect, the elaboration mechanism can make up the set of conditions that allow for a particular expansion choice to be made.

### 5.5.4 Continuity Conflicts

Given the fact that we can now insert information onto the semantic tree, we are now open to the possibility that the story may contain certain pieces of information that are in conflict with one another, creating inconsistencies. A new mechanism is now needed to determine whether such continuity conflicts arise on the tree, and if they are present, to provide some generative fix that would restore the consistency and continuity of the story. This is explored in Chapter 7.
5.5.5 Narrative Speed

As discussed by Hume (2005), *narrative speed* in works of fiction refers to “the narrative being accelerated beyond some safe comprehension limit.” The effect of this acceleration is to provide the reader with the sense that events in the story world are unfolding very quickly. This sense of speed is present even if the reader carefully reads the story, paying attention to every detail. It is usually through the deliberate lack of elaboration that the novel achieves a certain degree of speed.

Hume discusses three methods to increase narrative speed: *multiplying units*, the *subtraction* of expected material, and *fantastic actions*. Multiplying units uses the introduction of a large number of characters, events, or plot elements to invoke a sense of speed. In essence, the continual introduction of story elements and constant shifts of focus produce a sense of rapidity. Subtraction works by leaving out key logical details, which instills in the reader a sense that some important piece of information has been missed. Finally, fantastic actions introduce anomalies to the story which appear to be significant, yet have no logical explanation. The effect causes the reader to wonder if they have missed reading some important fact.

Using the elaboration controls discussed throughout this chapter, as well as the use of reader chronologies, it is possible to produce some of these effects. For example, subtraction can be achieved by marking a large number of pre-conditions as optional and preventing the elaboration of complex functions. The overall effect is that many of the logical details a reader would expect to see would not be expressed on the tree, resulting in the feel of an accelerated pace. Similarly, using elaboration to introduce a large number of details regarding somewhat irrelevant actions, in combination with a chronology that weaves quickly through these elements, can provide a multiplying effect.
5.5.6 Creativity

The elaboration mechanism does not alter the overall structure of the story, and as such, the story structures remain highly predictable. For example, our simple stories contained a villainy, a heroic act and a conclusion. While the amount of detail given with these items is subject to change, and in some cases may be very different from one another, this story structure is rigidly determined.

5.6 Summary

The concept of elaboration was introduced and defined as a method to enrich the semantic detail of a function through the use of random and guarded expansion choices. The set of functions described in Chapter 3 was expanded in different ways depending on the context provided by the semantic tree. Function declarations were enhanced to contain a set of pre-conditions and post-conditions that express these contextual requirements. Normally, if the pre-conditions of a function cannot be satisfied, then expansion will not occur. However, the elaboration process may make additions to the story content through the use of hooks or tree modifications in order to create the required context. Both mechanisms require finding topoi that contain the necessary context. Five mechanisms were proposed to control the extent of elaboration.
Chapter 6

Interpolation

While elaboration is sufficient to add new content to the story in areas where we desire more descriptive narrative, the generation process still suffers from structure predictability. This is because a story grammar, such as seen with Proppian folktales, is employed to provide the main structure of the story. Unfortunately, story grammars by their nature are meant to be predictable and repetitive. Thus we need an additional mechanism that operates on the structure produced by such a grammar in order to introduce creativity. A new mechanism - *interpolation* - will provide the means of modifying an otherwise rigidly produced story structure, and will result in new stories that could not be created by the story grammar alone.

6.1 Interpolating Between Plot Points

The theory behind interpolation is simple: between any two points of a plot, additional detail can be added. This detail can be as simple as adding descriptions as to how a particular object or person came to change location, or it can be an entire story that is meant to bridge the gap in between two major acts which take place in a play. This is best demonstrated with an example. Consider a portion of a semantic tree formed from a Proppian style folktale, shown in Figure 6.1.
This figure tells a portion of a Proppian folktale in which a donor, who will remain unspecified aside from the function donor, gives a task to the hero, the prince, in exchange for a magical object. Here, the donor tasks the prince to spend three years as a merchant. The prince agrees and completes the task, after which the donor gives the prince a magic sword in the topos provision_of_mobj.

To begin interpolation, we simply take two functions on the semantic tree, for example the hero_reaction and provision_of_mobj, and determine the set of functions that we are allowed to insert in between them. Allowable functions do not violate the set of active and dead links (see the next section for details) that may be in effect in the surrounding context. Currently, we choose a function at random from the set of allowable functions, however, there may be instances when a non-random approach is required. For example, we may wish to restrict selection based on a particular theme. While this presents an interesting problem, the purpose of this work is to broadly demonstrate the overall process of interpolation. This type of selection refinement is left for future work.
6.1.1 Active and Dead Links

*Links* occur between two functions. They are a visual depiction of whether a pre-condition has been satisfied by a semantic expression elsewhere on the semantic tree. If a pre-condition has been satisfied, an *active link* is formed between the expressions, indicated by a dotted line that joins the two together, labelled with the condition of the link. If a pre-condition has not been satisfied, it forms a *dead link*, indicated by a labelled dotted line that is left open at one end, and is connected to the pre-condition at the other. Dead links may also occur if a post-condition or semantic expression is not being used to satisfy a pre-condition. Each link has a *direction* toward which it points. Links originate at the semantic expression that satisfies the pre-condition, indicated by a circle, and point toward the pre-condition, indicated by an arrow. Before interpolation may occur, the generation process must check the set of links stemming from each expression, and sort out which are active and which are dead.

To illustrate active and dead links, consider the functions that appear in our example. The function `hero_reaction` contains a single post-condition, namely that (i) the hero has fulfilled the requirements stated by the donor. For the function `provision_of_mobj`, we will assume that there are three pre-conditions: (i) that the donor and the hero are in the same geographical location; (ii) that the donor has the object in question; and (iii) that the hero has fulfilled the requirements that the donor originally stated. The active and dead links that occur between these functions are provided in Figure 6.2.

Dead link 1 essentially posits that the prince and the donor need to be in the same location. This link is classified as being *dead* because no other semantic function on the tree provides that information. The second pre-condition, that the donor has the magical object, is shown by the Dead link 2. The third pre-condition, that the prince completes the task set out by the donor, is an active link, since the `complete_task` function is understood to have a post-condition which states that the task has been
completed by the prince. Dead link 3 stems from the `complete_task` function, and essentially is the fact that the prince is now a merchant. This link is dead since no other function on the semantic tree requires it as a pre-condition. For simplicity, we have provided only a minimal number of active and dead links. Each of the functions on the semantic tree may have many more optional pre- or post-conditions associated with it that are not shown here.

### 6.1.2 Simple Interpolation

Interpolation is the satisfaction of one or more dead links by way of an intermediate function. In our example, we wish to interpolate between `hero_reaction` and `provision_of_mobj`. The function we insert between these two functions must match one or more of the dead links stemming from the function `provision_of_mobj`,
or from the function hero\_reaction. Additionally, the pre-conditions and post-
conditions of the interpolated function must not invalidate or otherwise conflict with
the condition that occurs in the active link between the functions hero\_reaction and
provision\_of\_mobj. This latter restriction is justified, since it would not make sense
in the story to suddenly have an undoing of a previous plot point. In this example, we
will assume that we only want to create a single active link between the interpolated
function and its rightmost neighbour. Figure 6.3 illustrates this.

