Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

DISTRIBUTED SEMANTIC REPRESENTATIONS FOR GOAL/PLAN ANALYSIS OF NARRATIVES IN A CONNECTIONIST ARCHITECTURE

Geunbae Lee

March 1991
CSD-910009

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Semantic Representations for Goal/Plan Analysis of Narratives in a Connectionist Architecture

A dissertation

submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Geunbae Lee

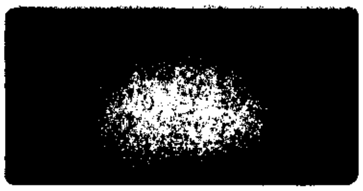
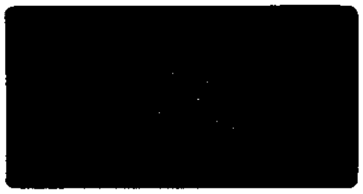
1991

Distributed Semantic Representations for Goal/Plan Analysis of Narratives in a Connectionist Architecture

Geunbae Lee

March 1991

Technical Report UCLA-AI-91-03

To my wife,
Yunsuk Lee
and my daughter,
Hae-In Lee

The dissertation of Geunbae Lee is approved.

Morton Friedman

Morton Friedman

[illegible]

Keith Holyoak

[illegible]

Andrew Kahng

[illegible]

Walter Karplus

[illegible]

Michael G. Dyer, *Committee Chair*

University of California, Los Angeles

1991

Table of Contents

•

List of Figures

List of Tables

Acknowledgments

Special thanks should go to my advisor, professor Michael Dyer, for continuous support, many enthusiastic discussions and excellent suggestions on earlier papers and drafts of this dissertation. He has also tirelessly corrected my writing style, English grammar and organization. I would also like to acknowledge committee members professor Walter Karplus and Andrew Kahng for constructive criticism on the plausibility of the research, claims and methodology, professor Keith Holyoak and Morton Friedman for providing psychological pointers and criticism for the research. The UCLA AI Lab formed a stimulating environment for research. Thanks also go to Margot Flowers who supplied advice on earlier stages of this project. Special thanks go to John Reeves, Trent Lange, and Risto Miikkulainen (now at UT Austin) for reading earlier drafts and for fighting my bad English. Other AI lab members especially Alan Wang, Ron Sumida, Alex Quilici (now at Univ. Hawaii), Charles Dolan (now at Hughes Research Lab.) were very kind and helpful during my stay. Professor John Merriam in the biology department also supported the research, and Joy Jonsen corrected my English in an earlier version of the draft. I would like to thank my wife, Yunsuk Lee, for her continuous support and encouragement during 5 years of my study, and also all of my family members in Korea for their concerns and support.

This research was supported in part by the JTF Project of the DoD (monitored by JPL), by a grant from the ITA Foundation, and by a Korean Government Scholarship. The simulations were carried out on Hewlett-Packard equipment donated to the UCLA AI Lab.

VITA

Mar. 20, 1961	Born, Chongna, Korea
1984	B.A., Computer engineering Seoul National University, Seoul, Korea
1984–1986	System programmer and manager University Computing Center Seoul National University, Seoul, Korea
1986	M.S., Distributed systems and computer network Seoul National University, Seoul, Korea
1987–1989	Research assistant AI Lab. Computer Science Department University of California, Los Angeles
1989–1991	Research assistant Biology Department University of California, Los Angeles

PUBLICATIONS

Geunbae Lee, Margot Flowers, and Michael G. Dyer. Learning distributed representations of conceptual knowledge and their application to script-based story processing. *Connection Science - Journal of Neural Computing, Artificial Intelligence and Cognitive Research*, 2(4):313–345, 1990

Geunbae Lee and Michael G. Dyer. The authors respond: "Portability and bindings with DSRs". *Neural-Network Review*, 4(2):73–74, 1990

John Merriam, Susan Adams, Geunbae Lee and David Krieger. Cloned genes of Drosophila Melanogaster and Literature guide. In S. J. O'Brien editor, *Genetic Maps*, Vol. 5. Cold Spring Harbor Press, N. Y., 1990

Geunbae Lee and Risto Miikkulainen. Distributed connectionist knowledge representations in script/goal-based story understanding. *Proceedings of the Seoul International Conference on Natural Language Processing*, Seoul, Korea, 1990

Geunbae Lee, Margot Flowers and Michael G. Dyer. A Symbolic/Connectionist Script Applier Mechanism. *Proceedings of the 11th annual conference of the cognitive science society*, Ann Arbor, MI., 1989

Geunbae Lee, Margot Flowers and Michael G. Dyer. Learning Distributed Representations of Conceptual Knowledge. Technical Report, UCLA-AI-89-13, Artificial Intelligence Lab., CS Dept., UCLA, 1989. (Abstract in the *Proceedings of the 3rd international joint conference on the neural networks*, Washington D.C., 1989)

Geunbae Lee and ChongSang Kim. Design of Extended Procedural Language in Heterogeneous Network Environment. *Proceedings of Korean Information Science Society (KISS) Conference*, Seoul, Korea, 1985

Abstract of the Dissertation

Distributed Semantic Representations for Goal/Plan Analysis of Narratives in a Connectionist Architecture

by

Geunbae Lee
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Michael G. Dyer, *Chair*

Recently there has been much research on natural language understanding using connectionist computations, but the knowledge representation schemes used in these models are at best ad-hoc and cannot support high-level cognitive tasks very well. This thesis presents a new method for developing distributed connectionist representations called *distributed semantic representations* (DSRs). The DSR scheme has been developed in order to serve as an adequate foundation for constructing and manipulating conceptual knowledge in connectionist natural language understanding systems.

DYNASTY is a distributed connectionist script/goal/plan analysis system that can read short narratives and produce goal/plan-based inference chains. DYNASTY can: (1) learn similarity-based word concept representations using the DSR technique; (2) automatically form high-level distributed representations for events, goals, plans, and scripts; (3) select relevant goals/plans and scripts with correct bindings from fragmentary input events; and (4) generate plausible inferences from the selected goals/plans and scripts. We believe that DSRs can serve as building blocks for constructing high-level connectionist natural language understanding systems since a DSR has both symbolic and distributed representational features. DYNASTY provides many desirable features of connectionist implementation, which are not possible in pure symbolic systems, such as automatic and statistically-biased generalizations, automatic knowledge encodings through training, fault tolerance and graceful performance degradation.

Part I

OVERVIEW

Chapter 1

Introduction

1.1 Task: Narrative comprehension

DYNASTY is an Artificial Intelligence (AI) model of a large-scale distributed connectionist system which can read script and goal/plan-based stories and produce appropriate inferences. DYNASTY's tasks at top level can be divided into 3 major components: (1) formation of word representations, (2) script processing, and (3) goal/plan analysis for narrative comprehension.

1.1.1 Formation of word representations

DYNASTY's first task is to form distributed word representations to support story comprehension and inferencing. DYNASTY's word-learning module (DSR-learner; chapter 2) reads a set of propositions and extracts word representations according to how each word is used in those propositions. As a result, a word representation in DYNASTY reflects all the usages of the word in a limited domain.

For example, from the following two propositions about **coffee**, we can see how the word coffee is semantically related to other words in the propositions and therefore can form a representation (concept) of coffee.

- p1: The man drinks coffee.
- p2: Coffee is hot.

According to these example usages of "coffee", the representations of coffee should reflect the idea of some physical object that humans can drink and that is hot. Of course, from these two propositions, the representation of coffee cannot be unique since the word "tea" can have exactly the same relations here. When we add the third proposition "p3: America imports coffee from Columbia", the representation of "coffee" can become different from that of "tea". DYNASTY reads several propositions and develops representations for each word (e.g. coffee as above) in a fixed-width vector representation. The propositions used in DYNASTY actually come from collected story comprehension training data, so DYNASTY develops representations for all the words used in the story.

1.1.2 Script processing

Script-based stories describe stereotypic event sequences in a given environment. These kinds of event sequences contain mundane knowledge about routine procedures which is shared by the members of a given linguistic community. For example, we all know what events occur in a restaurant, so when we read a short story (story 1) about a restaurant, such as:

(story 1) `John entered the Chart-House. John ate the steak. John left a tip.`

we can immediately produce numerous inferences which are not explicitly mentioned in the story [Schank and Abelson, 1977]. After reading the above story, we can easily fill in the missing events and generate a completed paraphrase, with all the script-based events fully expanded, such as:

```
John entered the Chart-House. The waiter seated John.
The waiter brought the menu. John read the menu. John ordered steak.
John ate the steak. John paid the bill. John left a tip. John left
the Chart-House for home.
```

As we can see in the above paraphrase, we can infer that John ordered the steak even though the "ordering" event was not explicitly mentioned in the input story (story 1) since everyone orders before eating in the restaurant.

DYNASTY reads script-based stories and can produce a complete paraphrase (such as story 1 and its paraphrase) once it is trained on the script data. Just as human beings obtain their knowledge about restaurants from their experiences of going to restaurants, DYNASTY builds its restaurant script knowledge by extracting statistical regularities from the training data.

1.1.3 Goal/plan analysis

DYNASTY's other task is a goal/plan analysis of stories that are not stereotypical. Not all the stories are stereotypic and routine. For example, consider the following story (story 2):

(story 2) `John was hungry. John asked his friend about MacDonald's locations. John drove to the MacDonald's. John had no money. John wanted to call up his friend. John borrowed a coin from the waiter.`

A reader of the story will infer that John asked about MacDonald's locations because he wanted to eat there and one must be near food in order to eat it. Also the reader can infer that John wanted to call up his friend in order to borrow some money and John borrowed a coin because he needed it to phone his friend. If a system is only script-based, then it would need a script that essentially corresponds to the story. In other words, the system would need a script for every conceivable situation it might hear about when people are hungry (something like hungry-no-money-script). This is almost impossible since there are so many possible actions that people can take when they are hungry. They can go to a restaurant, cook for themselves, order pizza by phone, buy some food, etc. People can understand the above story not only because they have all kinds of scripts, but also because they have more general goal/plan knowledge [Schank and Abelson, 1977; Wilensky, 1978].

DYNASTY reads non-stereotypical stories and produces goal/plan-based inferences to comprehend them. Actually, DYNASTY understands script-based stories as a special type of goal/plan-based story that uses only a single plan, so the same architecture can handle both script and goal/plan-based stories.

Goal/plan-based stories describe how the planner fulfills several preconditions to execute a plan for a current goal. The unsatisfied preconditions become the next current goals, and the planner pursues several chains of plans to satisfy these new current goals. There are two cases of goal/plan-based reasoning: (1) horizontal reasoning and (2) vertical reasoning. In horizontal reasoning, the planner tries to fulfill several preconditions to execute a single plan. In vertical reasoning, the planner tries to pursue a single precondition until it is satisfied. Goal/plan-based stories are usually combinations of these two extreme cases. Below we describe how DYNASTY can handle these two extreme cases, and therefore can handle any combination of horizontal and vertical reasoning.

(1) Horizontal reasoning: DYNASTY can understand how people accomplish various preconditions in order to execute a plan. Each action in the input story can match one of the several preconditions for a single plan. This type of reasoning is called horizontal reasoning because DYNASTY constructs goal/plan inference chains by traversing goal/plan trees in horizontal directions (traversing the sibling nodes rather than the child nodes). Suppose DYNASTY reads the following story (story 3):

> (story 3) `John was hungry.  John read the restaurant guide.  John borrowed money from his friend.  John got into a car`

DYNASTY can understand that all three actions in the story (i.e reading, borrowing, and getting into a car) are designed to meet the preconditions of eating at the restaurant. DYNASTY produces a goal/plan inference chain for each action to connect it to the previous context in the story. What follows are inference chains that DYNASTY produces for this story. The □ designates an input sentence, and an inference chain appears immediately after each input sentence. Each inference chain is the output of a single sentence or sentence fragment. The G stands for the top-level goal. The P and PC respectively stand for an executed plan and a satisfied precondition (instrumental goal) to execute the plan. All DYNASTY's output are past forms because what DYNASTY actually produced are states or actions that are related to the goals and plans (see chapter 5 for the output traces). For example, for the top-level goal G and precondition (instrumental goal) PC, DYNASTY produces a state of the planner *when the goal was already accomplished.* For the plan P, DYNASTY produces an action (or action sequence) *which was already executed by the planner* for the plan. DYNASTY does not deal with any plan or goal failure which usually needs more high-level abstract knowledge than goals/plans themselves [Dyer, 1983; Dolan, 1989]. DYNASTY expects that the input story will explicitly mention the necessary preconditions to execute the plan for the top-level goal. When a certain precondition is not mentioned in the input story, DYNASTY assumes that the precondition is already satisfied.

```
☐ John was hungry
G: John was not hungry

☐ John read the restaurant guide
PC: John was not hungry
P: John ate food at restaurant
PC: John knew restaurant location
P: John read restaurant guide

☐ John borrowed money from his friend
P: John ate food at restaurant
PC: John had money
P: John borrowed money from friend

☐ John got into a car
P: John ate food at restaurant
PC: John was inside restaurant
P: John drove to restaurant
PC: John was inside car
```

DYNASTY understands the first sentence as describing the planner's top-level goal. DYNASTY explains the second, third and fourth sentences in terms of John's goals and plans, and constructs the causal connections (goal/plan chains) using the inferred goals and plans. For example, DYNASTY's output inference chain for the second sentence can be interpreted in plain English as follows:

> John selected the eating-at-restaurant plan to achieve not-hungry state, and knowing the restaurant location is a precondition to execute the plan, and reading the restaurant-guide is a plan for knowing the restaurant location. So John read the restaurant-guide, and knew the restaurant location, and executed the eating-at-restaurant plan. John was not hungry anymore.

This interpretation is an explanation for why John read a restaurant-guide in the story. The other inference chains are also explanations for why John did the second and third actions in the input story.

(2) Vertical reasoning: DYNASTY can understand how a planner might fulfill the chain of preconditions to execute a series of plans. A planner pursues a single precondition until it is finally satisfied. Each action in the input sentence can match the intermediate plans to satisfy the current unsatisfied preconditions. This type of reasoning is called vertical reasoning because DYNASTY constructs the goal/plan inference chains by traversing the goal/plan trees in vertical directions (traversing the child nodes rather than the sibling nodes). Consider the following story (story 4):

(story 4) John was hungry. John wanted to get a Michelin-guide. John wanted to know the book-store location. John wanted to call up his friend. John walked to the pay-phone.

DYNASTY understands that the series of events in this story are related to the preconditions for getting a restaurant guide, and it performs the vertical reasoning to generate a single inference chain for the whole story. DYNASTY's outputs for this story are as follows:

```
□ John was hungry
G:John was not hungry

□ John wanted to get a Michelin-guide
PC: John was not hungry
P:John ate food at restaurant
PC:John knew restaurant location
P:John read Michelin-guide
PC:John had Michelin-guide

□ John wanted to know the book-store location
P: John had Michelin-guide
P:John bought Michelin-guide at book-store
PC:John knew book-store location

□ John wanted to call up his friend
PC:John knew book-store location
P:John asked friend about book-store location
PC:John had communication-link to friend
P:John called up friend

□ John walked to the pay phone
P:John called up friend
PC:John was near pay-phone
P:John walked to pay-phone
```

Each inference chain starts with the point left behind from the previous sentence, and all the inference chains are step-by-step explanations for John's plans of getting the Michelin-guide.