![Diagram](image)

Figure 6.3: The first step of interpolation, inserting an anonymous function x that
may form an active link.

In this figure, the link that we have created has turned the dead link of near(donor, 
prince) into an active one. We call the function x an anonymous function, since the
generation process has not yet found a function which has the effect that will create
the context of near(donor, prince). Essentially, this involves a search through all
of the semantic functions that exist in our library to find one that expresses the condition explicitly, or contains it implicitly in a post-condition\(^1\). The only additional constraint is that the pre-conditions and post-conditions of the function cannot violate any of the other active links around the function. One likely function would be `travel`, which describes the travelling a hero may undertake. The declaration for this function appears in Listing 6.1.

**Listing 6.1** An example listing of the `travel` function.

```plaintext
travel :: (entity, entity) -> action
travel(traveller, destination) =
  context 1:
    pre have(traveller, horse)
    mount(traveller, horse)
    ride_to(traveller, horse, destination)
    post near(traveller, destination)
  end
  context 2:
    ...
```

The `travel` function is understood to accept an entity as a `traveller` and a `destination` as formal parameters, and will expand to produce the details regarding the actual travel. In the case of `context 1`, which is the context we are interested in, the system requires that the `traveller` already has a horse. If so, then the `traveller` mounts the horse and rides it to the specified location. A consequence of this function is that the `traveller` is near the `destination` specified. This could have been a post condition attached to the `ride_to` function, but to make it clear, we have associated it with the `travel` function. Given that this post condition matches the requirements for the dead link `near(donor, prince)`, it is a candidate to be inserted between the two functions. Figure 6.4 provides a graphical example of the new semantic tree with the inserted function.

Once the interpolated function has been inserted into the semantic tree, several

\(^{1}\)This task is made somewhat easier if we tag each function with a list containing all of the other functions it appears in as either a post-condition or a pre-condition (see Section 5.5.2).
things may occur. The function may be elaborated to produce more details as described by Chapter 5. The newly inserted function may also serve for the basis of other interpolations. For example, there may be other dead links that stem from the travel function which can be used for successive interpolations. New functions may be inserted between the travel and provision_of_mobj functions, or between the travel and hero_reaction functions. Figure 6.5 provides an example of the set of links that is now available once the interpolated function has been inserted.

Given the nature of the simple example used to demonstrate interpolation, there are several points to discuss. The interpolated function does not need to have all of its pre- and post-conditions satisfied. In other words, there may be dead links that branch off of the interpolated function. The only condition that is imposed is that the interpolated function must form at least one active link, since failure to do so would mean that the interpolated function is in no way related to the surrounding
Figure 6.5: The new links that become available after interpolation is complete.
context. Following interpolation, the newly inserted function may undergo a series of elaborations. As we have observed in the previous section, the elaboration process does not change the overall nature of the pre- or post-conditions of a function, it simply adds additional details to the tree. Thus, a valid process would be to generate the story structures using a story grammar, perform an elaboration, add new functions to the tree using interpolation, elaborate the newly added expressions, etc. The development of the controls that allow the user to control the generation system with respect to these cycles is left to future work.

6.1.3 Satisfying a Dead Post-Condition Link

The previous example demonstrated how a single function could be inserted that satisfies the pre-condition of a particular function. The process is no different if we chose to satisfy a post-condition instead. For example, now that the travel function has been inserted, we will assume that we want to satisfy the dead link of \text{is merchant(prince)} - the prince is a merchant. Essentially, the interpolation process works the same way. In this instance, we look for a function that requires \text{is merchant(prince)} as one of its pre-conditions. For this example, we will define a function called \text{sell wares} that has this required pre-condition. Listing 6.2 provides the details of this declaration.

\begin{verbatim}
Listing 6.2 An example listing of the sell wares function.

sell wares :: (entity) -> action
sell wares(seller) =
  context 1:
  pre is merchant(seller)
  sell x to for(seller, unspec, unspec, money)
  post have(seller, money)
  end
  context 2:
  ...

In this listing, the seller must be a merchant before context 1 may be selected.
\end{verbatim}
Again, other contexts are left up to the imagination of the reader. The \texttt{sell\_x\_to\_for} is meant to be a function such that the \texttt{seller} sells something to someone, who in return provides the \texttt{seller} with money. The formal arguments for \texttt{sell\_x\_to\_for} require the \texttt{seller}, the object being sold, a person to sell to, and the payment for the transaction. Both the object being sold and the person it is being sold to are left as unspecified items. Presumably, the function \texttt{sell\_x\_to\_for} may itself be further elaborated to give much richer details\footnote{The declaration of this function is left out for the sake of brevity.}. The post-condition on the \texttt{sell\_wares} function basically says that the hero has money on completion of the function. Again, this could have been attached to the \texttt{sell\_wares} function, but is placed here for clarity. Since this function fits the need for our link, we may insert it as an interpolated function between \texttt{complete\_task} and \texttt{travel}. Figure 6.6 provides an example of the state of the semantic tree after this interpolation is performed.

### 6.1.4 Satisfying a Dead Pre-Condition and Post-Condition Link

It is also possible with an interpolation to satisfy both a pre-condition and a post-condition. Building on from the example above, we may wish to satisfy the dead link \texttt{have(prince, money)} stemming from the function \texttt{sell\_wares}, as well as the dead link \texttt{purchase(prince, horse)} stemming from the function \texttt{travel}. In essence, the generation system needs to find a single function that has as a pre-condition \texttt{have(prince, money)}, and produces a post-condition of \texttt{purchase(prince, horse)}. We will assume that a function \texttt{buy} is available, and that the function contains an expansion which contains a context in which a hero has money as a pre-condition, and subsequently uses that money to buy a method of travel. We can also assume that the horse is a parameter of the function, thus we can use the horse parameter from the \texttt{travel} function as the object the prince wishes to buy. An example of the
Figure 6.6: The insertion of a function which satisfies the is_merchant(hero) post-condition of the function complete_task.
function buy is provided in Listing 6.3.

Listing 6.3 An example listing of the buy function.

```plaintext
buy :: (entity, entity) -> action
buy(buyer, object) =
  context 1:
    pre have(buyer, money)
    pre near(buyer, merchant) : optional
    purchase(buyer, object)
    post have(buyer, object)
  end
  context 2:
    ...
```

The post-condition of the buy function is that the hero has the object that was purchased. Again, this post condition may have been inside of the purchase function, but for clarity, we attach it to the main function. Once the function has been inserted into the semantic tree, the dead links from sell_wares and travel now become active. Figure 6.7 illustrates this.

### 6.2 Complex Interpolation

The previous section provided an example of “simple” interpolation, where only a single pre-condition, a single post-condition, or a single pre- and post-condition were satisfied at one time. More complex interpolation occurs when multiple dead links are satisfied at once and when links span long distances. This section discusses these more advanced topics.

#### 6.2.1 Multiple Link Satisfaction

The strategy employed to satisfy links changes when multiple pre- or post-conditions are encountered. In essence, the generation system may do one of three things. First, it may attempt to satisfy only one of the pre- or post-conditions associated with a
Figure 6.7: The insertion of a function to satisfy multiple links: the have(prince, money) post-condition of the function sell_wares, and the pre-condition have(prince, horse) from the function buy.

set of functions, according to the strategy described in the previous section. Second, it may try to find a single function which supplies all of the necessary pre- and post-conditions for all of the links at once. This requires more effort. Third, it may try to fill all of the dead links, one at a time, using one or more functions to do so. This has an overall cumulative effect of satisfying all of the links, but introduces potential problems as to where the new functions are to be placed.
6.2.1.1 Satisfying Multiple Links with a Single Function

To use a single function that satisfies two or more links at once, the generation system must perform a search across all the available functions, and find candidates which contain the semantic expressions that are required. Given the nature of topoi, pre- and post-condition tagging, care must be taken to ensure that the function actually satisfies each of these links within a single context. For example, let us return to the original version of our semantic tree before we began interpolation, focusing on the function `provision_of_mobj`. Figure 6.8 contains a representation of this tree with the appropriate dead links.

![Figure 6.8: A trimmed semantic tree focusing on provision_of_mobj.](image)

This example contains the two dead links that originally appeared at the start of this chapter: that the donor has the magic sword, and that the prince is near the donor. We wish to interpolate and satisfy both of these links at the same time. In order to do so, the generation system needs to search through the topoi lists and post-condition lists contained within the functions `near` and `possess`. In essence these two lists tell the generation system what functions contain these two statements. For the purposes of this example, we will assume that the functions `near` and `possess` are declared in the semantic lexicon according to Listing 6.4.

This listing shows that the function `near` appears within the topoi `kidnap`, `steal_fortune`, `steal`, and `villainy`, and that it is a pre-condition for the functions
Listing 6.4 An example listing of the near and possess functions, with their topoi, pre- and post-condition lists.

```haskell
near :: (entity, entity) -> action
near(person1, person2)
"person1 is near person2"
topoi_list: kidnap, steal_fortune, steal, villainy
pre_list: kidnap
post_list: move_toward, donor_get_object_and_travel

possess :: (entity, entity) -> action
possess(person1, object)
"person1 has object in their possession"
topoi_list: steal_fortune, villainy
pre_list: give
post_list: take, steal, donor_get_object_and_travel
```

kidnap, and also appears as a post-condition to the function move_toward as well as donor_get_object_and_travel. Similarly, the function possess appears in the topoi steal_fortune and villainy, is a pre-condition for the function give, and is a post-condition to the function take, steal and donor_get_object_and_travel. Using the topoi and post-condition lists, the generation system must look for a function that contains both near and possess. At first glance, it appears that the functions villainy and steal_fortune are valid candidates, since they are common to both sets of lists. However, the generation system must perform extra checking to make sure that both of the functions in question appear suitably within the definition of these functions. Listing 6.5 provides the declarations of these functions.

In the listing above, the villainy function may expand into a kidnap expression or a steal_fortune expression. Both the near and possess functions are marked as occurring within the steal_fortune function, so the villainy function may be a valid candidate for the interpolated function under two circumstances: (i) we guide the process so that villainy turns into steal_fortune; and (ii) the steal_fortune function contains a suitable expression of both near and possess. Knowing this, the generation system would then move to examine the steal_fortune function. Within
Listing 6.5 An example listing of the villainy and steal_fortune functions.

villainy :: (entity, entity) -> [action]
villainy(villain, victim) =
  random
  > kidnap(villain, victim)
  > steal_fortune(villain, victim)
end

steal_fortune :: (entity, entity) -> [action]
steal_fortune(villain, victim) =
slist(
  near(villain, dwelling_of(victim)),
  possess(villain, qual(key, unlock(REL, dwelling_of(victim))))),
  unlock(villain, qual(door, belong_to(REL, dwelling_of(victim)))),
  enter(villain, dwelling_of(victim)),
  steal(villain, qual(fortune, belong_to(REL, victim)))
)

this listing, the function basically has an English equivalent of the following story:

The villain is near where the victim dwells. The villain has a key that can unlock the dwelling of the victim. The villain unlocks the door to the victim’s dwelling. The villain enters the victim’s dwelling. The villain steals the fortune that belongs to the victim.

The generation system needs to check the two statements of interest, the near and possess expressions. In order for the steal_fortune function to be considered a candidate for the interpolated function, the statements in the body of the function must express near(prince, donor) and possess(donor, magic_sword). As a first pass, the generation system checks to see if there are any unspec entities already involved within the statements near(villain, dwelling_of(victim)) and possess(villain, qual(key, unlock(REL, dwelling_of(victim))))). Neither statement has an unspec as a formal parameter that could be replaced by either the prince or the donor. The generation system then checks to see if there are formal parameters that could be used to pass in the values prince and donor. The steal_fortune function does indeed have two argument places, one called villain and one called victim which could be used to hold prince and donor respectively.
The generation system then checks to see if the substitution of these two parameters would result in the desired statements. It is at this point that the search fails, since passing in those values would result in the statements `near(prince, dwelling_of(donor))` and `possess(prince, qual(key, unlock(REL, dwelling_of(victim))))`, neither of which resembles the required statements. Thus the functions `villainy` and `steal_fortune` are removed from the list of possible interpolated candidates.

The next candidate is the `steal` function, since `near` appears within its body, and `possess` is listed as one of its post-conditions. In order to determine if the `steal` function is a valid candidate, we must know more about its composition. Listing 6.6 provides the declaration of the `steal` function.

**Listing 6.6** An example listing of the `steal` and `move_toward` functions.

```plaintext
steal :: (entity, entity, entity) -> action
steal(villain, object, victim) =
    slist(
        move_toward(villain, victim),
        take(villain, object, victim)
    )

move_toward :: (entity, entity) -> action
move_toward(person1, person2) =
    post near(person1, person2)
    "person1 moves toward person2"
```

In this listing, we see that the `steal` function involves two different actions. First, the `villain` moves toward the `victim`. Second, the `villain` takes the designated `object` from the `victim`. This particular function is a candidate because a post-condition of the `move_toward` function contains `near(villain, victim)`. Additionally, the post-condition of the `take` function is `possess(villain, object)`, as seen in Listing 5.7. If we can substitute `donor` for `villain`, `prince` for `victim`, and `magic_sword` for `object`, then this function may work as an interpolated function. However, one of the pre-conditions listed on the `take` function is that the `victim`
must actually have the **object** in his possession. Presumably, somewhere at the start of the story, the semantic tree would have stated that the **prince** did not possess the **magic_sword**, a condition which would prevent this particular function from being applicable. Thus, the **steal** function is removed from the list of candidates.

The final function common to both post-condition lists is the **donor_get_object_and_travel** function. To determine if this function is a valid interpolation candidate, we need to examine its declaration more closely. Listing 6.7 contains the declaration of the function.

**Listing 6.7** An example listing of the **donor_get_object_and_travel** function.

```haskell
donor_get_object_and_travel :: (entity, entity, entity) -> action
donor_get_object_and_travel(donor, object, hero) =
  slist(
    travel_to(donor, object),
    take(donor, object, unspec),
    move_toward(donor, hero)
  )
```

In this function declaration, we can see that the **donor_get_object_and_travel** function is a simple **slist**, where the **donor** will travel to the location of the **object**, **take** it from an unspecified entity, and then **move_toward** the **hero**. While the functions **near** and **possess** are not directly stated within the contents of the function, they appear as post-conditions associated with the actions **move_toward** and **take**. The generation system would need to check that the correct statements are generated if it were to substitute **prince** for **hero** and **donor** for **donor**. In this case, the correct statements are generated, and thus the function **donor_get_object_and_travel** becomes the function used for interpolation. Figure 6.9 contains a graphical representation of the tree after this function has been inserted.
6.2.1.2 Satisfying Multiple Links with Multiple Functions

A second strategy to satisfy multiple links involves using many different functions to fill the gaps. This particular strategy is useful if a single function cannot be found that satisfies all of the required links at once. We will return to the example of the function provision_of_mobj that we saw at the beginning of the chapter. This particular example is provided once again in Figure 6.10.