Goal/plan-based stories usually consist of random combinations of these two types of reasoning (horizontal and vertical). Story 2 is an example of this combination: up to the event "John had no money", the story tells about how a planner (John) fulfills the several preconditions of eating-at-the-restaurant plan (horizontal reasoning). After the "no money" event, the story switches to how series of preconditions for the getting-money goal are fulfilled (vertical reasoning). DYNASTY can process any combination of these two types of goal/plan-based reasoning (see section 5.2.1 for the I/O traces of story 2).

1.2 Background and motivation

Story understanding, which is a core task of knowledge-based natural language understanding (NLU) research, has long adhered to one paradigm in cognitive science: symbolic computation. The symbolic approach has its origins in the celebrated physical symbol system hypothesis (PSSH) [Newell, 1980], in which cognition is modeled by syntactic operations over abstract symbols and symbolic structures. Within the symbolic approach, there have been many theories concerning the underlying knowledge representations necessary to generate the inferences for understanding stories. Conceptual dependency (CD) theory [Schank, 1973], script/goal/plan/theme theory [Schank and Abelson, 1977], MOPs/TOPs/TAUs [Schank, 1982; Dyer, 1983] and frames/schemata [Minsky, 1981; Rumelhart, 1975] are all knowledge structures that have been used to generate underlying inferences for story comprehension within the symbolic paradigm. Employing these knowledge representation theories, there have been many implementations of symbolic story understanding systems. MARGIE [Schank, 1975] was the first system which paraphrases stories using CD representations. SAM [Cullingford, 1978], Ms. Malaprop [Charniak, 1978], and FRUMP [DeJong, 1979] were implemented based on script/frame theory, and PAM [Wilensky, 1978] was implemented based on goal/plan theory. BORIS [Dyer, 1983] applied structures involving themes and affects. Moreover, memory structures which can be changed by reading stories have been developed and used in the story understanding programs [Lebowitz, 1980; Kolodner, 1980].

Then why bother to re-implement another script/goal/plan-based story understanding system like DYNASTY? The answer comes from a recent apparent paradigm shift [Kuhn, 1970] in the cognitive science field. Recently, an approach which seems to be incompatible with the PHHS has been introduced for cognitive modeling. This new paradigm results partially from a resurrection of the old perceptron-style neural network research on brain modeling which was popular around 1960 [Rosenblatt, 1962; Minsky and Papert, 1988]. Even though there are many names for this new approach, we will use the term *connectionism*.[1] In connectionism, cognitive processing is modeled by networks of connected simple units. Signals are propagated along connections of different strengths and the signals arriving at a unit are summed and thresholded to fire the unit. This theory is still somewhat detached from complete neural realism, but it is obviously more neurally inspired than the symbolic paradigm.

Why do we want to implement a connectionist story understanding system? There are four major motivations underlying the development of DYNASTY as a distributed connectionist story understanding system. These are described in the following sections.

[1]This new approach of brain-style computation has several names widely used in the research community: "connectionism" [Feldman and Ballard, 1982], "parallel distributed processing" (PDP) [Rumelhart et al., 1986c], "artificial neural systems" [Grossberg, 1988], "subsymbolic" paradigm [Smolensky, 1988] and "neurocomputing" [Anderson and Rosenfeld, 1988]. We choose *connectionism* because it is most generally used in the natural language research community.

1.2.1 Developing distributed representations for symbol processing

We would like to develop a distributed knowledge representation scheme that maintains both symbolic and distributed features. Connectionism needs knowledge representation schemes which can support high-level symbolic tasks such as natural language processing. Previous connectionist representations (see section 6.2) lack the features that are necessary to perform these tasks since they are usually developed internally [Hinton, 1986; Pollack, 1988] or by hand (manual encoding) [McClelland and Kawamoto, 1986; Touretzky and Hinton, 1988]. For connectionism to be an alternate method for designing natural language understanding systems, we need a distributed representation that can support both symbolic and connectionist operations. On the one hand, this representation should be globally accessible and should encode the structure of symbolic representations; on the other hand, it should be automatically learned and similarity-based such as distributed representations (see section 2.2 for detailed criteria).

1.2.2 Evaluating connectionist technology

It is important to evaluate the power of distributed connectionism to perform high-level symbolic cognitive tasks. Several claims have been recently made which make it worthwhile to attempt high-level cognitive tasks, such as story understanding, using connectionist networks. The claims are either positive or negative: (1) connectionism is the only computational mechanism needed for cognitive models [Rumelhart et al., 1986c; Hinton and Anderson, 1981], or (2) there are symbolic phenomena in cognition which cannot be modeled by connectionism alone, such as the infinitive generative capacity [Pinker and Prince, 1988] and the systematic compositional nature of cognition [Fodor and Pylyshyn, 1988]. To support or refute these claims, we need several working AI models in the connectionist paradigm which can perform high-level symbolic tasks, and story understanding is a perfect symbolic cognitive task for this purpose. Story understanding requires many symbol structure manipulations, and these tasks are composed of many sub-tasks such as parsing, memory retrieval, memory updating, generation, etc. DYNASTY is an AI model which brings current up-to-date connectionist theories together. Also in DYNASTY, new representation theories have been developed to implement a story understanding system. Our motivation is to provide a working model so that the research community can evaluate the powers and limitations of connectionism for modeling high-level symbolic cognitive tasks.

1.2.3 Overcoming limitations of symbolic systems

Previous symbolic script/goal/plan-based story understanding systems have many deficiencies, due to the nature of symbolic computation. We hope to eliminate these deficiencies by encoding symbolic knowledge in distributed forms. We choose scripts and goals/plans as basic knowledge structures for our connectionist modeling of story understanding because these two knowledge structures are fundamental and well-developed, but play very different roles in story understanding [Wilensky, 1978]. Besides, their symbolic implementations are by no means complete and natural. There are several limitations to symbolic implementations of script processing (e.g SAM [Cullingford, 1978]) which can be overcome by connectionist implementations, including: (1) symbolic scripts are too rigidly defined, so symbolic implementations cannot handle script deviations properly, (2) it is difficult to invoke the right script for the input story fragments using symbolic script headers [Cullingford, 1978; Schank and Abelson, 1977; Dyer et al., 1987], and (3) there is no explanation about how the original script might be automatically acquired in symbolic implementations, so there remain difficult knowledge engineering problems in script formation: e.g. what event should be included in the script and what event should not be? These limitations can be overcome by the flexibility/generalization capabilities, similarity-based associative retrieval, and trainability of connectionist implementations. Moreover, the idea that human beings form their scripts through statistical generalizations from experience conforms well to connectionist rather than to symbolic implementations.

Symbolic goal/plan-based story processing systems such as PAM (Plan Applier Mechanism) [Wilensky, 1978] have suffered from the same scale-up problems as other symbolic rule-based systems, because goal/plan analysis was actually performed by fragile and complex rule manipulations. Those complex rules cannot be learned but must be pre-encoded by knowledge engineers. Connectionist implementations do not use pre-encoded rules to analyze goal/plan structures, but the rules are statistically generalized from the training data in order to recognize goals/plans and to generate goal/plan inference chains.

1.2.4 Improving current connectionist systems

Finally, current distributed connectionist script/goal/plan-based story understanding models are incomplete and restricted, so they need continuous improvements. For example, a number of connectionist script-processing models have been proposed in the past to overcome weaknesses in the symbolic models [Golden, 1986; Chun and Mimo, 1987; Sharkey et al., 1986], but while they have nodes for their objects and events, none of them has the semantics needed for representing constituency of concepts and events in their node representations. They use random bit strings [Golden, 1986] or localist node representations [Chun and Mimo, 1987; Sharkey et al., 1986] as a connectionist representation scheme. Dolan and Dyer [1987] are the first to consider micro-feature based distributed representations in connectionist script processing to build their representations with similarity properties; i.e. similar concepts have similar representations. But as noted in [Dyer et al., 1988; Miikkulainen and Dyer, 1988] micro-features are arbitrary, lack recursive/hierarchical structures and create a knowledge engineering bottleneck.

In the goal/plan analysis field, many kinds of network-based systems have been proposed to overcome symbolic implementation problems, including the use of marker-passing schemes [Norvig, 1986; Hendler, 1988; Sumida et al., 1988] and localist-level connectionist networks [Lange and Dyer, 1989]. But no one has yet implemented a goal/plan analysis system using distributed connectionist networks. One of the main reasons is the lack of appropriate distributed knowledge representation schemes. These limitations of current connectionist implementations supply us with another good motivation for developing a new distributed connectionist representation theory and architectural techniques. These new techniques have been tested in DYNASTY and are presented here.

word /phrase	representations
John	(HUMAN name (John) gender (male))
ate	(INGEST actor x [HUMAN] object y [FOOD])
apple	(FOOD type (apple))
x [human] "ate" y [food]	(INGEST actor x [HUMAN] object y [FOOD])

Figure 1.1: **Symbolic lexicon example.** The representations usually consist of slot-fillers with proper header names. The concept headers (e.g. HUMAN, INGEST, FOOD) restrict the possible bindings as designated in brackets. For each filler, a procedure can be attached to find the proper filler values and bind the variables [Dyer, 1983].

1.3 Symbolic approaches for natural language understanding

Knowledge-based natural language understanding (NLU) [Lehnert, 1988] emphasizes the need for implicit inference generation for comprehension of written text. To generate the inferences implicit in the text, this type of research focuses on representing and applying world knowledge referred to by the sentences, rather than focusing on low level syntactic structures. Hence knowledge-based NLU has a different research focus than either syntax-based computational linguistics or natural language interface (NLI) research. Story understanding is a core task in knowledge-based NLU research. Much knowledge-based NLU research has focused on story understanding: from [Charniak, 1972; Winograd, 1972] to [Dyer, 1983]. In story understanding, scripts and goals/plans are important and constitute basic knowledge structures for modeling the world knowledge needed to understand stories. Below, we will briefly describe the word representations in a lexicon for symbolic natural language understanding systems and script and goal/plan theory as a background to understanding the DYNASTY task/domain.

1.3.1 Word/phrase representations

Word/phrase representations in symbolic NLU systems are organized using symbolic structures. Word or phrase representations are kept in the lexicon along with other information about how the word or phrase should be interpreted in different situations. In general, a symbolic lexicon is a list of word symbols or phrasal patterns with pointers to conceptual memory. The memory contains syntactic and semantic knowledge about the lexicon entry in the form of declarations or procedures [Dyer, 1983; Zernik, 1987; Arens, 1986]. Figure 1.1 shows a symbolic lexicon for parsing the sentence: John ate an apple.

Symbolic systems parse the sentence and build the conceptual structures by hooking up each word/phrase representation from the lexicon. However, each symbol in these lexical representations is only meaningful to the humans in a certain linguistic society. The same symbols are meaningless ASCII strings to the computers. The word symbol itself does not have any semantics, so even though the word "John" and "Bill" are conceptually similar to each other, their ASCII codes are very different. Consequently, these word/phrase representations have to encode every possible relation for the word *explicitly* in their structures. Otherwise, the system becomes fragile and could break down if conceptually similar inputs are given.

1.3.2 Script application

A script [Schank and Abelson, 1977; Dyer et al., 1987] is a knowledge structure that describes appropriate sequences of events in stereotypical situations. According to psychological experiments [Bower et al., 1979], people use scripts to understand and remember narrative texts. A full construction of a script is made up of two parts: (1) *slots* designating script *roles, props, entry conditions and results* with restrictions about what can fill those slots, and (2) *event sequences* based on various *tracks* in a script. For example, roles and props in a restaurant script include diner, waiter, food and tips; tracks include fancy and fast-food restaurants. Script *roles* usually specify animate entities which play essential roles in performing the script, while script *props* are inanimate things that are used in the script (e.g. menu). *Entry conditions* are preconditions that must be met before applying the script, and *results* is the outcome of script application. Once a script is instantiated, the slots are filled with appropriate fillers which satisfy the specified restrictions. So in that sense, scripts are similar to more general slot-filler knowledge representation schemes such as frames [Minsky, 1981] and schemata [Rumelhart, 1975].

In this dissertation, we use a simplified script format which focuses on the script roles and standard event sequences, but without deviations. The standard event sequence consists of *backbone events* in the script. Once the script and track is defined, the backbone event sequence is applied, and all deviations from the normal execution of script backbone events are simply filtered out. Our script roles cover the roles and props in the above definition, but entry conditions and results are ignored. For example, to understand the restaurant story:

```
John entered the Chart-House.  John ordered the steak.  John paid the
bill.
```

the restaurant script should have the following form. Here uppercase designates script roles.

Restaurant-Script
Roles: CUSTOMER, RESTAURANT-NAME, FOOD

```
Event sequence:
CUSTOMER enters RESTAURANT-NAME
waiter seats CUSTOMER
waiter brings menu
CUSTOMER reads menu
CUSTOMER orders FOOD
CUSTOMER eats FOOD
CUSTOMER pays bill
CUSTOMER leaves a tip
CUSTOMER leaves RESTAURANT-NAME for home
```

Here the event sequence forms the backbone of the restaurant script. When the system is given a restaurant story, it produces these backbone events as a paraphrase through a script application process. The symbolic script application process has three major steps: script selection, script instantiation (role-binding), and inference generation. When a (symbolic) script application system reads the above story, it searches the script memory first to select an appropriate script, using some parts of the input story as the script selection header. For example, the first sentence mentions somebody entering some restaurant. This event corresponds to the locale header[2] and matches the first event of the restaurant script. Once the proper script is selected, the system instantiates it by replacing each script role with the proper instance. So CUSTOMER and RESTAURANT-NAME are replaced by John and Chart-House, and the role-bindings are kept in a binding list to be used throughout story processing. Once the appropriate script is selected, the standard event sequence (backbone events) is used to fill-in the unmentioned events in the story during inference generation. Script application is a very powerful mechanism because it reduces the number of superfluous inference paths to be analyzed if the story conforms to the script. If the script knowledge does not guide the inference generation, then we will run into the problem of a combinatorial explosion of inference paths because each event in the story can produce many inferences and inference chains grow exponentially [Rieger, 1975].

1.3.3 Goal/plan application

For every story that conforms to a single script, script application is the most efficient mechanism for story understanding. But it is unlikely that every story consists of only stereotypical situations. Humans can understand situations which are not stereotypical in nature. People can deal with novel situations because they have access to a more general planning mechanism underlying the script. Goal/plan theory [Schank and Abelson, 1977; Wilensky, 1978] was developed to account for this kind of capability. A goal/plan structure is intended to contain general information that will connect events in the story that cannot be connected by use of an available script.

A plan is a knowledge structure which is made up of action sequences serving a specific goal. So a script can be one form of a standard plan structure [Wilensky, 1978]. A goal is a desirable state for the planner which can be realized by using a specific plan. People have many goals in their lives, and have individual priorities about those goals. According

[2]Schank and Abelson described different kinds of script selection headers: precondition, instrument, locale and direct headers. See [Schank and Abelson, 1977] for details.

goal category	example goals
crisis goal	c-health, c-fire, c-storm
satisfy goal	s-hunger, s-sex, s-sleep
preservation goal	p-health, p-position
achieve goal	a-power-position, a-possessions, a-good-job
enjoy goal	e-travel, e-entertainment, e-exercise
delta goal	d-know, d-control, d-prox(imity)

Table 1.1: **Major goal taxonomy.** High priority goals appear first, but the precedence is not a total ordering. Often the precedence depends on the situation.

to Schank and Abelson [1977], the major goal taxonomy looks like table 1.1.

Delta goals designate state-change goals and function to organize knowledge about how to achieve other higher-level goals. For example, a "d-control (food)" goal should be performed to achieve an "s-hunger" goal. Delta goals usually have standard plans (often being scripts). As an example, consider the d-prox goal, which is a goal to change one's physical proximity. The "pb-bus" script can be used as a standard plan to satisfy this goal.

How are these goals/plans utilized to understand stories which describe nonstereotypical or novel situations? Suppose the story reads:

```
John was hungry.  John picked up the restaurant guide.  John got into
a car.
```

This story is not stereotypical (at least for this author). People do not pick up restaurant guides whenever they are hungry. There are many possible actions that people can take when they are hungry, including cooking, going to a restaurant, ordering food, etc. So we cannot expect a script that completely conforms to this story. Moreover, when we apply straightforward inference rules such as found in [Rieger, 1975], we get the wrong conclusion, i.e. that John will eat the restaurant guide. To understand this story, we need a explanation of why John picked up the restaurant guide and why John got into a car. To construct an explanation for John's actions, we need to interpret John's actions in terms of goal/plan knowledge. Figure 1.2 shows the goal/plan knowledge needed to understand the above story. Goal/plan knowledge can be organized using a goal/plan graph. Each node designates the goal or plan structures with variables which are to be bound to instances in the input structures (the variables have a leading ? mark). The graph has two different types of arcs that denote two different relations: plan-for and sub-goal [Wilensky, 1981]. The plan-for relation designates that the goal has a certain plan or that the plan realizes a certain goal. This relation is a OR relation since any plan under the relation can satisfy the goal. The

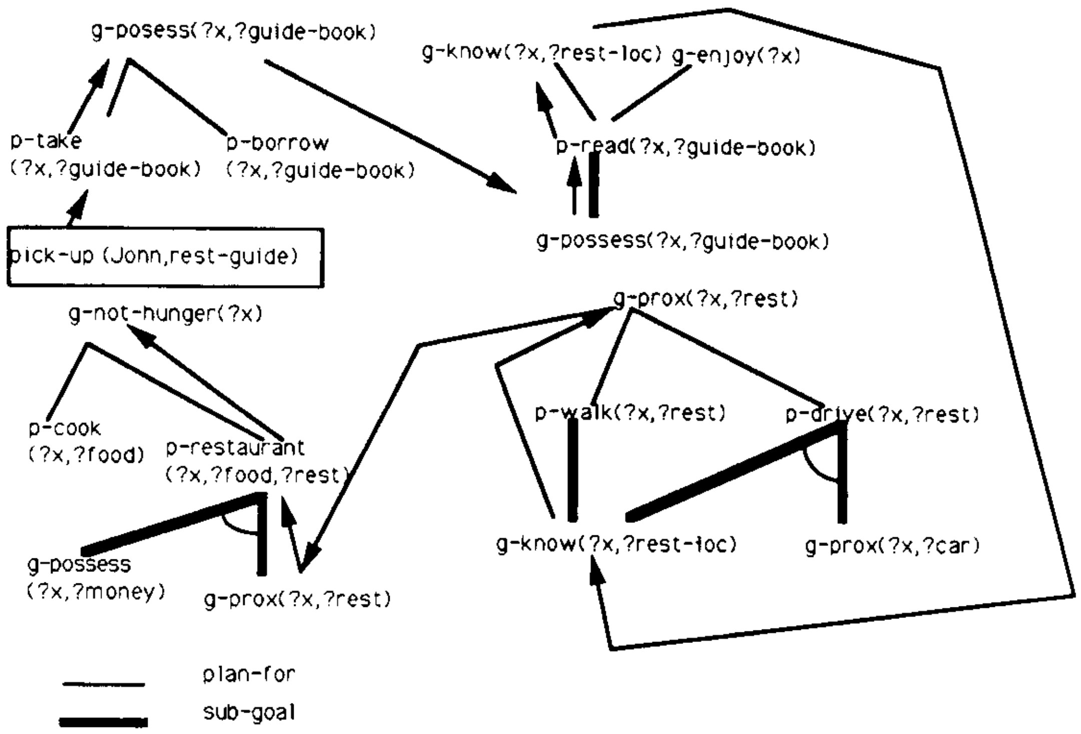


Figure 1.2: **Goal/plan graph to describe the goal/plan relations.** The thin lines designate plan-for relations while the thick lines designate sub-goal relations. Circles in the node junction designate that the sub-goal relations are AND relations. Each variable has a leading ? mark and restricts the class of instances that it can be bound to. The arrows show the built-up inference chain to explain the "pick-up" event by hooking up several goal/plan relation graphs.

sub-goal relation designates that the plan has a certain subgoal. This relation is an AND relation since all of the subgoals must be satisfied to execute the plan. So the goal/plan graph is a AND/OR graph that has goal/plan structures with their relations to one another. Goal/plan analysis consists of 3 steps: (1) search the goal/plan graph, (2) instantiate the goal/plan structures with variable binding and (3) dynamically link the goal/plan nodes to build up an inference chain. In the figure 1.2, the goal/plan inference chain constructed for the second sentence (picking-up event) in the above story is depicted with arrows. The variables ?x and ?guide-book are bound to John and rest-guide respectively, and the bindings are propagated to the other goal/plan structures to instantiate them. The variable types (e.g. ?guide-book) restrict the classes of instances (e.g. rest-guide) which can be bound to the variables. DYNASTY's goal/plan graph consists of AND/OR graphs, and DYNASTY can handle both AND and OR relations using horizontal and vertical reasoning. DYNASTY knows that a certain plan has several preconditions that must be satisfied together (horizontal reasoning) and a certain preconditions must be satisfied sequentially (vertical reasoning) (see section 4.3.2 for the processing model).

The entire goal/plan chain constructed for the above story can be translated as follows:

> John is hungry. John picked up the restaurant guide to read it because he wants to know the restaurant location. John wants to know the restaurant location because he wants to go there to eat the food. John walked to the car to drive to the restaurant.

Goal/plan-based story understanding systems generate explanations of a planner's actions, and the explanations consist of relating actions to goals by means of plans. Symbolic systems such as PAM [Wilensky, 1978] represent goal/plan graphs using "request" structures, a special form of if-then rules. The system uses four different types of "request" rules [Wilensky, 1981]: (1) the *instantiation rules* that relate events to the plans they may be part of, (2) the *planfor rules* that relate plans to the goals to which they may be applicable, (3) the *subgoal rules* that relate goals to the plans to which they may be instrumental, and (4) the *initiate rules* that relate themes to the goals they may give rise to. The following four different types of rules show examples of the rules used to process the second sentence in the story above. The numbers in the parentheses designate rule types.

(1) if a person picked-up something, the person is using a take-plan.
(2) if a person wants to possess something, then one of the plans is a take-plan.
(3) if a person wants to execute a read-plan, then the person should possess a book.
(4) if a person is hungry, then the person has a not-to-be hungry goal.

Using these four different types of rule structures with an explanation-finding algorithm, plan application systems build goal/plan inference chains by dynamically connecting the paths in the goal/plan graphs. Symbolic plan application theory defines the basic story understanding cycle as follows (modified from [Wilensky, 1978]):

Explanation-finding algorithm for one action:

1. Is the action part of a known plan? If yes, the plan is an explanation. If no, go to next.

2. Can a plan be inferred from the action? If yes, go to next. If no, stop.

3. Can the inferred plan be a plan for known goal? If yes, the plan is an explanation. If no, go to next.

4. Can a goal be inferred from the plan? If yes, go to next. If no, stop.

5. Can the inferred goal be instrumental to a known plan? If yes, then the plan-goal chain is an explanation. If no, go to next.

6. Can the inferred goal have a known theme (e.g. default recurring human goal such as s-hunger)? If yes, the plan-goal-theme chain is an explanation. If no, go to next.

7. Can a plan be inferred to which the inferred goal is instrumental? If yes, go to step 3. If no, stop.

We will follow this algorithm with the above sample story. First, the initial rule connects the hungry event to the hungry theme, which is now automatically explained. Next, the second event is converted to the take-plan through the instantiation rule (step 2). Since the take-plan is not known yet, the possess-goal is inferred through planfor rule (step 4). This goal is still unknown, so the read-plan is inferred through subgoal rule (step 7) and the cycle repeats until the system infers the known goal or plan. The search can be in a depth-first fashion or breadth-first fashion [Wilensky, 1978]. For instance, the read-plan has two goals associated with it: know-goal and enjoy-goal (see figure 1.2). The depth-first approach tries one of them and continues to expand the node. If there is no rule applicable, the system backtracks to the original point, and tries another rule. The breadth-first approach tries two of them in turn, and expands the nodes from both goals. One of the paths will be cut off later if the node has no applicable rules any more.

Sometimes we need to deal with goal interactions to understand the stories which involve several planners at the same time. Goal interaction theory deals with several interesting cases, such as goal-subsumption, goal-conflict, goal-concord, and goal-competition [Wilensky, 1978]. DYNASTY currently does not deal with goal interactions or theme analysis [Dyer, 1983] but focuses only on a single planner case with a single top-level goal.

1.4 Distributed connectionist approaches

1.4.1 Symbolic vs. connectionist computation

Symbolic computation has its theoretical basis in the *Physical Symbol System Hypothesis* (PSSH) [Newell, 1980]. In the PSSH, cognition is modeled by syntactic operations on abstract symbols and symbolic structures (expressions), where the operations are those which can be efficiently carried out in a traditional von Neumann computer. Symbolic computation has long been a major computational paradigm in the cognitive sciences, and uses the metaphor of a von Neumann computer as the brain and software as the mind. Formally, symbolic computations must consist of the following six elements [vanGelder, 1989]:

- A basic vocabulary consisting of a finite set of disjoint symbol classes (types); each class has an unbounded number of symbol tokens
- A finite set of syntactic rules
- A concatenative mode of combination
- A set of expression classes (types), such that each class contains an unbounded number of expression tokens, and where expression tokens are generated from symbols by concatenation according to the syntactic rules
- Primitive semantic assignment to symbols

- Principles for making semantic assignment to every expression based on the semantics of the symbols (compositional semantics) and the syntactic structure of the expression

These elements define an extremely large class of computational disciplines used in linguistics, computer science and AI. For the past twenty years, from SHRDLU [Winograd, 1972] to OpEd [Alvarado et al., 1990], knowledge-based NLU systems have been implemented based on this symbolic computational paradigm. Recently, an approach which seems to be incompatible with the PSSH has been introduced, called *Parallel Distributed Processing* (PDP) [Rumelhart et al., 1986c] or *Connectionism* [Feldman and Ballard, 1982]. In this approach, cognition is modeled by networks of highly connected simple processing units. Signals are propagated along connections of different strengths, and are excitatory or inhibitory. The signals arriving at a given unit are summed and the unit fires if the signal exceeds a threshold. Formally, connectionist computing consists of the following eight major elements [Rumelhart et al., 1986a]:

- *A set of simple processing units*: usually three types (input, hidden, output) of units.
- *A state of unit activation*: discrete or continuous activation values of units.
- *Output of the units*: output function maps the current activation state to an output signal.
- *The pattern of connectivity*: represented by a weight matrix in which the entry w_{ij} represents the strength (positive or negative) of the connection from unit u_j to u_i.
- *Propagation rule*: calculating net input to a unit as the weighted sum of the outputs of connected units.
- *Activation rule*: a net input is combined with the previous activation to produce the new activation of the unit.
- *Training rule*: modifying the pattern of connectivity as a function of training data (experiences).
- *Representation of the environment* in which the network operates.

Figure 1.3 shows the operation of a connectionist unit. The net-input to a unit is calculated as some function (usually a weighted sum of the outputs of the neighboring units). This net input is converted to an activation value through some activation function F. Finally the activation value is converted to the output of the unit through some output function f (usually a sigmoidal function), and propagated to other neighboring units. In this model, the knowledge resides in the connectivity pattern (of weights) hence the name "connectionism". Each unit is very simple and restricted in its operations, such as summing and thresholding.

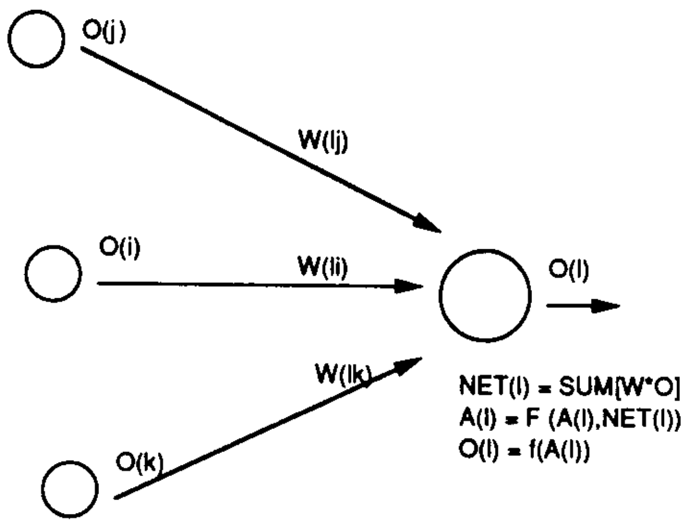


Figure 1.3: **Connectionist computational unit.** Computation in each node consists of 3 different calculation steps: net-input, activation, and output calculations (see text).

1.4.2 Localist vs. distributed connectionism

There are two different, but not incompatible representational approaches in the connectionist research: (1) *localist* representation approaches [Feldman and Ballard, 1982] use one unit to represent each conceptual level entity, and (2) *distributed* representation approaches [Rumelhart et al., 1986c] use an ensemble of units to represent a conceptual level entity, where one unit contributes to representing many different entities at the same time, so no single unit is critical in representing entities. Connectionist research that utilizes distributed representation approaches is called *distributed connectionism* [Touretzky and Hinton, 1988].

There has been an ongoing debate over whether distributed or localist representations should be used to represent high-level knowledge in connectionist cognitive systems. Feldman [1986] has given arguments against both extreme localist and extreme distributed representations. PDP (Parallel Distributed Processing) researchers, such as Rumelhart et. al. [1986c], have listed numerous advantages that distributed representations have over localist representations. At the same time, a number of techniques have been developed for forming distributed representations. Such representation-forming techniques include backpropagation [Rumelhart et al., 1986b], extended backpropagation [Miikkulainen and Dyer, 1988], conjunctive- and coarse-codings [Hinton et al., 1986], microfeature-based representations [Waltz and Pollack, 1985; McClelland and Kawamoto, 1986], and tensor product representations [Dolan and Smolensky, 1989; Smolensky, 1987a].