This time, to begin the interpolation process, although we eventually want to satisfy both of the links that stem from provision_of_mobj, we will pick one and satisfy it first. We will start with the link possess(donor, magic_sword). We know from the previous section in Listing 6.6 that the function steal has in its set of post-conditions of one of its functions, that the villain will be in possession of the object specified. Thus, the generation system may insert steal(donor,
\texttt{magic\_sword, \textit{unspec})} as the first of its interpolated functions. This essentially says that the \texttt{donor} moves toward an unspecified victim and takes the \texttt{magic\_sword} from him. The resulting semantic tree is shown in Figure 6.11.

Figure 6.11: A simple semantic tree, after the \texttt{steal} function has been inserted to satisfy one of the dead links.

Once the \texttt{steal} function has been inserted, the links surrounding the function \texttt{provision\_of\_mob} may change, providing a more accepting environment for the insertion of different functions. However, the generation system still needs to find a function to satisfy the dead link \texttt{near(\textit{prince, donor})}. In this example, the function \texttt{move\_toward} provides the necessary set of semantics, and can be directly inserted into the tree. The results of this are shown in Figure 6.12.

\subsection{6.2.1.3 Link Order}

The choice of which link to satisfy first has consequences. Logically any function that satisfies a dead link may be chosen to start, however, the context created by the insertion of the function may invalidate or make available other options. For example, if the \texttt{steal} function had many more dead post-condition links originating from its declaration, then it might have been possible to find a function other than
move_toward that satisfied both a post-condition from steal and a pre-condition for provision_of_mobj. Thus, the order in which the links are satisfied has an impact on the sets of functions that are available when satisfying other dead links. Simple methods, such as pre- and post-condition numbering, to control the order that links are satisfied can be introduced as required.

### 6.2.2 Long-Distance Link Satisfaction

The satisfaction of dead links need not be relegated to the immediate neighbours surrounding a function. A dead link that appears a great distance away can be satisfied by a function inserted somewhere else on the tree. There are, however, limits to this link satisfaction. As a general principle, when the generation system tries to build an active link between two dead links, it cannot skip over other dead links that could be used in place of the long distance link. The reasons why are best demonstrated with the example shown in Figure 6.13, where the generation system is attempting to satisfy the dead link possess(prince, money) associated with the
expression buy.

Figure 6.13: A semantic tree attempting to satisfy the dead link $\text{possess}(\text{prince, money})$ stemming from the $\text{hero\_buys\_travel}$ function.

In this particular scenario, there are two ways that the dead link may be satisfied. It may join up with the dead link stemming from the $\text{steal}$ function, or it may join up with the dead link stemming from the $\text{x}$ function. Faced with these two choices, the system should choose the nearest link. In this example, an active link occurs between $\text{x}$ and $\text{buy}$. The reason for choosing the nearest link is that if we skip over a link that satisfies a requirement, we run the risk of introducing inconsistencies if we perform other interpolations which then make use of that dead link.

To understand the problems that may occur, let us assume that we connect the dead link from $\text{buy}$ to the dead link on the $\text{steal}$ function, thus violating our link rule for a moment. It is possible that future interpolations may insert expressions between $\text{x}$ and $\text{y}$, and that this new context may invalidate the long distance link between $\text{buy}$ and $\text{steal}$. For example, if we insert the function $\text{become\_bankrupt}$, which we will
assume acts in an obvious way, then we have a situation where the semantic tree has a continuity error. The `become_bankrupt` function has a post-condition that says `possess[neg](prince, money)`, yet later on in our semantic tree, we say that the `prince` had sufficient money to buy a horse! This is obviously a problem for the continuity of the story. This tree problem is provided in Figure 6.14.

![Semantic Tree](image)

Figure 6.14: An example of a semantic tree where a continuity conflict has occurred due to the failure of using the nearest neighbour when satisfying a dead link.

While this issue can be solved by using a continuity fix (see Chapter 7), the problem might have been avoided altogether if the generation system has joined the function `buy` and `x` together.

### 6.3 An Example

Now that elaboration and interpolation have been developed, it is useful to provide an example to see the types of stories that can be produced using these two mechanisms.
Recall that in Section 4.2.1.3, we produced a story using one of the schemas that Propp (1968) explicitly defined. The original story appears below:


This simple story can be augmented by employing both elaboration and interpolation. First, assuming functions exist that can elaborate the details for the kidnapping and the fight, we can transform the folktale above and produce enriched descriptions. Second, interpolation may produce details between the Prince leaving home and seeing the witch, resulting in an augmented story structure. The result of applying elaboration and interpolation is given below. Text appearing in italic and labelled with $[\text{E}]$ and $[\text{I}]$ indicate where elaborations and interpolations produce enrichments, respectively.

Once upon a time, a witch sneaked into the castle where Princess Olivia lived. The witch sneaked into Princess Olivia’s room. The witch hid herself under the bed. Princess Olivia came to Princess Olivia’s room. The witch cast a sleeping spell on Princess Olivia. The witch wrapped Princess Olivia up in a bag. The witch cast a transport spell. The witch and Princess Olivia disappeared. The witch and Princess Olivia appeared at the witch’s house. $[\text{E}]$

The father of Princess Olivia asked Prince Montague to rescue Princess Olivia. Prince Montague decided he wanted to save Princess Olivia. Prince Montague left home. $[\text{I}]$

Prince Montague saw the witch. $[\text{E}]$

Prince Montague drew his sword. Prince Montague charged at the witch. The witch jumped back and cast a freeze
 Prince Montague killed the witch. Prince Montague freed Princess Olivia. Prince Montague returned home. The father of Princess Olivia gave Prince Montague lots of money.

Unfortunately, we have surpassed the limits of paper, and cannot show the semantic tree due to the fact that it cannot be printed at a legible resolution. However, this example is meant only to demonstrate how a folktale may be enriched, given suitable function definitions.

6.4 Discussion

Now that we have developed a method for adding additional structures to a semantic tree, we are able to create more complex stories than if we used a story grammar alone. The introduction of this new mechanism raises several issues.

6.4.1 Component Reuse

One of the main benefits to using interpolation in the way that has been defined is that all of the required changes to the functions are already in place. The process of creating a mechanism for elaboration already required that we endow each of the semantic functions with lists of pre- and post-conditions. Fortunately, these lists could be re-used by the interpolation mechanism. In addition to this, the elaboration mechanism outlined a method of how to search for a given topos that supplies the semantic functions that can satisfy various pre- and post-conditions. The interpolation mechanism makes use of that topos search when looking for its own set of functions. The same mechanism of tagging various functions with additional lists of the pre- and post-condition functions that they appear within saves us the same problem of having to determine a functional hierarchy.
6.4.2 Insertion of a Story

The interpolation process is not limited to the insertion of a single expression. Consider the fact that our semantic lexicon may include functions which themselves consist of an slist containing an entire story. As long as the pre-conditions and post-conditions of the story match up with the set of dead links, then there is no reason why the story may not be inserted during the interpolation process. This process is slightly more complex however, since the story to be inserted must be checked first to see what sorts of dead and active links it contains.