Localist connectionism is inherently limited in coping with systematicity and compositionality, which are essential in high-level cognitive systems [Fodor and Pylyshyn, 1988]. Distributed connectionism has more potential than localist connectionism in this respect. There are a couple of reasons we are interested in distributed connectionism (DC) rather than localist connectionism (LC): (1) In DC, the representation can be automatically learned

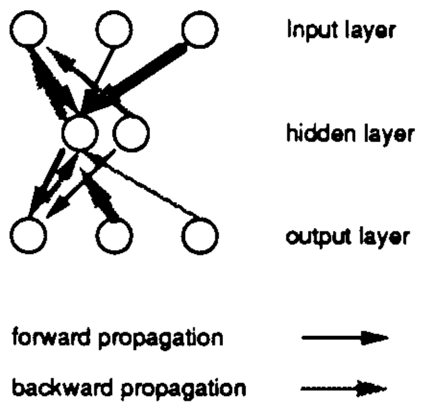


Figure 1.4: **Backpropagation network.** The algorithm consists of two stages: forward propagation to calculate the activation of each node, and backward propagation to adjust the weights between each layer. Each layer is fully connected to the next layer. The adjusted weight values are shown by the thickness of the weight lines. Not all the connections are shown. Only the connections to the first unit in each layer are shown here.

from the training data [Hinton et al., 1986; Miikkulainen and Dyer, 1988; Lee et al., 1990]. In LC, the representation still requires complex and careful knowledge engineering e.g.[Shastri. 1987]. (2) In DC, there are well defined training algorithms to define network connectivity [Rumelhart et al., 1986b; Ackley et al., 1985]. In LC, the network connectivity must still be defined by hand [Lange and Dyer, 1989; Cottrell and Small, 1983; Waltz and Pollack, 1985]. However, the LC scheme can exploit knowledge-level parallelism [Sumida and Dyer, 1989], but DC systems, e.g. [Touretzky and Hinton, 1988] are sequential at the knowledge-level [Feldman, 1986].

1.4.3 Backpropagation network

Backpropagation (BP) [Rumelhart et al., 1986b] is a learning algorithm for a multilayer perceptron network [Minsky and Papert, 1988]. The algorithm repeatedly adjusts the weights in the network so as to minimize a global difference (error) between actual outputs and desired outputs. Figure 1.4 shows a backpropagation-based network architecture. The network consists of three layers: input, output, and a hidden layer.

During performance (forward propagation from input to output layer), the algorithm computes the activation values of the units in each layer. The total input to unit j, x_j, is a linear function of all the outputs from units i that are connected to j, and of the weight, w_{ji}, on these connections:

$$x_j = \sum_i y_i w_{ji} \tag{1.1}$$

A unit j has a real-valued output y_j that is a non-linear differentiable function of its total input. The commonly used non-linear differentiable function is the so-called sigmoid squashing function since it limits activity values of a unit in a certain range (usually 0 to 1). Here

θ_j designates a bias of the sigmoid (its displacement from the origin) at unit y_j. Now the output for unit j looks like:

$$y_j = \frac{1}{1 + e^{-(x_j + \theta_j)}} \tag{1.2}$$

During training (backward propagation from the output to input layer), the algorithm adjusts the weights between the layers according the following equation.

$$\Delta w_{ji} = \eta \delta_j y_i \tag{1.3}$$

where η is a learning rate, δ_j is a difference between desired output and actual output of unit j, and y_i is the output of the unit i. The δ_j can be rewritten as follows according to the layers when the sigmoid function is used.

(1) when unit j is at the output layer:

$$\delta_j = (d_j - y_j) y_j (1 - y_j) \tag{1.4}$$

(2) when unit j is at the arbitrary hidden layers:

$$\delta_j = y_j (1 - y_j) \sum_k \delta_k w_{kj} \tag{1.5}$$

where d_j is a desired output of unit j, w_{kj} is the weight between unit k and unit j (unit k is a next layer of unit j in forward direction). The term y_j (1 - y_j) is a differentiation result of the sigmoid function.

The major problem with BP is its slow convergence rate. To get around the speed problem, one can use an acceleration method in which the current gradient is used to modify the velocity of the points in weight space [Rumelhart et al., 1986b]. The weight adjustment equation now reads:

$$\Delta w_{ji}(t) = \eta \delta_j y_j + \alpha \Delta w_{ji}(t-1) \tag{1.6}$$

where t is increased by one for each epoch through the whole set of input and output pairs, and α is an exponential decay factor (called momentum) between 0 and 1 that determines the relative contribution of the current and past gradient to the weight change.

The BP learning algorithm is entirely deterministic, so if two units within a layer start off with the same connectivity and weight, there is nothing to make them differ from each other. This symmetry can be broken by starting with small random initial weights. Another problem with steepest gradient descent procedures such as BP is that they cannot guarantee finding the global minimum in the error space. If the error space is complex and contains many ravines (local minima), the BP algorithm can get trapped in a local minimum. The local minimum problem can become more severe in large scale applications. Fortunately, there are not many cases in which BP has gotten stuck in local minima in real applications [Rumelhart et al., 1986b]. BP has been successfully applied to many problems, including conversion of English text to speech [Sejnowski and Rosenberg, 1986] and past tense learning [Rumelhart and McClelland, 1987].

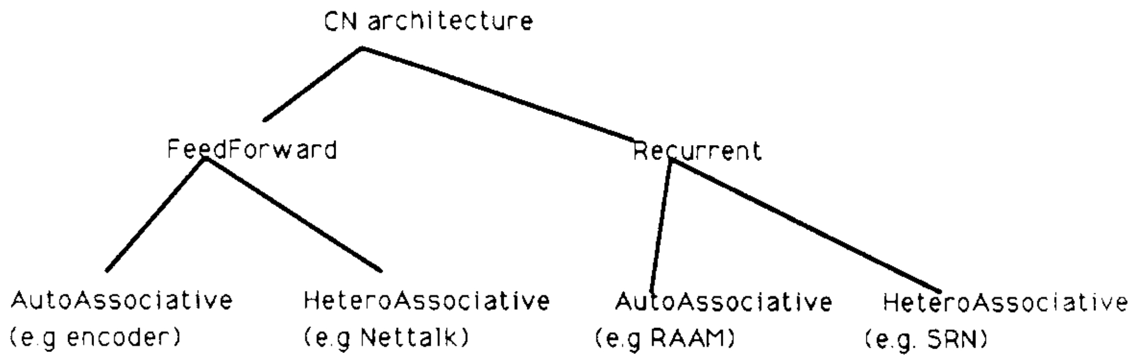


Figure 1.5: **Classification of connectionist network architectures.**

1.4.4 Connectionist network architectures

Connectionist architectures can be classified using two different architectures: feed-forward vs recurrent and auto-associative vs hetero-associative (see figure 1.5).

A feed-forward architecture has only feed-forward connections, but a recurrent architecture has backward connections from the hidden (or output) layer to the input layer. A auto-associative architecture has the same input and output patterns, and the network's objective is to make compressed patterns in the hidden layer. A hetero-associative architecture has different input and output patterns and associates the input and output patterns. The combination of two different architectures can give four different kinds of connectionist network architectures: (1) auto-associative feed-forward (AF), (2) hetero-associative feed-forward (HF), (3) auto-associative recurrent (AR), and (4) hetero-associative recurrent (HR) network architectures. For example, the encoder-network [Rumelhart et al., 1986c] that builds compressed hidden layer representations is an example of AF architecture. The nettalk [Sejnowski and Rosenberg, 1986] has an HF architecture. RAAM (recursive auto-associative memory) [Pollack, 1988] is an example of an AR architecture. SRN (simple recurrent network) [Elman, 1988; Jordan, 1986] is an example of an HR architecture. DYNASTY uses several modules of different types, such as AR, HR and HF architectures (see chapter 3).

1.4.5 The appeal of connectionism

Why has this simple-unit-complex-connectivity computational model gained popularity as a cognitive system modeling? Compared to symbolic computation, connectionist computation has many advantages [Dyer, 1990b; Dyer, 1990a] as a cognition modeling theory and those advantages are listed below.

- Automatic learning: A connectionist network can be automatically trained by using several kinds of training algorithms [Hinton, 1987].

- Automatic generalization: The trained network can perform similar tasks without further training, so generalization is automatic.

- Fault tolerance: A connectionist system can tolerate noise/faults arising from either input or intermediate component failures.

- Graceful degradation: In the face of unit/weight failures, performance degrades gracefully according to the portion of failures.

- Natural mapping to brain's architecture: A connectionist unit is a mathematical simplification of neurons in the human brain.

- Massive parallelism: Each connectionist unit can sum its input and compute its output at the same time as the other units.

- Natural soft constraint satisfaction: A connectionist network can naturally express varying degrees of constraint with continuously varying values for weights and activations.

- Real-time application: Once trained, the network can be used in real-time applications. since its execution is very fast and hardware implementation is natural.

1.5 Methodology

Our methodology is based on what we call a connectionist/symbol processing approach to story understanding, that is, *knowledge and design constraints come from the symbolic system theory and the system is implemented using connectionist components* [Dyer, 1990b]. Then why not base our design on pure symbolic systems or on pure connectionist systems? The reasons are two-fold. First, connectionism provides many fascinating properties in AI applications compared with the pure symbolic systems. However, natural language understanding still needs symbols or symbolic properties such as structure encodings, recursiveness, pointers, and variables [Dyer, 1990b; Dyer, 1990a]. So we need to simulate symbolic behaviors using connectionist components.

In developing a connectionist story understanding system, we follow the classical procedures developed by Schank and his colleagues [Schank and Riesbeck, 1981; Wilensky, 1978; Dyer, 1983; Lehnert, 1978; Alvarado, 1990], that is, first develop basic knowledge representation formalisms such as conceptual dependency, scripts, goals/plans [Schank, 1973; Schank and Abelson, 1977] and then implement story understanding systems which employ those knowledge representation formalisms. We first develop a distributed connectionist knowledge representation scheme called distributed semantic representations (DSRs) for the basic word level and event/state/goal/plan level, and then build modular connectionist architectures using this representation scheme to process script/goal/plan-based stories.

This thesis presents a new method for developing distributed connectionist representations of symbolic structures in order to serve as an adequate foundation for constructing and manipulating conceptual knowledge, along with modular connectionist architectures to process script/goal-based stories. In our approach, distributed representations for word symbols (and symbolic structures) are formed from a group of propositions which describe DYNASTY's domain knowledge. Technically, the representations are formed by *recirculating* [Lee et al., 1990] the hidden layer in two auto-associative recurrent (AR) networks. One of the networks encodes the symbols with respect to the propositions in which the symbols are involved. The other network encodes the propositions themselves which describe the symbols. For example, the meaning of the symbol "coffee" can be partially represented via the two propositions: (p1) "John drinks coffee" and (p2) "Coffee is hot" as long as we have the representations of p1 and p2 ready to be accessed. The accuracy of the representations correlates with the number of provided propositions. For example, when we add the third proposition (p3) "America imports coffee from Columbia", then the representation of "coffee" can now be distinguished from the representation of "tea" which was not possible when we only use the above two propositions. We call these developed distributed representations in the propositional context ***distributed semantic representations*** (DSRs).

We store DSRs in a global dictionary (GD), and use them in the connectionist story understanding system as a common vocabulary. Each module in DYNASTY accesses the same DSRs via this global dictionary. The global dictionary has two entry points, one for the word symbols themselves and the other for the DSRs as representations of the word symbols, so that we can retrieve DSR patterns from the symbolic names and *vice versa*.

The objective of DSR theory is to develop distributed knowledge representations that can be utilized in high-level cognitive systems. Just as von Neumann symbolic representations are utilized as building blocks in symbolic AI systems, we want to use DSRs as building blocks in connectionist or connectionist/symbolic hybrid models [Dyer, 1990b; Dyer, 1990a] which are able to support such tasks as story understanding. On top of the word-level of DSRs, we build high-level (e.g. event, state, goal, plan) representations, and the architecture is designed to make the best use of those high-level representations. In the architecture, our approach is function oriented and similar to Dolan's vertical integration approach [Dolan, 1989] and Dyer's symbolic neuroengineering approach [Dyer, 1990b]. We define each symbolic sub-task (function) first and then modularize the architecture according to the defined symbolic sub-tasks. The functional constraints of the system come from the symbolic text-understanding theory [Schank and Riesbeck, 1981]. Next, we implement each sub-task module in the distributed connectionist framework (except working memory). DYNASTY has been built using this function-oriented approach. At the top (task) level, DYNASTY works like symbolic systems do, but at each module level, its operation is far different from that of symbolic components, with each conforming to the connectionist paradigm. However the control of each module is still symbolic and the episodic memory parts (working memory) are implemented using symbolic pointers.

1.6 A guide to the reader

Chapter 2 deals with our main contributions to the theory of distributed connectionist representations. We provide the theoretical foundation of the DSR scheme and discuss desirable criteria for distributed representations in general, and the formation of DSRs from the domain knowledge (provided as a set of propositions). Also in chapter 2, we discuss the implementation of DSRs within the connectionist framework. This chapter deals with the XRAAM (extended recursive auto-associative memories) architecture, encoding and decoding algorithms, and experiments and analysis of example DSRs from a group of user-supplied propositions.

Part II concerns modular connectionist architectures and processing for story understanding and script/goal/plan analysis. We discuss here the architecture and implementation of the several subsystems of DYNASTY. *Chapter 3* presents DYNASTY architecture, including its linguistic front-ends, such as the parser, the generator, and the global dictionary. Chapter 3 also presents several goal/plan processing modules, including the Triple-Encoder, the Plan-Selector, the Goal/Plan-Associator and the Action-Generator, and shows how these modules are trained to perform the necessary pattern transformations for goal/plan analysis.

Chapter 4 discusses processing in DYNASTY. We discuss here how the various network modules are combined for script/goal/plan processing. The structure matching and variable binding mechanisms are also discussed.

Part III brings the technology discussed in Part I and Part II together to evaluate the entire DYNASTY system, which is then compared to related research. In *chapter 5*, we evaluate DSRs and present performance analysis of DYNASTY. We discuss here how DSRs are learned and altered depending on the training set. In chapter 5, we also discuss the performance of DYNASTY with example output traces. Generalization and fault tolerance abilities are analyzed with separate experiments.

Chapter 6 discusses previous research which influenced the design of DYNASTY. We discuss prior connectionist representations, connectionist systems and symbolic systems and compare them against DYNASTY.

Chapter 7 contains the current status, limitations and possible extensions to DYNASTY. Proposals for future work which can make use of DSRs and modular connectionist architectures are discussed, such as machine translation and question/answering tasks.

Chapter 8 contains a summary of the dissertation, along with major conclusions.

The appendices contain DYNASTY training data and training, execution and analysis code.

Chapter 2

Distributed semantic representations

2.1 Introduction

DSR (Distributed Semantic Representation) is a distributed connectionist knowledge representation scheme that can serve high-level symbolic cognitive tasks, such as natural language understanding. In this chapter, we discuss the background and theoretical foundation for DSRs. We also describe how DSRs are actually learned by using distributed connectionist networks.

Below are some notations which will be used throughout the thesis. The semantic content of a sentence, for example, "The man drinks milk with a straw", can be represented as a proposition, which is a set of [proposition-label, role-name, filler] triples, such as:

[p1 ACT **drink**]
[p1 AGENT **man**]
[p1 OBJECT **milk**]
[p1 INSTRU **straw**]

where each word-concept (e.g. **drink**) has its own DSR representation. The following notation will be used for all triples. First, all triples will be in square brackets, e.g. [**milk** OBJECT p1]. All propositions (or events) will be labeled p1, p2 (or ev1, ev2...), etc. All goals (or plans) will be labeled g1, g2 (or p1, p2...), etc. All semantic case-role names [Fillmore, 1968; Schank, 1973] will be in capital letters (e.g. AGENT, ACT, OBJECT). All word-concepts (abbreviated as w-concept) will be in bold (e.g. **milk, straw**).

2.2 Criteria for forming distributed representations

Developing distributed representations, which are able to support higher-level reasoning and represent conceptual knowledge, is a non-trivial task. Whereas symbolic representations start with a random bit string (like ASCII code) and build structural relationships between meaningless symbols to represent conceptual knowledge, distributed (or so-called "subsymbolic" [Smolensky, 1988]) representations must encode both structures and semantics below the symbolic level, namely, as a pattern in an ensemble of neuron-like elements (i.e. creating a "connectionist symbol"). Our goal is to develop schemes of distributed representations which can serve as an adequate foundation to construct and manipulate conceptual knowledge, which is essential in high-level cognitive tasks. Such a goal calls for a distributed representation scheme which exhibits the advantages of symbolic representations (i.e. portability of representations across modules, structure encodings, variables and propagation of bindings), while at the same time retaining the benefits of distributed representations (i.e.

automatic formation of representations through learning, generalizations based on similarity-based representations, fault-tolerance, and graceful degradation).

To develop such a scheme, we need guidelines to tell us which properties are desirable for distributed representations in high-level cognitive applications. Below we describe four such criteria and analyze previous schemes according to those criteria.

1. *Automaticity* – The representation must be acquired through some automatic learning procedure, rather than encoded by hand. Otherwise one is faced with a knowledge-encoding bottleneck – a problem which has plagued symbolic AI systems and also many PDP systems. For instance, the microfeature-based representations used in PDP systems, e.g. [McClelland and Kawamoto, 1986], does not have an automatic learning procedure. Therefore, connectionists must act as "knowledge engineers" by defining each microfeature in advance and by hand-coding each input representational vector.

2. *Portability* – The representation should be global rather than locally confined to a given training environment. That is, the representation learned in one training environment must be able to be ported to another task environment, so that the same knowledge can be applied to different tasks. For example, the representation in Hinton's family tree example [Hinton et al., 1986] can be said to meet the automaticity criterion, since it is automatically learned by the back-propagation (BP) algorithm. However, these internal representations are local to that task only; they cannot be used in any other task.

3. *Structure Encoding* – Feldman [1986] has argued that any conceptual representation must support answering questions about structural aspects of that concept. For example, part of the meaning of "irresponsible" is that there was an obligation established to perform an action and that obligation was later violated. To answer a question about the meaning of "irresponsible" requires accessing these constituent structures [Dyer, 1990b]. Any conceptual representation must have structural information in the representation itself about the constituent elements of the concept. This structure-encoding criterion implies systematicity, compositionality, and inferential coherence – the three properties that Fodor and Pylyshyn [1988] mention when criticizing connectionism. RAAM [Pollack, 1988] is a good example of distributed representations which encode the entire structure into a fixed length vector representation. For example, from the RAAM representation of syntactical structures, we can decode out constituents of the structures to answer structural questions.

4. *Similarity-based representations* – Distributed representations gain much of their power by encoding statistical correlations inherent in the training set, which are used to characterize the task environment. These statistical correlations give connectionist models the ability to generalize. To support generalization, distributed representations should exhibit semantic content at the micro level, i.e., similar concepts should end up (by some metric) with similar distributed representations. This criterion provided the original impetus for microfeature-based encodings, since similar concepts are similar because they share similar microfeature values. Unlike microfeatures, however, DSRs are formed automatically from training data.

customer entered restaurant-name
John entered Chart-House
Jack entered Chart-House
waiter seated customer
waiter seated John
waiter seated Jack
waiter brought menu
customer read menu
John read menu
Jack read menu
customer ordered food
John ordered steak
Jack ordered chicken
customer ate food
John ate steak
Jack ate chicken
customer paid bill
John paid bill
Jack paid bill
customer left tip
John left tip
customer left restaurant-name for home
John left Chart-House for home
Jack left Chart-House for home

Table 2.1: **Example proposition space in a script-based story processing domain.** The w-concept which behaves similarly in this proposition space (e.g. **customer, John, Jack**) develops similar DSRs. Note that the number of propositions in the proposition space is not enough to semantically distinguish all the words, so the usage of some of the word-concepts (e.g. (bill tip), (ordered ate), (paid left), etc.) cannot be clearly distinguished in this particular proposition space.

2.3 DSR: A new technique for forming distributed representations

DSR [Lee et al., 1989a; 1989b; 1990] is a new technique for automatically forming distributed representations designed to meet the above four criteria. DSRs are formed automatically from the domain knowledge of application systems, where the domain knowledge is represented as a set of propositions. We call such a set of propositions the *proposition space* of the DSRs developed in that application. So if the application domain is script-based story understanding, then all the propositions converted from the script-based sentences (which are to be processed) form a proposition space of the DSRs. The DSRs are automatically formed from a proposition space by using two connectionist networks, which provide necessary similarity-based semantics for the application. In other words, DSRs of two word-concepts are similar if those two words are used similarly in a proposition space from which the DSRs are formed. The union of two proposition spaces form one proposition space for the combined applications. For example, if the application domains are script-based and goal/plan-based story understanding, the propositions converted from both script-based and goal/plan-based sentences form one proposition space. DSRs for these two applications can be formed from this combined proposition space. As a result, knowledge acquired for one task domain can be applied to other task domains. Table 2.1 shows a proposition space for a script-based story understanding domain. From this proposition space, DSRs of the word-concepts in the given propositions can be formed and each word concept has limited semantics in the proposition space.

2.4 Representing propositions and word-concepts

In this section we show how propositions and word-concept are represented in order to be accessed by the connectionist learning architecture. Traditionally, there are two alternate views on the semantic content of words: (1) The *structural view* defines a word meaning only in terms of its relationships to other meanings. (2) The *componential view* defines meaning as a vector of properties (e.g. microfeatures). We take an intermediate position - that word meaning can be defined in terms of a *distributed representation of structural relationships, where each relationship is encoded as a proposition.*

The intuition behind DSRs is based on our observation that people often learn the meanings of words through examples of their relationships to other words. For example, after reading the four propositions below, the reader begins to form a hypothesis of the meaning of the word **foo**.

- p1: The man drinks foo with a straw.
- p2: The company delivers foo in a carton.
- p3: Humans get foo from cows.
- p4: The man eats bread with foo.

The meaning of **foo** should be something like that of milk. The interesting fact is that the semantics of **foo** is not fixed; rather it is *gradually refined* as one experiences more propositions in varying environments. In other words, the semantics of **foo** is based on the *usage* of the word **foo**. The four propositions form a proposition space from which a concept of **milk** can be formed properly. To form reasonable DSRs for the other word-concepts such as **man, drink, straw, company** etc., we need more propositions in this proposition space.

To develop DSRs based on a set of propositions, we have to define the structural relationships between concepts with respect to those propositions. For this purpose, we use thematic case-roles, originally developed in [Fillmore, 1968], and extended in several natural language processing systems, e.g. [Schank, 1973; Schank and Riesbeck, 1981]. Nobody has completely listed the necessary case-roles for natural language semantics yet. Each system tends to have its own case-roles [Bruce, 1975]. Table 2.2 lists the basic case-roles used in DYNASTY for action-oriented propositions. These case-roles for action-oriented propositions can be extended when DYNASTY needs to process the mental states arising from goals and plans as well as actions.

Suppose the previous four propositions form a proposition space for a word-concept **milk** (replace **foo** with **milk** in each proposition). Each proposition is defined as the composition of the constituent thematic case components (case-role plus w-concept) that are themselves combinations of structural relationships with their corresponding meaning representations of other words. For example, the first proposition (p1) can be defined as follows:

p1 = F (G(AGENT, **man**), G(ACT, **drink**), G(OBJECT, **milk**), G(INSTRUMENT, **straw**))

Case-roles	Descriptions
ACT	an action described by a verb
AGENT	an actor intentionally causing the action
OBJECT	a thing that was affected by the action
OBJ-ATTR	a description of the attributes of the object
CO-OBJECT	a secondary object which often modifies the first object
INSTRUMENT	a force or tool used in causing the action
FROM (SOURCE)	the original value in a state change caused by an action
TO (GOAL)	the final value in a state change caused by an action
LOCATION	the spatial region where the action occurred
TIME	the temporal duration in which the action occurred

Table 2.2: **Semantic case-roles used in DYNASTY.**

where the bold typeface is the meaning representation of the word; F is some *integration function* over all case-roles in p1 and G is some *combination function* of structural relationships with respect to the corresponding word concepts. In this formula, p1 is a proposition which has **man** as its AGENT role, **drink** as its ACT role, **milk** as its OBJECT role, and **straw** as its INSTRUMENT role. All other propositions can be represented in the same fashion.

The DSR of **milk** is now defined as the composition of structural relationships, e.g. with respect to the four propositions above. These are then combined as follows:

milk = F (G(OBJECT, p1), G(OBJECT, p2), G(OBJECT, p3), G(CO-OBJECT, p4))

where **milk** is the meaning representation of the milk; F is some integration function over all the propositions involving **milk**; and G is some combination function of structural relationships with respect to the corresponding propositions. For the above two formulas, the arguments of the function G are represented as patterns of activation in two banks of a layer of a recurrent connectionist network. The function G operates by compressing its arguments into the hidden layer of the connectionist network. The function F operates by recycling each compressed pattern (in the hidden layer) back into the input layer. The architecture that implements these operations will be described in full in the next section.

2.5 DSR learning architecture

DSRs are learned using two BP-based recurrent connectionist networks. We use XRAAMs (extended recursive autoassociative memories) [Lee et al., 1990] for automatically learning DSRs. XRAAMs are based on RAAMs, originally developed in [Pollack, 1988]. Pollack showed that RAAMs could be used to encode recursive data structures, such as trees and lists, by feeding the compressed representations in the hidden layer back into the input/output layers. RAAMs, however, lack an external storage for each representation formed. In contrast, XRAAMs make use of a global dictionary (GD) to store and retrieve these compressed representations. The GD is a distributed lexicon network which contains the concept name along with its DSR pattern. The basic idea of XRAAM is to *recirculate* [Dyer et al., in press] the developing internal representation (hidden layer of the network) back out to the environment (input and output layers of the network) using a global symbol memory. Figure 2.1 shows two modules of our architecture, where each one is an XRAAM.

Each XRAAM contains a symbolic/connectionist memory (global-dictionary or proposition -buffer) and a 3-layer AR (autoassociative recurrent) network (section 1.4.4) . The input and output layers of each network have 3 banks of units: bank1, bank2, bank3. These banks represent either a proposition [proposition-label,case-role,w-concept] or a word concept [w-concept, case-role, proposition-label] as triples. After each of the 3 banks is properly loaded with the elements of a proposition, the DSR emerges in bank1 by an unsupervised auto-associative BP [Rumelhart et al., 1986b].[1]

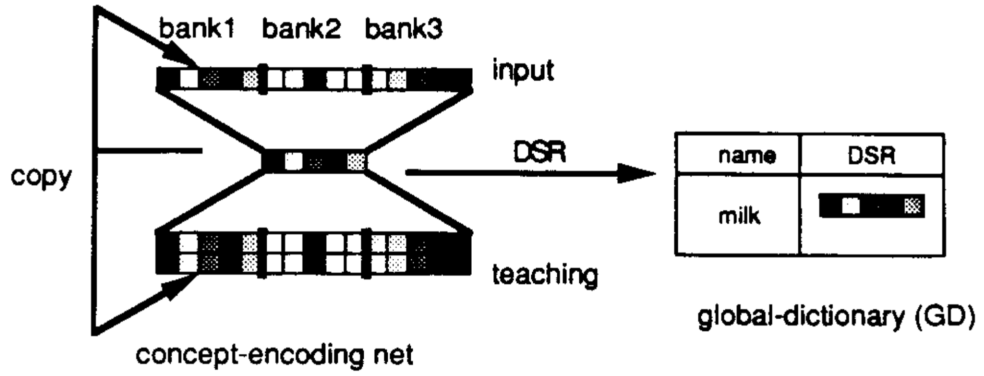
The DSR learning procedure consists of two alternating cycles: Concept Encoding and Proposition Encoding. Below we describe each cycle. In each, all concept and proposition representations start with a *don't know* pattern (i.e. 0.5 in all units), with the activation value range of each unit in the network being 0.0 to 1.0. The case-role representation (for AGENT, OBJECT, CO-OBJECT, etc.) is fixed, using orthogonal bit patterns for minimizing interference (Figure 2.2).

Figure 2.3 shows the information flow during the concept-encoding cycle. Each number in parenthesis in figure 2.3 corresponds to a numbered step in the following concept encoding cycle.

Concept Encoding Cycle:

1. Pick one concept to be represented, say CON1.
2. Select all relevant triples for CON1. In the **milk** example, they should be triples like [**milk**, OBJECT, p1], [**milk**, OBJECT, p2], [**milk** OBJECT p3], and [**milk**, CO-OBJECT, p4]. For the first triple, load the initial representation for CON1 into bank1.
3. Load the case-role into bank2, and load its corresponding filler (i.e. proposition) into bank3. In the **milk** example, for the first triple [bank1, bank2, bank3] is loaded with bit patterns for **milk**, OBJECT, and p1.

[1]When input patterns are used as teaching patterns, BP can be considered to be an unsupervised learning algorithm, since we do not need a separate teaching pattern for each input.