6.4.3 No Backtracking

One important feature to note about the interpolation process is that it does not involve any kind of backtracking. That is to say, the interpolation process works by inserting a single valid function onto the tree before it attempts to insert a second valid function. The interpolation process will end if no valid function can be found for insertion. The state of the tree once interpolation ends should be consistent, since the interpolation ensures that the correct conditions are always met before performing an insert.

6.4.4 Impact on Content Creation

The use of the interpolation mechanism, coupled with the elaboration mechanism, now makes it possible to create very complex stories. This type of creation is extremely useful. Consider the generation of a plot for a daily soap opera. In general, given the fact that the show is to air daily with new content, writers for the show consistently have to generate new content that makes use of, and fits in with the old plot threads from shows aired on previous days, weeks, months, years, or even decades. The interpolation mechanism would be valuable in such a circumstance. To
generate a new daily episode for the show, the generation mechanism need only exam-
ine the list of available threads from every single show, and simply interpolate. Once
a function has been interpolated, it may be elaborated, in effect adding the details
necessary for the next daily instalment of the show. This process may be repeated
again and again, presumably forever. Other areas where this type of infinite regress
is useful include comic strips, serial novels, etc. An entire season of a television show
can be written simply by specifying the general outline of a story arc. Interpolation
and elaboration could then produce details sufficient to fill the required number of
episodes with content.

6.4.5 Structure Creativity

As mentioned previously, Pérez y Pérez and Sharples (2004) believe structure pre-
dictability has a negative impact on a generation system’s c-creativity. The addition
of the interpolation mechanism helps to offset the predictability of stories by making
successive additions to the overall structure initially generated by the story grammar.
By adding to the structure, we are able to create stories that the story grammar
alone would be incapable of producing. In addition, depending on how many func-
tions are available, and their sets of optional pre- and post-conditions, there may be
literally an infinite number of ways of modifying the initial story structure with new
and interesting content. Clearly this new mechanism provides an improvement to the
resultant c-creativity of the system.

6.5 Summary

The concept of interpolation was introduced to reduce the predictability of a story
grammar by modifying the structures it creates. A function may be inserted in
between any two successive functions that occur in a story to enhance creativity. A set
of rules based on pre- and post-conditions were developed to identify an appropriate set of functions. The pre-conditions and post-conditions of the inserted functions must agree with the context created by its surroundings. Strategies for inserting more than one function during interpolation were proposed.
Chapter 7

Continuity

Up to this point, we have not explored the possibility that conflicting sets of information may be placed on the tree during elaboration or interpolation. These conflicts essentially destroy the continuity of the story. For the purposes of this research, continuity is defined as a logical sequence of events that occur from one moment to the next in a story. As readers, we take continuity for granted. For example, it would be illogical for a character to leave on a long journey to visit relatives, and then have the character show up in the next scene without some sort of explanation as to his appearance. We rely on the author to explain how these two seemingly different sources of information can exist at the same time. The purpose of this chapter is to explore these issues more fully and describe ways in which the generation system can check for continuity problems, and fix them, should the need arise.

7.1 Continuity Conflicts

A continuity conflict is defined as any set of information placed on the semantic tree that violates a requirement of some other function already present on the tree. This usually occurs when an elaboration is performed, since the resultant set of semantic expressions are not checked against the requirements of other functions waiting to be
expanded. Consider the generation of a folktale involving Prince Rutherford and his brother, Prince Charming. At the start of the generation process, the system reads in a story grammar which contains a list of functions, structured to produce a coherent story. For the following examples, we will only concern ourselves with a portion of the story, represented in Figure 7.1.

Figure 7.1: Part of a plot from a Proppian folktale containing a continuity conflict.

In this example, the function family_tragedy occurs. As we saw in Section 5.3.4.2, the family_tragedy function involves an incident where one individual kills another accidentally. The motif behind the function is that through this accidental killing, the individual responsible for the death is forgiven by his father. In this figure, we have left out the details of the function, and instead choose only to show the dead link that occurs as a result of the function: dead(prince_charming). When the generation system continues to expand the next function, it comes across the brothers_fight_villain function. Listing 7.1 contains the declaration of this function, as it would appear in our semantic lexicon.
Listing 7.1 An example listing of the brothers_fight_villain function.

brothers_fight_villain :: (entity, entity, entity) -> action
brothers_fight_villain(hero1, hero2, villain) =
    pre brother_of(hero1, hero2)
    fight(elist(hero1, hero2), villain)
    post qual(villain, dead)

In this listing, we find that both heroes engage in a fight against the villain. Here is where we reach our continuity conflict: in the previous elaboration, prince_charming dies. Unless the function explicitly checks for this semantic expression as a pre-condition, then the conflict will remain on the semantic tree. This set of conditions, which we will label the dead brother scenario, illustrates a problem with modifications to the tree. Specifically, the continuity of story events can be violated. In order to prevent this from happening, after generation, the system needs to perform a check on the details regarding who knows what, where an object is, and how things come to be, so that the details of the story make sense when told to the reader. As an alternative, or in combination with this process, additional mechanisms may need to be introduced to elaboration or interpolation to prevent these types of conflicts from occurring in the first place.

7.1.1 Preventing Conflicts During Generation

Continuity conflicts arise in several ways. For example, it may be that the set of pre-conditions to the brothers_fight_villain function were underspecified, or it could be that the pre-conditions did not take a variant, yet equivalent statement into account. Once these types of situations are understood, the generation system can be modified to prevent these conflicts from coming into existence in the first place. The following section describes how these conflicts occur, and how they can be avoided during the generative process.
7.1.1.1 Underspecified Pre-conditions

The problem we observed with the dead brother scenario was that the function `brothers_fight_villain` did not take into consideration the possibility that one of the two heroes may be dead. The immediate solution to the problem is to modify the function declaration so that a series of pre-conditions checks to ensure that the entities involved are actually alive. An example of this declaration is provided in Listing 7.2. In this listing, the pre-conditions explicitly state that both the heroes must be alive.

Listing 7.2 An example listing of the `brothers_fight_villain` function, fixed to check whether one of the heroes is dead.

```plaintext
brothers_fight_villain :: (entity, entity, entity) -> action
brothers_fight_villain(hero1, hero2, villain) =
    pre brother_of(hero1, hero2)
    pre qual(hero1, alive)
    pre qual(hero2, alive)
    fight(elist(hero1, hero2), villain)
    post qual(villain, dead)
```

While explicitly stating that the heroes must be alive works to prevent continuity problems, it comes with certain side effects. The first is that we have limited the use of the function to a set of specific contexts. For example, as suggested by Levison et al. (forthcoming), if we were to enforce the restriction that entities must be alive in the function `sleep`, then we effectively prevent the generation system from using that function to express metaphor. In other words, it would become impossible to produce “ideas that sleep furiously,” since ideas may not technically be alive. This issue can be resolved by defining a different version of the `sleep` function, or by explicitly building in a second context to the function that does not require the argument to be alive. In either case, it is up to the programmer to exercise care to ensure that the function is free from continuity errors.