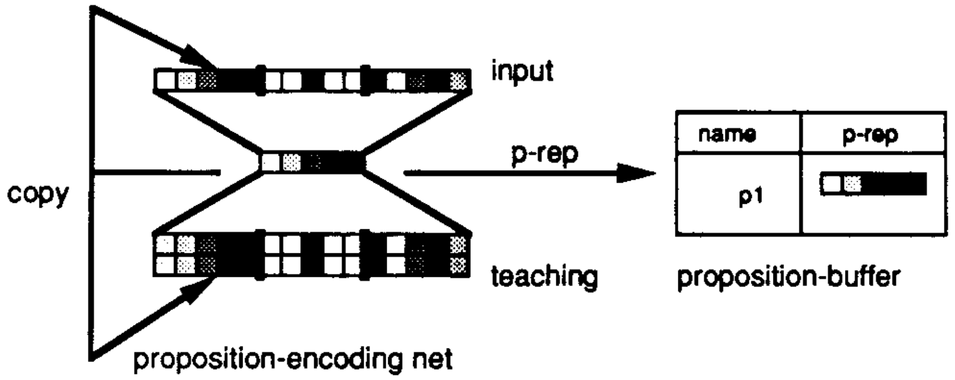


Figure 2.1: **Two XRAAM architectures for learning DSRs.** In each network, there are input, hidden, output, and teaching layers. Once a concept (or proposition) in triple form has been auto-associated, the pattern of activation in the hidden layer units is stored in the corresponding global dictionary (or proposition buffer) as the representation of the concept (or proposition). The concept-encoding network forms a distributed representation of a symbol by encoding all propositions involving that symbol, while the proposition-encoding network forms a representation of a proposition by encoding all the symbols involved in that proposition. Thus, distributed symbols in the global dictionary are fed to the input/output layers of each XRAAM. Each bank is drawn to have only 5 units in this figure due to space limitations. In the actual network, each bank has 10 units.

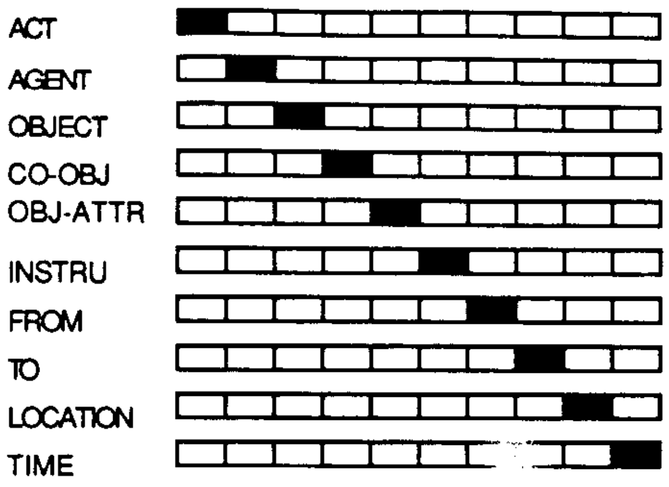


Figure 2.2: **Case-role representations.** An orthogonal encoding scheme is used to minimize interference between case-roles. The fixed case-role representations provide the necessary structural variation for learning each DSR.

4. Run the auto-associative BP algorithm, where the input and output layers have the same bit patterns.

5. Recirculate the developed (hidden layer) representation into bankl of both input and output layers, and perform step3 to step5 for another triple until all triples are encoded.

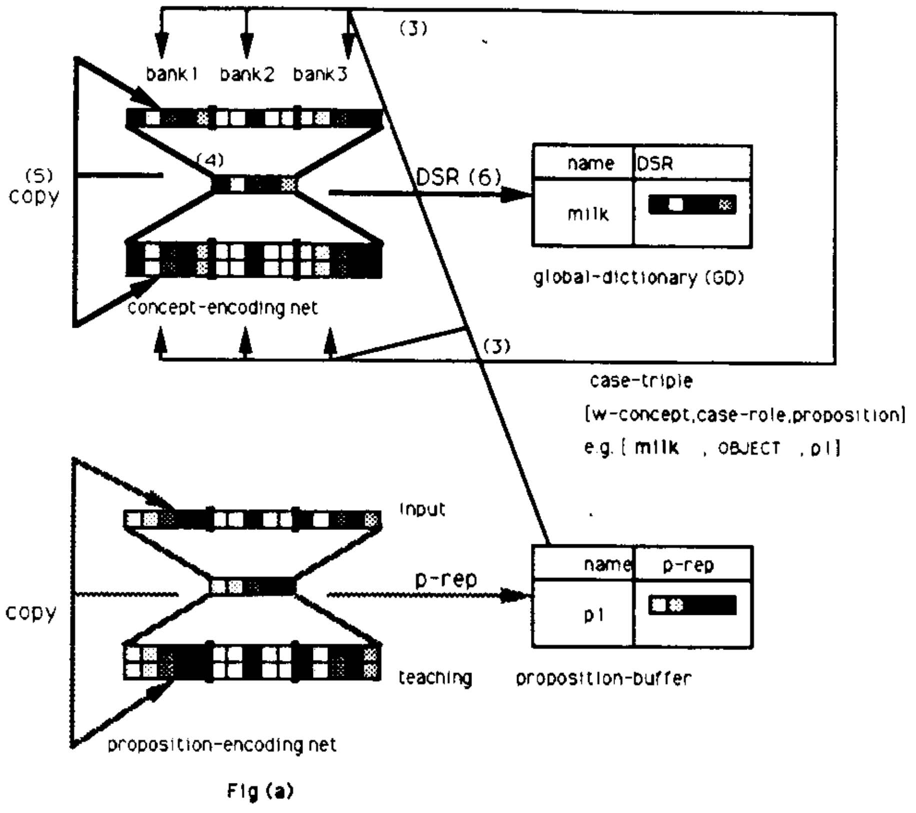
6. Store the developed DSR into the global dictionary and select another word concept to be represented.

Proposition Encoding Cycle: Basically this cycle undergoes the same steps as the Concept Encoding Cycle except that, this time, we load bankl, bank2, and bank3 with (respectively) the proposition to be represented, the appropriate case-role, and its corresponding concept representation (DSR). The result of the encoding is stored in the proposition-buffer. Figure 2.4 shows the information flow in the proposition-encoding cycle.

Notice, to encode a proposition (e.g. p1), the DSRs for all w-concepts (e.g. **milk, straw**) appearing in that proposition must be accessed from the GD and used in the proposition encoding process. Likewise, to form the DSR of a w-concept, the distributed representations of all propositions containing that w-concept must all be accessed from the proposition buffer. So concept encoding relies on proposition encoding and *vice versa.* Consequently, the overall DSR learning process is:

1. Perform the entire *concept encoding cycle.*

2. Perform the entire *proposition encoding cycle.*



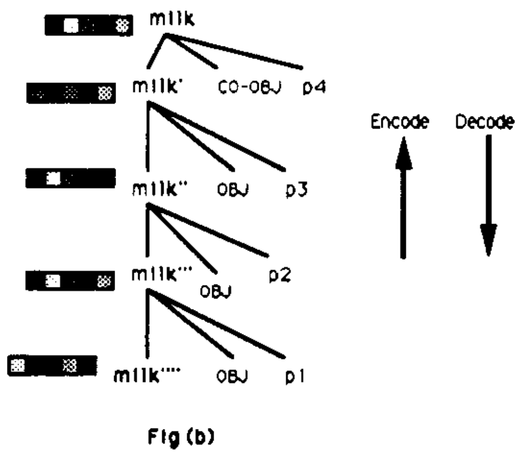


Figure 2.3: **Concept encoding cycle in the DSR learning architecture.** Each number in parentheses designates a corresponding procedure number in the text. The primes on each "milk" in (b) indicate that DSRs are constantly changing to reflect the new propositional relations. The black lines in (a) show information flow in the concept-encoding cycle, while the grey lines show the information flow in the proposition-encoding cycle, which will be described in Figure 2.4. Due to space limitations, each bank is drawn with only 5 units, as before.

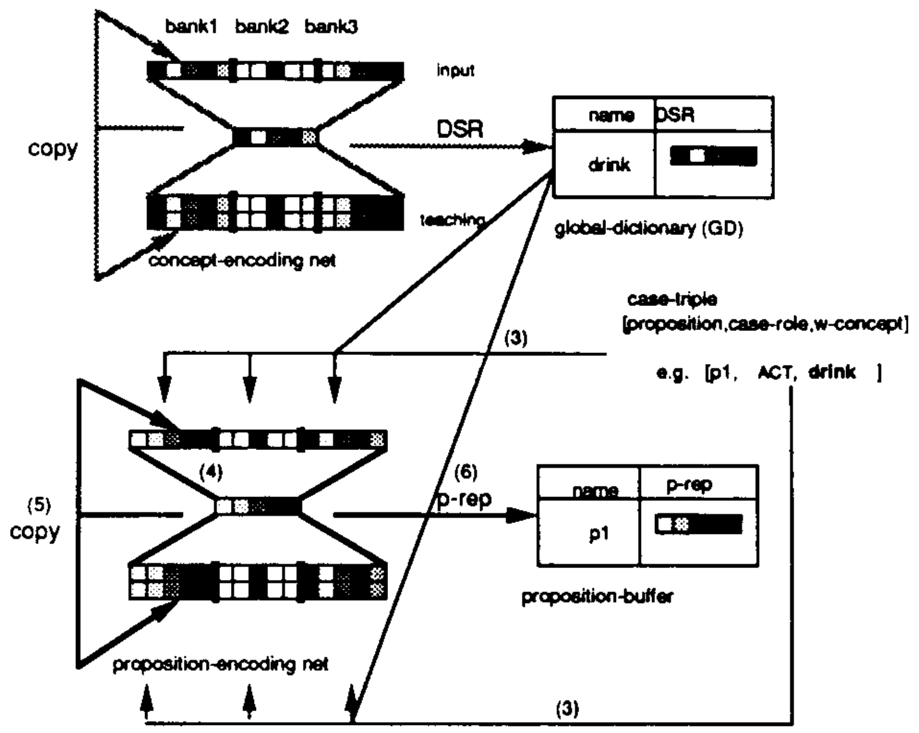


Fig (a)

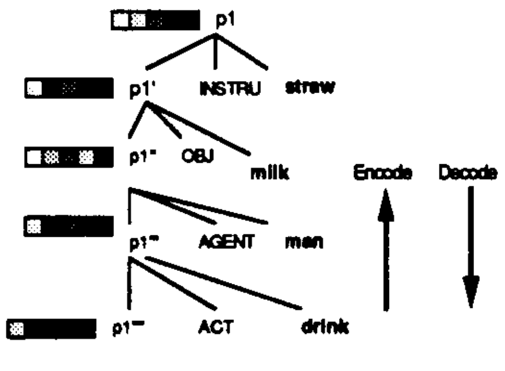


Fig (b)

Figure 2.4: **Proposition encoding cycle in the DSR learning architecture.** Each proposition is encoded using the same algorithm as in the concept encoding cycle, except that the [proposition, case-role, w-concept] triples are used instead of [w-concept, case-role, proposition] triples.

3. Repeat step1 and step2 until we get stable DSR patterns for all the word concepts to be encoded.

In this process, the composition function F (see section 2.4.) is embodied in the dynamics of the RAAM stacking operation [Pollack, 1988; Pollack, 1990] and the combination function G is embodied by *compressing* the concatenation of representations in the three banks. So the XRAAM architecture forms a DSR representation by compressing propositions about a concept into the hidden layer, and then uses those compressions in the specification of propositions that define *other* concepts. Then it *recycles* the compression formed for *this* concept back into the representation of the original concept (doing this over and over until all DSRs stabilize). Thus ***each DSR has in it the propositional structure that relates it to other concepts***, where each of those concepts are also DSRs. The proposition-encoding network provides the necessary propositional representations for w-concept encoding, and the proposition-buffer is a temporary storage for these proposition representations. This *symbol recirculation* method [Dyer et al., in press; Dyer, 1990b; Dyer, 1990a] produces what can be viewed as generalizations of Hinton's "reduced descriptions" [Hinton, 1988].

2.6 Decoding DSRs into the constituents

The decoding process[2] is the reverse process of encoding: We load the concept representations (DSRs) into the hidden layer of the concept-encoding network and perform value propagations from the hidden layer to the output layer until we get the desired case-role relationships in bank2 and propositions in bank3 of the output layer. Next, we load the resulting proposition representations into the hidden layer of the proposition-encoding network and get back the constituent case-role relationships and concept representations. Figure 2.5 shows the decoding architecture.

For example, if the DSR for **milk** is loaded into the hidden layer of the concept-encoding network, then a [**milk'**, CO-OBJECT, p4] triple will appear in the output layer after forward propagation. In the same way, if we copy this interim w-concept representation to the hidden layer, then a [**milk"**, OBJECT, p3] will appear in the output layer (see figure 2.3). Similarly, if we load the p4 representation from the bank3 in the concept-encoding network into the hidden layer of the proposition-encoding network, the [p4', CO-OBJECT, **milk**] triple will appear in the output layer after forward propagation. In this way, all concept and propositional information originally encoded can be extracted by recycling the remaining partial representations (in bank 1) into the hidden layers of each network. Since we can think of each DSR as a stack of (case-role, proposition) pairs, the decoding operation is like a stack-popping operation. We get constituent pairs in a Last-In-First-Out (LIFO) fashion. Once the DSR is completely stabilized during learning, the decoding performance does not degrade when the popping position varies from stack top to bottom.

[2]An advantage of the XRAAM network lies in that we can decode constituent structures from representations by using the same XRAAM network.

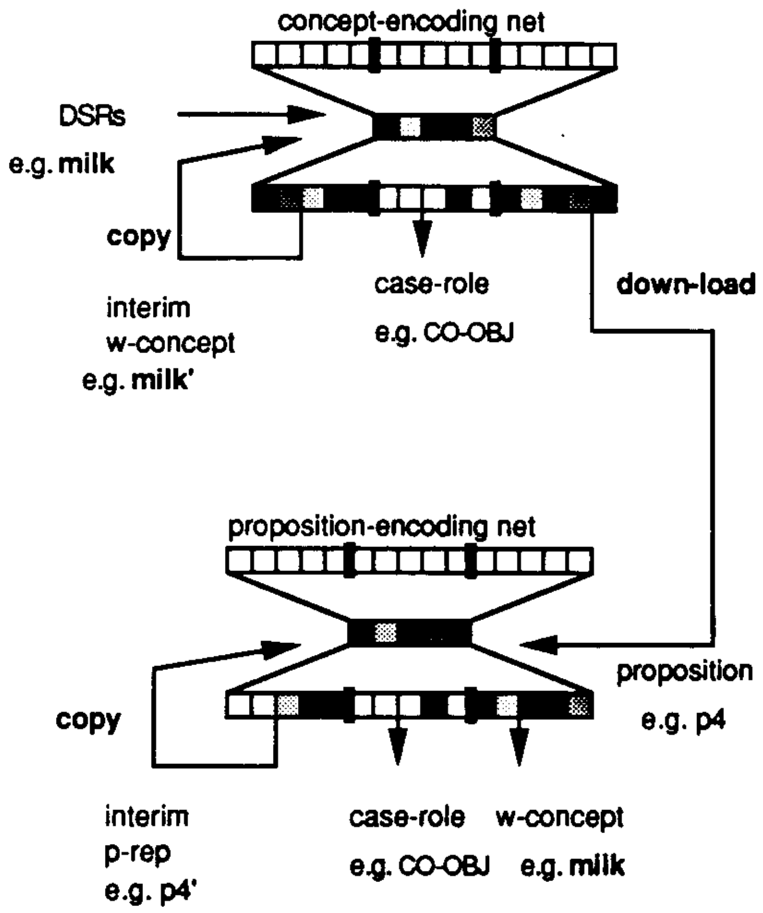


Figure 2.5: **Decoding architecture for DSRs.** The example is according to the tree in Figure 2.3b. The decoding sequence is from the tree root to the leaves (the reverse sequence of encoding).

P numb.	P Generator	Case Structures
p1	human ate	AGENT-ACT
p2	human ate food	AGENT-ACT-OBJECT
p3	human ate food with food	AGENT-ACT-OBJECT-COOBJ
p4	human ate food with utensil	AGENT-ACT-OBJECT-INST
p5	animal ate	AGENT-ACT
p6	human broke fragile-object	AGENT-ACT-OBJECT
p7	human broke fragile-object with breaker	AGENT-ACT-OBJECT-INST
p8	breaker broke fragile-object	INST-ACT-OBJECT
p9	animal broke fragile-object	AGENT-ACT-OBJECT
p10	fragile-object broke	OBJECT-ACT
p11	human hit thing	AGENT-ACT-OBJECT
p12	human hit human with possession	AGENT-ACT-OBJECT-COOBJ
p13	human hit thing with hitter	AGENT-ACT-OBJECT-INST
p14	hitter hit thing	INST-ACT-OBJECT
p15	human moved	AGENT-ACT
p16	human moved object	AGENT-ACT-OBJECT
p17	animal moved	AGENT-ACT
p18	object moved	OBJECT-ACT

Table 2.3: **Proposition generators used in the experiment.** The proposition generators are presented with their proposition numbers and case structures. Each category slot (e.g human) can be filled with any of the concepts in table 2.4 (e.g. man). The OBJECT role in the Case Structures is different from the category name "object" in the Proposition Generators.

Categories	Filler Concepts
human	man, woman
animal	dog, wolf
object	ball, desk
thing	human, animal
food	cheese, spaghetti
utensil	fork, spoon
fragile-object	plate, window
hitter	ball, hammer
breaker	hammer, rock
possession	ball, dog

Table 2.4: **Categories and their filler concepts used in the experiment.**

2.7 Experiments in forming DSRs

We conducted a number of experiments to see how well XRAAM networks learn DSRs for nouns and verbs. We used proposition generators similar to the ones used in [McClelland and Kawamoto, 1986] and made up over 60 propositions, replacing each category by proper fillers in the proposition generators. (Program and data format for the DSR-learner is listed in appendix E.1.) These propositions provide the necessary proposition space for the word concepts to be encoded.[3] We analyzed each proposition's case structure in order to load them into our network architecture. Table 2.3 shows proposition generators with their case structures and table 2.4 shows concept categories with their fillers.

In this simulation, both the concept-encoding network and proposition-encoding network have a 30 unit input layer (each bank has 10 units), a 10 unit hidden layer, and a 30 unit

[3]The proposition generator themselves are included in the proposition space to develop the DSR representations for the category concepts.

output layer. So the DSR and proposition representation sizes are 10 units. Figure 2.6 shows DSRs learned for a number of nouns and verbs. These are snapshots of 120 epochs, where one epoch is 200 cycles of autoassociative BP for each concept and proposition. Notice that the learned representations are similarity-based according to the concept categories, that is, *words in the same semantic category have similar representations* because they behave similarly in the given proposition space. Interestingly, words with multiple categories (e.g. **dog**) develop less similar representations compared with those words with a single category (e.g. **wolf**). This is because words with multiple categories can be considered to have multiple *usages*. For example, the word **dog** is used as both the AGENT and CO-OBJECT in the proposition generators.

In order to see this similarity structure more clearly, we have run a merge clustering algorithm [Hartigan, 1975] on the learned DSRs. The clustering program is adapted from [Miikkulainen, 1990a] and thus is not listed in the appendix. Figure 2.7 shows the clustering analysis results. We can see that the DSRs in the same category start to merge together.

Even if the two DSRs are in the same category, the clustering steps are different depending on the homogeneity of their usages. For example, **cheese** and **spaghetti** are clustered at early time steps since they are mainly used as OBJECT, but **dog** and **possession** are clustered at later time steps because **dog** is also used as an AGENT (in the *animal* category) as well as a CO-OBJECT (in the *possession* category). The somewhat non-intuitive clustering of **human** with **food** can be explained in the same way. The **human** category also has multiple usages, that is, **human** is used as both AGENT and OBJECT (note that **human** is also a concept filler for the *thing* category). But since **human** and **food** are not in the same category, they are clustered at a later step (step 13). As we can see in this experiment, DSRs exactly reflect the usages of each word in the given proposition space. The usages of a word are defined in terms of the semantic roles (e.g. AGENT, OBJECT, etc) the word played in the given sentences. Since each semantic case-role has distinct representations (figure 2.2), each DSR ends up with unique representations unless the two words have exactly the same usages in the given proposition space. Interestingly enough, the representation of each proposition also exhibits similarity structures [Lee et al., 1990], i.e. *propositions involving similar case-roles and fillers have similar representations.* These proposition representations can also be regarded as higher-level representations for event structures. We postulate that this kind of event representation could be used in connectionist schema processing systems such as [Dolan and Dyer, 1987; Sharkey et al., 1986; Chun and Mimo, 1987].

DSRs show many similar characteristics to those reported in [Miikkulainen and Dyer, 1988; Miikkulainen and Dyer, 1989], but unlike FGREP representations, DSRs appear to be more portable because they are directly encoding propositional content. Each DSR can also reconstruct its constituent information through the decoding process. Moreover, DSRs are learned *independent of any particular processing task*, so the representations should be useful in any task requiring access to the propositional content of word meanings. DSRs also show many similar properties to the Recursive Distributed Representations (RDR) discussed in [Pollack, 1988; Pollack, 1990] with respect to recursiveness and structure encoding/decoding. But unlike RDRs, DSRs incorporate word-level semantics so that they can be utilized not only in syntactic-level applications [Chalmers, 1990], but also in conceptual-level applications such as script and goal/plan-based story processing.

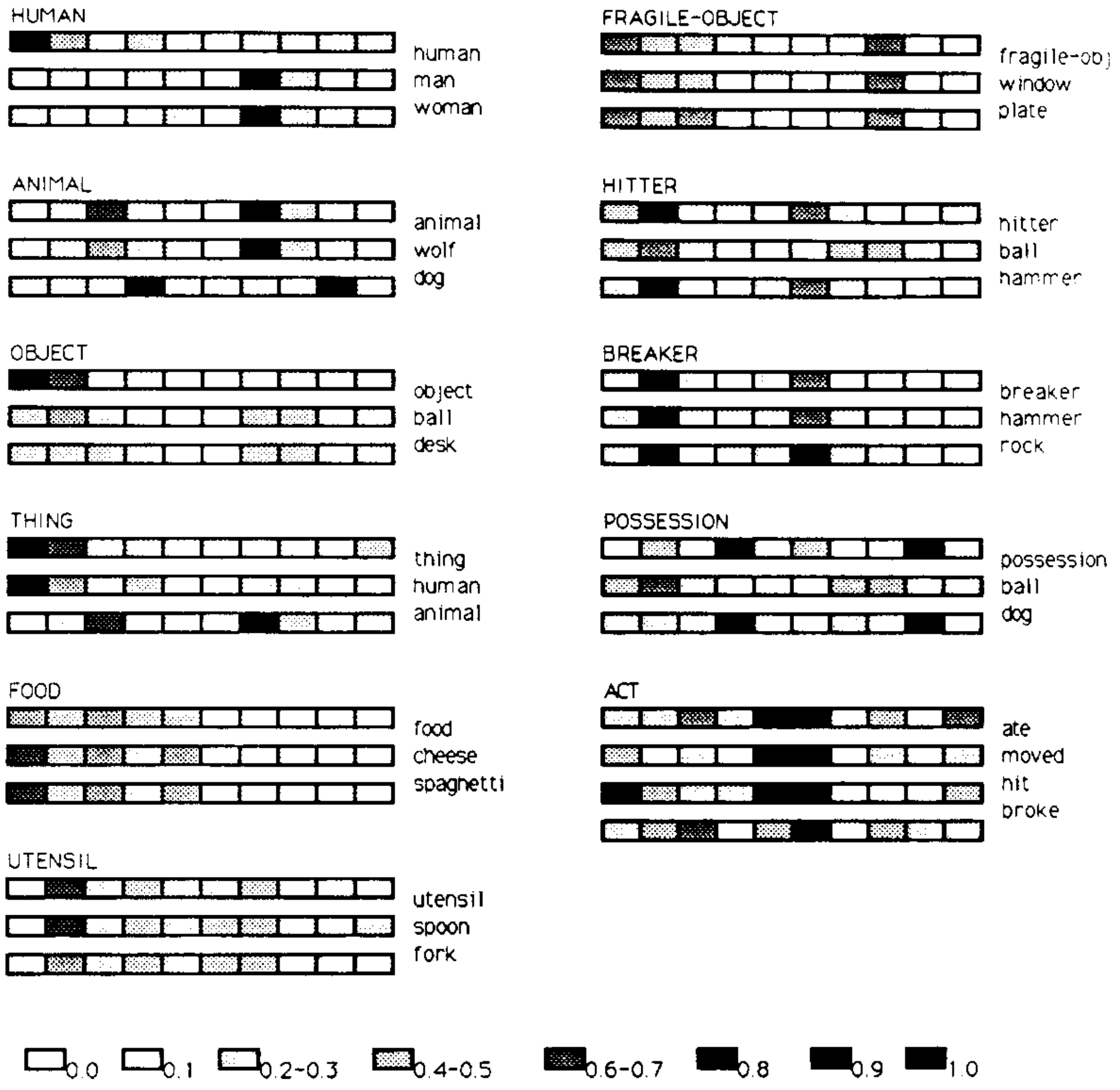


Figure 2.6: **Learned DSRs of concepts with their categories.** The experiment was performed using momentum accelerated backpropagation [Rumelhart et al., 1986b, page 330]. The learning rate varied from 0.07 to 0.02; the momentum factor varied from 0.5 to 0.9. There were 120 epochs used for learning each concept and proposition; one epoch is 200 cycles of auto-associative backpropagation. The value range is 0.0-1.0 continuous, which is shown by the degree of box shading.

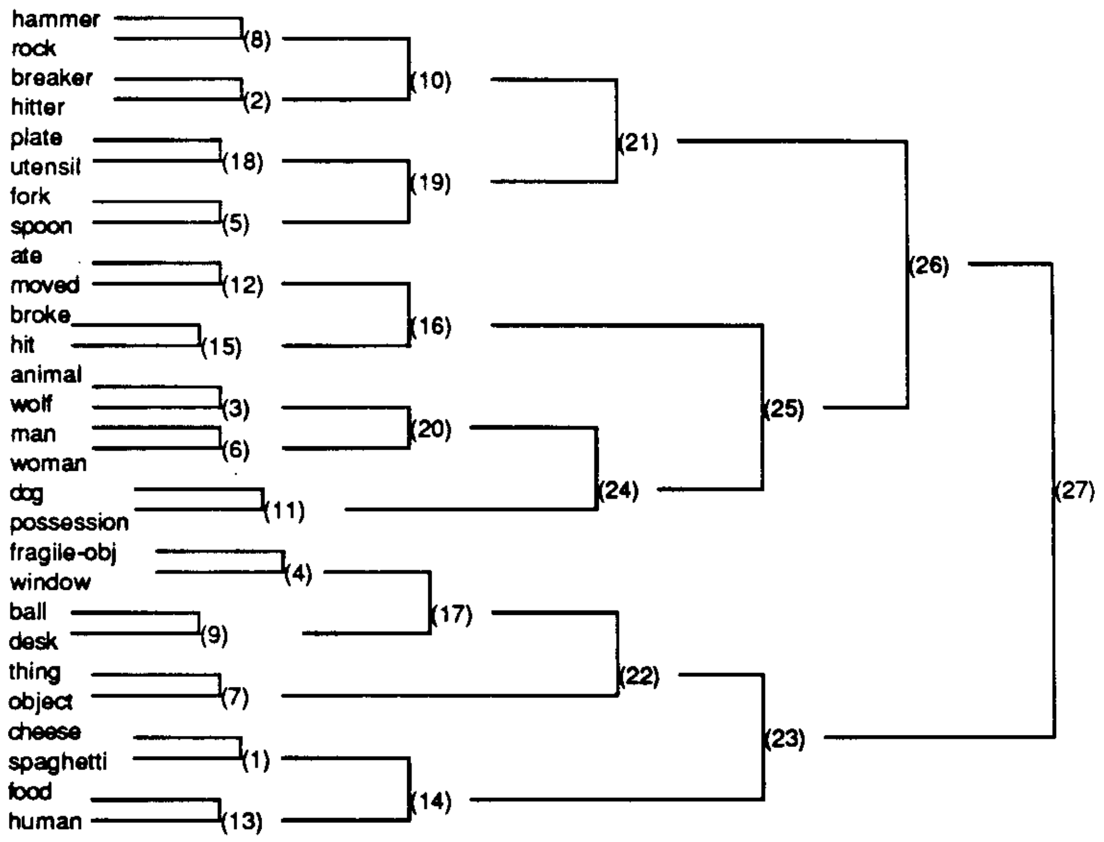


Figure 2.7: **Merge clustering the learned DSRs.** The numbers designate the time step. At each step, the clusters with the shortest average Euclidean distance were merged.

2.8 Properties of DSRs

DSRs have several desirable properties for the high-level connectionist symbol processing systems when compared with previous distributed representations. These properties basically come from the efforts to meet the four criteria describe in section 2.2.

(1) *Automaticity* - DSRs are learned automatically using XRAAMs, rather than built by hand using explicit nodes and links (as in the systems of Schank and his colleagues, e.g. [Schank and Riesbeck, 1981]). In addition, DSRs are better than the hand-coded microfeature-based representations [McClelland and Kawamoto, 1986] in which a PDP knowledge engineer must define each microfeature in advance and hand-code each representational vector.

(2) *Portability* – DSRs are learned without dependence on any particular task, so their encoded propositional contents can be ported to any application environment. In other words, DSRs are global rather than confined to local training environment, and DSRs learned in one task environment can be applied in another task environment. To demonstrate this kind of portability more clearly, we have developed a distributed connectionist story understanding system which can process scripts and goals/plans using the same DSR scheme.

In contrast, the internal representations developed in Hinton's family tree example [Hinton, 1986] are not global and cannot be used in another task environment, even though they are automatically learned.

(3) *Structure Encoding* – DSRs encode propositional structures with constituencies. Since DSRs are learned by stacking case-role and proposition pairs, we can extract the used case-role patterns and proposition patterns from each DSR. These propositions can be decoded again to return the constituent case-roles and concepts. Therefore DSRs support answering structural questions about concepts and events [Feldman, 1986]. Because of these structure-encoding capabilities, DSRs are compositional, that is, the semantics of a DSR representation is a function of its constituent case-roles and propositions. Therefore DSRs can be considered to be a counter example to Fodor and Pylyshyn's criticism concerning connectionism's lack of structure [Fodor and Pylyshyn, 1988]. DSRs can be utilized in structure-sensitive symbolic tasks such as natural language processing, which require propositional semantics for word meanings.

(4) *Similarity-Based Representations* – DSRs are similarity-based, i.e. similar concepts end up with similar representations in the DSR learning process. This is because similar concepts function with similar case-roles for similar propositions. This similarity-based feature supports generalizations, i.e., novel but related inputs will be processed in similar ways. For example, the concept of **milk** functions as a case-role similar to the concept of **juice** in **drink** type propositions. Thus **milk** and **juice** will end up acquiring more similar DSRs. A DSR's similarity structure can be controlled by adding or deleting propositions in the corresponding proposition space, so a DSR has variable similarity structures based on the given propositions in its proposition space.

The eventual objective of DSR theory is to develop distributed knowledge representation schemes that can be utilized in high-level reasoning systems. Just as the von Neumann symbolic representation is utilized as a building block in symbolic AI systems, we want to use DSRs as a building block in connectionist or connectionist/symbolic hybrid models [Dyer, 1990b] which are able to support such tasks as natural language processing. Previous distributed representation schemes such as micro-features [McClelland and Kawamoto, 1986], coarse-codings [Touretzky and Hinton, 1988] and learning internal representations by BP [Hinton, 1986] have limited utility in high-level connectionist cognitive applications such as connectionist natural language understanding. To be useful in performing connectionist natural language understanding tasks, the distributed representations should have both symbolic and distributed features [Dyer, 1990b]. DSR is a new technique to automatically form distributed representations which satisfy some valuable criteria (section 2.2). DSRs have both symbolic and distributed features in the representation, so they can be used in implementing high-level connectionist cognitive systems which can access the propositional contents of word semantics.

2.9 Cloning instances with ID units

The DSR scheme is based on the philosophy that the meaning of a word is determined by its usage. In this scheme, the usage is manifested in the semantic case-roles of the word as used in propositions. No two words can be used exactly the same way in the real world, so in principle, the DSR scheme can develop different representations for all words. However, an AI system can only be exposed to a limited world, and thus separating representations of similarly-used words is a problem [Miikkulainen, 1990a]. For example, in the figure 2.6, the words in the same category have almost identical DSRs (e.g. man vs. woman) unless they have multiple usages (e.g. wolf vs. dog). One method to separate their representations is to attach small random identification (ID) units to each representation, which was first developed in [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press].

The ID+content technique [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press] designates a subset of representation components to maintain distinct identities for each word. Under this technique, the word representation consists of two parts: the content part, which encodes the meaning of the word, and the ID part, which is unique for each instance of the same concept category. This ID+content technique makes it possible to deal with a large and open-ended set of semantically equivalent words without confusing them. During training, the units within the ID parts of the words are set up randomly for each input presentation, and the network is required to produce the same ID pattern at its output. In effect, the network is trained to process any ID pattern in a word representation by passing ID parts unchanged from input to output. Under the ID+content technique, backpropagation network has to learn the identity function as far as the ID part processing is concerned. This identity function turns out to be essential to the variable binding propagation in natural language understanding systems. Once a unique instance (word with specific ID part) is bound to a variable, the binding must be propagated to other structures without confusion, and passing ID parts unchanged throughout the network makes it possible to safely propagate the bindings. The ID+content technique can handle an exponential number of sentences with polynomial costs [Miikkulainen and Dyer, in press] because the number of sentences that can be processed increase exponentially while the number of instances that must be trained increase only polynomially.

We have borrowed this ID+content technique for use in DYNASTY. As a result, word representations are composed of 2 units of IDs plus 10 units of DSR representations (figure 2.8). The word representations in figure 2.8 are obtained using the same proposition generators and concept-fillers listed in table 2.3 and table 2.4. The two XRAAM networks in figure 2.1 (DSR-Learner) developed 10-unit DSR representations for each concept as before, and random 2-unit ID patterns were attached to these DSR representations. The concept category has null (00) ID patterns, and the instances in the same category are assigned with orthogonal random ID patterns. The word representations (ID+DSR) formed are now entered into the global-dictionary, which can serve as common vocabularies for the other DYNASTY modules (chapter 3).

ID units can be viewed as representing unique sensory information and DSRs as providing categorical information for a word concept [Harnad, 1989]. In the DYNASTY system, the ID units help keep role-bindings straight, while DSRs support similarity-based generalizations. For example, the similar DSRs in the representations of **John** and **Mary** support the generalized (predicted) behaviors for the **person** category, and the ID units separate **John** from **Mary** whenever specific bindings need to be propagated (see section 4.5). During training of DYNASTY using the word representations (ID+DSR) in the global-dictionary, the ID parts are set up randomly between 0 and 1, and DYNASTY is trained to process identity function for the ID parts. In other words, DYNASTY is only trained with concept categories with random ID (which is not necessarily specific instances), and tested with specific instances with particular ID patterns for the generalizations (see chapter 3 for details of each module training with IDs).

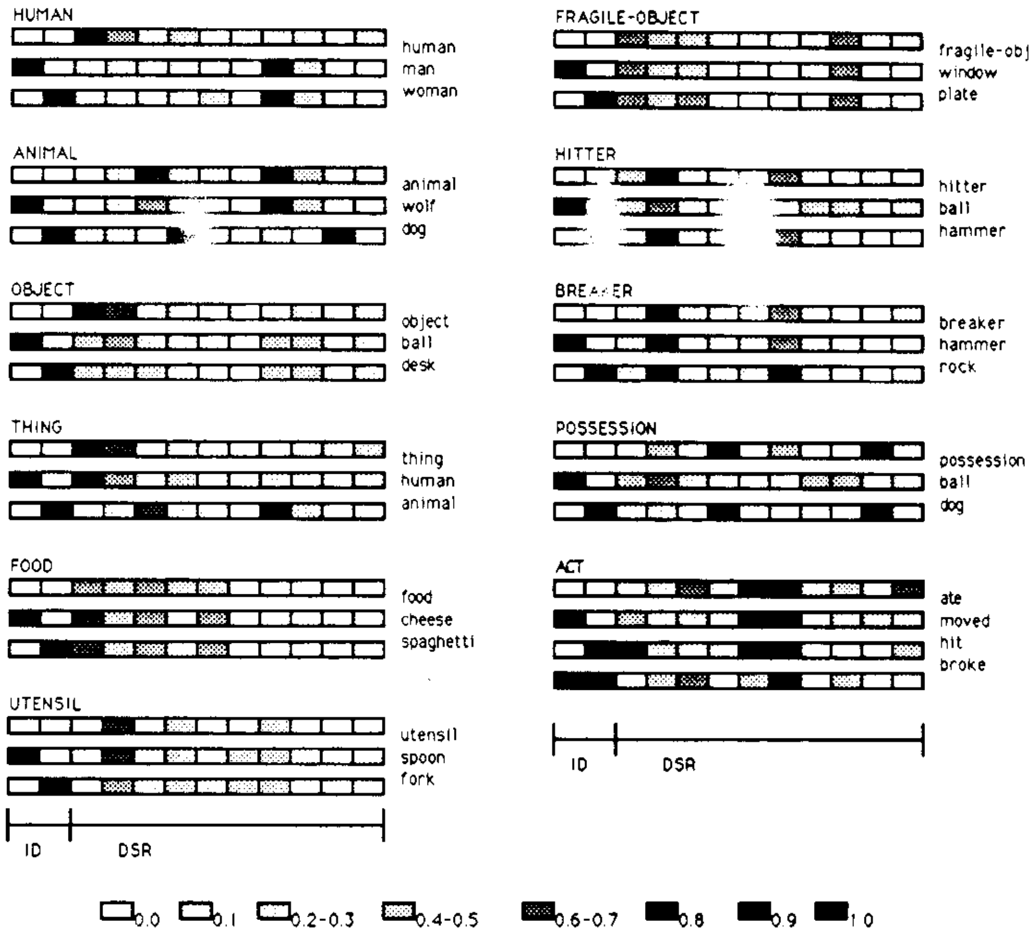


Figure 2.8: **Final representations composed of IDs plus DSRs.** Each 10 unit DSR representation in figure 2.6 is now expanded to a 12 unit distributed word representation. These final word representations are stored into the global-dictionary to give common vocabularies to DYNASTY.

Part II

Architecture and Processing

Chapter 3

DYNASTY architecture

3.1 Introduction

DYNASTY is a large-scale connectionist system that reads in natural language stories as input, and paraphrases them according to their script/goal/plan knowledge. DYNASTY consists of multiple connectionist modules including AR (auto-associative recurrent), HR (hetero-associative recurrent), and HF (hetero-associative feed-forward) architectures in one system (see section 1.4.4). These modules are not based on entirely new architectures. For example, [Pollack, 1988] used an AR architecture, called a RAAM, to generate recursive distributed representations of stacks and parse trees. HR architectures have been used by many researchers for several applications: natural language question-answering [Allen, 1988], parsing [Hanson and Kegl, 1987], and sentence comprehension [St. John and McClelland, 1989]. Here, our HR architecture is most similar to the one used in [Elman, 1988]. What is new is that DYNASTY uses these different connectionist sub-architectures as modular components, communicating via a Global-Dictionary (GD) of DSRs to achieve a high-level task, namely, story understanding. Besides being modular, DYNASTY also employs a functional decomposition approach, i.e. each module is classified according to its function/task in the system.

DYNASTY will be described in two different phases in the thesis: a training phase (this chapter) and a performance phase (chapter 4). In the training phase description, each modular component is fully described, including their general functions and training algorithms. In the performance phase description, we describe how several modules are connected to perform one coherent task.

In DYNASTY, several modules are separately trained to perform the following 3 major subtasks for script/goal/plan analysis during story processing: (1) Parsing and generation between natural language and event-triple forms (linguistic subsystem), (2) Encoding high-level event and script/goal/plan representations using the word-level representations provided in the GD network (representation subsystem), and (3) Generating knowledge-based inferences from the input events using script/goal/plan knowledge (goal/plan analysis subsystem). Figure 3.1 shows DYNASTY's top-level configuration during the training phase.

DYNASTY, during training, consists of 8 modules; organized in 3 subsystems according to the 3 major subtasks. Each module is briefly described below.

The linguistic subsystem has syntactic and semantic knowledge, such as word order and semantic case-role structures, which is needed to do the conceptual analysis and generation of the sentences. The linguistic subsystem contains the following two modules.

Linguistic subsystem:

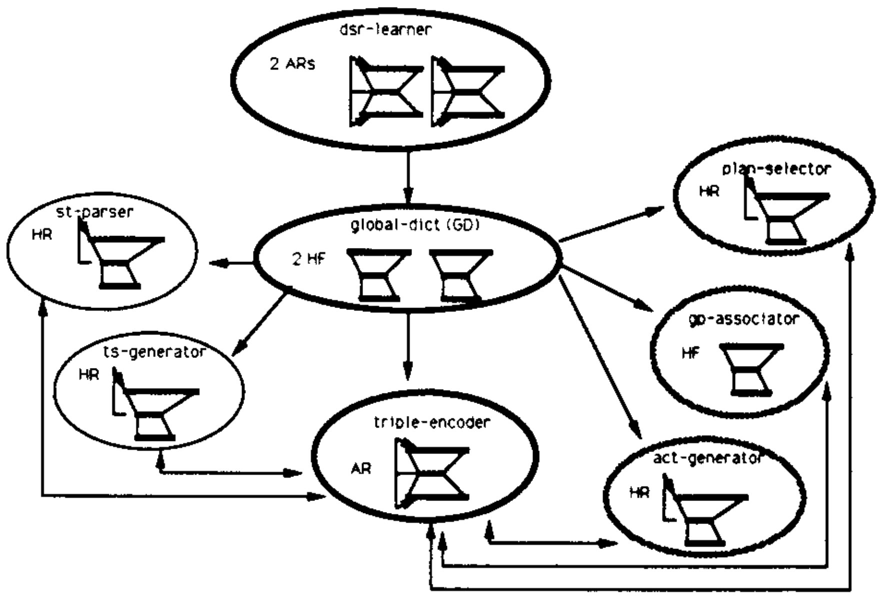


Figure 3.1: **Top level configuration during the training phase.** The arrows designate data flow during training. Each module accesses its own training database. The thick black circles designate the modules in the representation subsystem, and the thick shaded circles designate the modules in the goal/plan analysis subsystem. The thin black circles are for the linguistic subsystem.

- ST-Parser: ST-Parser consists of one HR module. The inputs are word representations in the sentence and the outputs are case-role assignments of the sentence. The general function is to parse input natural language sentences into event-triple forms.

- TS-Generator: TS-Generator also consists of one HR module. The inputs are case-role assignment forms and the outputs are word representations. The general function is to perform the reverse process of the ST-Parser, that is, to convert event-triples back into natural language output.

The representation subsystem has the knowledge necessary to form the semantic representations of word and high-level event, goal, plan structures. It also has the lexical/semantic associative knowledge to do the translation between symbols and distributed representations. The representation subsystem has the following 3 modules.

Representation subsystem:

- DSR-Learner: The DSR-Learner (described in chapter 2) consists of two AR networks (XRAAMs). The inputs are set of propositions and the outputs are DSR representations for each word in the propositions. The general function is to automatically develop distributed representations for each word used in DYNASTY.

- Global-Dictionary (GD): The GD consists of two HF modules. The inputs are ASCII representations and the outputs are distributed representations (ID+DSR) for each word. The general function is to select a distributed word representation when given an ASCII symbol string for a word and vice versa.

- Triple-Encoder: The Triple-Encoder consists of an AR module. The inputs are triples for events and goals/plans and the outputs are 12-unit distributed representations for the triples. The general function is to encode event and goal/plan triples into compressed representations.

Goal/plan analysis subsystem has the knowledge necessary to do the script and plan application. It knows that a certain sequence of events (states and actions) can be organized into a script or plan, and that the goal/plan relations can be organized into recursive tree structures. The goal/plan analysis subsystem has the following 3 modules.

Goal/plan analysis subsystem:

- Plan-Selector: The Plan-Selector consists of an HR module. The inputs are sequence of event representations and the outputs are goal/plan representations. The general functions is to select appropriate goals and plans to be associated with the given event or event sequence.

- G(oal)P(lan)-Associator: The GP-Associator consists of an HF module. The inputs are goal/plan representations and the outputs are related goal/plan representations. The general function is to encode goal/plan relations (plan-for and sub-goal relations described in section 1.3.3) in DYNASTY goal/plan trees.

- Action-Generator: The Action-Generator consists of an HR module. The inputs are goal/plan representations and the outputs are sequence of event representations. The general function is to generate an event (or event sequences) which are results of the given goal/plan applications.

The training phase enables DYNASTY to extract the statistical regularities underlying the training data to process the input stories and to generalize to new stories. According to data dependencies in DYNASTY, there are four hierarchical training steps: (1) training to learn DSRs, (2) training Global-Dictionary to store the word representations (DSRs plus IDs), (3) using the vocabularies of the GD to train the Triple-Encoder module to develop high-level distributed representations for events/goals/plans, and (4) training the remaining processing modules which utilize the Triple-Encoder and the vocabularies of the GD. Each module has its own training data, and is trained in parallel within the data-dependency range. This chapter describes DYNASTY's training phase. The performance phase description will be followed in the next chapter.

3.2 