The second side effect is that we have added complexity to our semantic functions.
In the worst case, the same extremely long lists of conditions may need to be added
to a large number of functions, and repeated for each context. Keeping track of all of
the required sets of pre- and post-conditions would then become tedious. Clearly a
method of grouping sets of pre-conditions into a class would be helpful. This solution
has already been proposed by Levison et al. (forthcoming). Formal arguments can
be restricted so that the function will only accept members that belong to the class
as parameters. For example, with the brothers_fight_villain function, we could
define a class called living that requires the entity to be alive. Or, if we wanted a
different type of restriction, we could create a class called human that would require
the entity to be qualified with traits that make it human. This type of class system
applied to the brothers_fight_villain function is provided in Listing 7.3.

**Listing 7.3** An example listing of the brothers_fight_villain function, making
use of semantic classes.

```plaintext
brothers_fight_villain :: (entity/living, entity/living, entity/living) -> action
brothers_fight_villain(hero1, hero2, villain) =
    pre brother_of(hero1, hero2)
    fight(elist(hero1, hero2), villain)
    post qual(villain, dead)
```

In this listing, the type signature of the function has changed to specify the semantic
class that is expected for each of the entities. This eliminates the programmer needing
to list large numbers of pre-conditions in the body of a function declaration, which
is especially useful if the definition of various classes is based on a large number of
semantic conditions.

While adding conditions to a function, either as a set of pre-conditions or as
a class, can help with the detection and prevention of continuity errors, there is a
new problem. Essentially, we lack semantic equivalences. Currently, the generation
system has no knowledge regarding the similarity of two functions. For example, when
specifying the definition of a qualifier such as dead, it is necessary to have some link
to the qualifier alive, since these two functions are semantic opposites. The result is that pre-conditions may require the generation system to look for very specific functions that represent meaning, and that a search of the tree can fail to return a match, since equivalent meaning is expressed in a different form. For example, there is no equivalence to link \( \text{qual(prince_charming, dead)} \) to \( \text{qual(prince_charming, alive[neg])} \). This information has to be explicitly encoded on each of our semantic functions. Taking this into account, a declaration of the function dead is provided in Listing 7.4.

**Listing 7.4** An example listing of the dead function.

```plaintext
dead :: qualifier
dead =
  post alive[neg]
"dead"
```

In this listing, the qualifier dead is declared to have a post condition of alive[neg]. This expression is essentially a statement of equivalence, declaring dead to be equal to alive[neg]. The addition of this post-condition now allows the generation system to perform a comparison of dead to alive[neg]. Thus, in areas where the semantic expressions may have produced alive[neg], the generation system can properly understand the statement as being dead, and vice versa.

### 7.1.1.2 Runaway Elaboration

Continuity conflicts may also be caused by runaway elaboration, in which the elaboration mechanism generates large amounts of semantic information that do not properly take into account the greater context provided by the semantic tree. For example, assume the same semantic environment is created during generation that we saw in Figure 7.1. In this example, prince_charming is dead as a result of the family_tragedy function. Let us also assume that we have implemented a series of pre-conditions
to the function \texttt{brothers\_fight\_villain} as proposed in the previous section. The listing of this function is repeated below in Listing 7.5.

\textbf{Listing 7.5} An example listing of the \texttt{brothers\_fight\_villain} function, fixed to check whether one of the heroes is dead.

\begin{verbatim}
brothers_fight_villain :: (entity, entity, entity) -> action
brothers_fight_villain(hero1, hero2, villain) =
    pre brother_of(hero1, hero2)
    pre qual(hero1, alive)
    pre qual(hero2, alive)
    fight(elist(hero1, hero2), villain)
    post qual(villain, dead)
\end{verbatim}

Given the methodology outlined in Chapter 5, the elaboration mechanism will check the surrounding context to see whether or not the pre-condition of \texttt{qual(prince\_charming, alive)} exists on the semantic tree. Assuming that the qualifiers \texttt{alive} and \texttt{dead} have been defined to include equivalences, then the generation system will find that \texttt{qual(prince\_charming, dead)} is equivalent to \texttt{qual(prince\_charming, alive[not])}. Since the function requires the pre-condition \texttt{qual(prince\_charming, alive)}, and since that context does not exist, then the elaboration mechanism will happily generate new content that describes how this condition can come into effect, possibly by describing Prince Charming’s birth, or childhood in great detail.

It is here that we have a continuity conflict. What we really want the elaboration mechanism to do is check to see if there is a \textit{conflicting} set of information on the tree, rather than just check that the condition occurs. If there is a conflict, we would like the generation system to implement an interpolation where the pre-condition is \texttt{qual(prince\_charming, dead)}, and the post-condition is \texttt{qual(prince\_charming, alive)}. This type of generative fix is discussed in more detail below in Section 7.2.2. The main point of the argument is that we can use interpolation to “bridge the gap” between two conflicting sets of conditions during generation, rather than have
elaboration blindly add details.

7.2 Fixing Continuity Problems After Generation

The previous section discussed some of the continuity conflicts that may occur during the course of generation. Unfortunately, it may be unavoidable that the story produced using elaboration and interpolation mechanisms contains some sort continuity conflict. Thus, a separate mechanism capable of running after generation is complete is needed to check the overall story continuity and fix errors when they occur. This section discusses methods that can be used to calculate continuity in a story, and fix any issues that it finds.

7.2.1 Calculating Continuity

In Section 3.5 we discussed semantic threads and their uses in a story. These same threads can be used to check for continuity errors. If the system creates a thread for every single semantic object that is instantiated in a story, it can follow along each thread and ensure that the set of actions associated with each object is consistent from moment to moment. Imagine a folktale in which the hero has obtained a magical sword from the donor. A semantic thread for the sword would be created from the moment it is instantiated within the tale. During the expansion of each node, and during elaboration and interpolation, the thread would be continued. This is illustrated in Figure 7.2.

To check for continuity, the thread for the sword would be examined. In the context of our folktale example, if the sword suddenly changed ownership between two characters without some function that describes the transaction, then a continuity error has occurred. Such a continuity error is detectable when we examine the links that occur from one function to the next. In this example, we can see along the
Figure 7.2: Part of a plot from a Proppian folktale, with a thread represented for the magic_sword.

magic_sword thread, that the links involved in each of the functions are satisfied, until we come to the second instance of the fight function. Here we find that the king needs to be in possession of the magic_sword as a pre-condition to the fight function. Since there are no other dead links which could be used to satisfy this pre-condition, there is a conflict on the tree, since it is assumed that someone else has the sword and has not transferred it to the king.

7.2.2 Using Interpolation to Fix Continuity Problems

A decision must be made as to how continuity conflicts can be repaired. One class of solution involves removing information from the semantic tree in an effort to trim away the function that causes the conflict. While this is a valid approach, it will not be examined here. In this work, the solution used to solve continuity conflicts involves using interpolation to provide an explanation as to how conflicting sets of conditions came to be on the semantic tree. At first glance, this particular solution does not
appear viable; after all, continuity problems occur precisely because there are two conflicting sets of information that exist on the tree at the same time. However, the primary purpose of interpolation is to examine the set of contexts on the semantic tree and insert new expressions that are consistent and continuous with the remainder of the story. If a function exists that is capable of reconciling conflicting information, then we can use it to explain away the continuity conflict.

Continuing the example in the previous section, by tracing through the magic_sword thread, we find that the king needs to be in possession of the magic_sword in the second instance of the fight function. However, the prince is in possession of the magic_sword in the first fight function. The continuity problem here is that for the second instance of the fight to be valid, somehow the king must be in possession of the sword. This conflict is highlighted by the dead link which states \texttt{possess(king, magic_sword)}. Since the link is dead, there is a problem with continuity, since the king is using the sword without having previously obtained possession of it.

To fix this issue, we need to interpolate a function between the two instances of the fight functions that explains the change in ownership. The post-condition of the interpolated function must be \texttt{possess(king, magic_sword)}. This is so that an active link may form between the fight function and the newly interpolated function. Figure 7.3 illustrates this requirement.

The problem now is in determining the set of pre-conditions that should be applied to the interpolated function. Normally when using interpolation during generation, the search for a valid interpolated function would be restricted by the set of pre- or post-conditions between two nodes on the semantic tree. However, in the instance where we need to apply a continuity fix, part of the problem is that we do not know what specific sets of conditions should be used to restrict the search. Any existing set of contextual links will do. Thus, we make a design decision when using interpolation to fix continuity: the pre-condition of the interpolated function will essentially be a
Figure 7.3: Part of a plot from a Proppian folktale, with a thread represented for the \textit{magic_sword}, and a function inserted to fix the continuity problem.
restatement identical to that of the required effect. However, the condition will differ based upon the outcome effect we wish to achieve.

Relating this to our example, we know that the post-condition of the interpolated function should be $\text{possess(\text{king}, \text{magic\_sword})}$. The issue is that the $\text{king}$ does not currently possess the $\text{magic\_sword}$. Thus, we need to know who currently has the $\text{magic\_sword}$, and make that a pre-condition to the interpolated function. In turn, that condition will be used to constrain the search for the interpolated function. The problem now is to determine who was in possession of the sword prior to the interpolated function. To get this information, we move back along the $\text{magic\_sword}$ thread and consult the last active link that expressed the $\text{possess}$ function. This is a simple matter of a query along a particular thread. In this case, the result of this query is that the $\text{prince}$ is in possession of the $\text{magic\_sword}$. Figure 7.4 contains a graphical example of where the query would look.

Figure 7.4: Part of a plot from a Proppian folktale, with a thread represented for the $\text{magic\_sword}$, and a function inserted to fix the continuity problem.
Once both the pre- and post-conditions of the context are known, it is just a matter of using interpolation to find a function that is capable of matching the required context. For example, \texttt{take(king, magic\_sword, prince)} would be a valid function, or \texttt{give(prince, magic\_sword, king)}, or perhaps even \texttt{lost\_treasure\_found(prince, magic\_sword, king)} in which the prince loses his magic sword, and it is found by the king. We assume that there is a wealth of semantic functions in the lexicon that are capable of being used in a wide variety of contexts.

### 7.2.3 Justification of a Thread-Based Approach

The use of threads to check continuity simplifies the task of determining what links need satisfaction, and where in the story the semantic information needs to be inserted. In the previous example, we had a very simple story, where the continuity issue was apparent in very close proximity to where the sword was last used. In reality, the story may have been quite complex, with any number of semantic functions appearing in between. Figure 7.5 provides an example of the same continuity conflict, with many more nodes appearing in between functions.

In this example, the same continuity issue occurs, except functions \(a\), \(b\) and \(c\) occur in between. Since these functions do not make use of the \texttt{magic\_sword}, they are not considered as functions that may cause a continuity conflict with the \texttt{magic\_sword}. This simplifies the situation, since the generation system does not have to consider them when searching for the previous change of ownership of the sword.

### 7.3 Discussion

There are several consequences involved in detecting and fixing continuity threads. Some are relatively minor, but a few are more important. The following section discusses some of these issues in more detail.
Figure 7.5: Part of a plot from a Proppian folktale, with a thread represented for the magic_sword.
7.3.1 Irreconcilable Continuity Errors

Given the fact that there are only a finite number of functions in any semantic lexicon, it is possible that there is no set of interpolations that will fix the continuity error. At this point, the generation system has no choice but to flag an error and report to the user what has happened. Presumably the generation process will provide enough information to the programmer so that he or she may introduce new functions into the semantic lexicon that can be employed to ultimately solve the continuity problem. With respect to the dead-brother scenario, the programmer may be able to introduce a function that bridges the gap between the offending links, by introducing magical resurrection, spiritual communication or some other fix suited to this problem.

While this particular solution may appear to be a shortcoming to the generative system, it is one that can be ultimately overcome. Future work on the continuity system may add the ability to transform various forms of topoi, in order to better fit them to areas where continuity is a problem and no pre-existing function is applicable. We can imagine a system where the transform, recall and adapt methods (TRAMs) seen in MINSTREL (Turner, 1992) are applied to this system in particular. Such adaptive processes would be useful to generate the type of continuity fixes that may be required when checking continuity. This would prevent a user from having to come up with an imaginative function fix manually.

7.3.2 An Iterative Process

As discussed above, using interpolation to fix continuity errors introduces new semantic functions to the tree. Rather than simply end the generation process once continuity has been fixed, it is now possible to apply both elaboration and interpolation to the newly added node. This same type of iterative process was discussed with the introduction of interpolation. The generation system may choose to fix continuity problems, and then use the newly inserted nodes as the basis of further elaboration.
This may, in turn, create new continuity errors, especially if elaboration is employed to add new detail to each of the newly inserted functions. However, the continuity checker would re-thread the new semantic tree and check for continuity conflicts. Controls can be introduced directly to the generation system to limit the number of times this process is repeated. For example, the system can be told to only perform one round of elaboration after continuity. If further errors are introduced, then when the system employs the next round of continuity fixes, these newly inserted nodes are not elaborated or interpolated further.

7.4 Summary

The concept of a continuity error was introduced and discussed. The origin of continuity errors during the course of generation was examined. A method of checking for continuity errors by means of semantic thread traversal was provided. A fix for continuity errors was developed using interpolation.
Chapter 8

Conclusions and Recommendations for Future Work

8.1 Conclusions

In order to enrich the set of statements appearing in a story, several key components are needed. At the lowest level, a semantic formalism capable of expressing detailed natural language meaning is required. A suitable formalism, such as the one defined by Levison et al. (forthcoming) is appropriate, but requires additional constructs in the form of queries and threads, so that it can produce coherent stories as dictated by a story grammar. In turn, the definition of a story grammar requires a literary analysis of the works in a genre such as folktales. Research by Propp (1968) provides this analysis, which forms the basis of the semantic functions in the generation system. The combination of all of these components allows for a focused examination of the thesis objectives of developing methods to (i) provide enriched details for a story event, (ii) enhance an otherwise rigidly defined story structure, and (iii) check for and repair the inconsistencies in the continuity of the story. The realization of these objectives results in the three contributions of this work.
First, elaboration was developed. This process enriches the set of semantic details that relate to a story event. Enrichment is accomplished by enhancing each function to include a set of pre-conditions and post-conditions which ensure that the enhanced description is suitably matched to its surrounding context. If the story context does not allow for any expansion to occur, the elaboration mechanism searches a library of functions, selects one that provides the necessary context, and adds it to the story. Optional pre-conditions and context selectors are examples of controls that can be introduced to limit the extent of elaboration.

Second, interpolation was developed. This process works by inserting a set of functions between two successive story events, which results in stories that have greater variation. Rules that operate according to pre-conditions and post-conditions are used to identify a set of candidate functions that, when inserted, remain consistent within the context of the story.

Third, continuity error checking and correction was developed. The concept of a semantic thread was defined and used to detect continuity errors. Threads work by providing a connection between story elements, making it possible to verify the consistency of state information between successive events. If continuity errors arise, interpolation can be used to as a way of reconciling inconsistent states.

The work is applicable to folktales that describe imaginary worlds, where unreal or fantastic events and objects such as magical swords and talking animals, are common. While the strategies developed in this work may have applications in the greater realm of fiction, additional scrutiny would be required to verify that the result of elaboration and interpolation is coherent and logically acceptable.
8.2 Future Work

While the elaboration and interpolation mechanisms discussed in this thesis provide a new way of expanding upon the contents of various semantic functions, there is still more work yet to be undertaken.

8.2.1 Enhancing Tree Additions

While tree additions provide a mechanism of introducing required semantic content to the tree, a decision was made to restrict the addition so that the required content appears directly before the elaborated function. There are other options available when adding content to the tree. For example, it may be more natural to group the content with functions of a similar type that appear elsewhere on the tree, or introduce the content as an interpolation between earlier functions. A more detailed study is required to determine how these additions could function, and under what circumstances they might be applied.

8.2.2 Pragmatics, Control and Reader Chronologies

Currently, elaboration and interpolation may be performed at any point in the story, and may be applied in a recursive fashion (see Section 6.4.4). However, within this work, it is not discussed why certain areas of the story may be better candidates for elaboration or interpolation than others. As mentioned in Section 1.2, an examination of rhetoric and pragmatics could reveal the nature of the controls responsible for directing when and where elaborations and interpolations occur. Additionally, it is not yet clear whether the processes of elaboration, interpolation and continuity checking are sufficient to generate the types of stories that are written by a human. It may be that other processes are required to supplement the techniques described here. However, in order to reach this conclusion, a number of other studies must be
undertaken.

First, a study to see if elaboration and interpolation provide any added value to a story should be completed. The study should also determine if elaboration and interpolation are necessary and sufficient to generate the content that appears in various genres. Such a study may start by examining a particular author within a genre. A human, upon reading a candidate text, would be responsible for encoding the set of semantic functions that capture the meaning of the text, as well as specifying the structures necessary to control where and when elaboration and interpolation would occur. Once a general outline of the plot was understood, then the human could capture it using the framework and would use the generation system to produce a text. Several versions of the story could be produced using different combinations of elaboration and interpolation, and each one could be evaluated by a human reader. For example, the first set of stories produced by the system would be simple in nature, with the elaboration and interpolation mechanisms turned off. A second set of stories would be produced with only elaboration turned on, a fourth only with interpolation, and a fifth with both turned on. The resultant outputs could then be judged by human readers, who would score the resultant texts in areas such as complexity, coherence, and interest. Conceivably, the results of this study would indicate whether or not elaboration and interpolation provide the necessary tools to produce enriched stories.

A second study would need to be undertaken to examine how the generation of story content through elaboration and interpolation is affected by various rhetorical goals. Again, a human would need to examine a set of works by a particular author in a specific genre to determine what goals are in effect, and what impact they have on the overall style. Once the relationship between the underlying semantic content and the realized story is understood, it should be possible to express how rhetorical goals control the elaboration and interpolation processes. A study could then be conducted to turn the rhetorical goal system on and off, and have the results evaluated by a set
of human readers.

A third study would be used to determine a method of generating reader chronologies. The actual content that appears in the realized text may reflect only a fraction of the semantic information produced by the generation system. Depending on the requirements of the reader, author, or genre, some information may appear in the final text in a highly summarized form, while other information may not appear at all. How this type of information appears in the final text is up to a separate mechanism that is responsible for weaving a reader chronology throughout the story. Much work would be needed to determine how such a threading mechanism may change depending on genre, author, and reader, as well as determine if or how it would be controlled by various rhetorical goals.

Finally, it may be possible to produce an interactive control system, where the author is consulted during the generation process to confirm various choices where elaboration and interpolation are applied. The exact nature of this system has yet to be determined.

### 8.2.3 Topoi Hierarchies

An interesting linguistic development occurs when functions are tagged with topoi, as discussed in Section 5.3.6. Recall that in order to determine what semantic effects any given topos contained, we labelled each function with the set of topoi that they appear in. This in effect, creates a rudimentary hierarchy that provides us clues as to how various functions are related to one another. Consider the example of a simple topos called *villainy*. This topos contains three possible expansions, as depicted in Figure 8.1.

Now, assume that we have a set of semantic expressions which summarize the *villainy* topos as “a villain commits an evil act”. We now have a powerful relationship that exists between *villainy, kidnap, pillage* and *steal*. This relationship
Figure 8.1: The topos villainy, which can be expanded into kidnap, pillage or steal.

says that all three of these acts can interpreted as a type of evil action. In essence, we have created an “is-a” relationship between the functions. More importantly, this relationship was created “for free” by the simple act of tagging functions with topoi. Given this set of information, we can ask more general questions about the story. For example, we can ask “what is the evil act that took place in this story?” More work would need to be undertaken to see exactly how these collections of semantic tags could be mined for interesting information, or used in learning environments to test users about the semantic content of a story.

8.2.4 Story Difference

How can we tell if the story being generated is novel? To answer this question, we need a definition of story difference that can be easily calculated. On the face of it, checking for tree differences appears to be quite easy. Simply checking for different expressions, and counting the number of differences would appear to provide a measure of story difference. Unfortunately, on further investigation, the situation is more complex. Choices near the root of the tree appear to have a larger impact than choices near the terminal branches of the tree. Aesthetically speaking, changing semantic identities without altering the overall structure has a minimal impact on difference. However, a more detailed exploration is warranted before this conclusion can be confirmed. One certainty, however, is that the study of semantic differences appears to be much more complex than it initially appears. Clearly more work on the nature of the semantic
tree will need to be undertaken in order to develop a mechanism that is capable of measuring the wide varieties of differences that may occur between stories.

### 8.2.5 Elaboration and Interpolation in Other Systems

It may be possible to use the elaboration and interpretation mechanisms presented in this dissertation with other existing systems to augment the amount of descriptive narrative available. For example, consider the the examples of output we saw in Section 2.5.2.1. Clearly TALE-SPIN could benefit from the enrichment processes described in this thesis. To do so, we would need to transform Meehan’s original semantic elements encoded in Conceptual Dependency into a set of semantic functions compatible with our framework. Once built and arranged in a semantic tree structure, we could employ the same elaboration and interpolation mechanisms seen throughout this thesis to add more detail to each node. This action would be a distinct and separate process from the original character simulation, and thus would not create any more undue complexity within the original TALE-SPIN system. Our newly defined mechanisms sit on top of the output, provided that a suitable transformative device exists. This is true of a number of other systems described in Chapter 2.
References


Veselóvskij, A.N. (1913). Poètika. 2(1).

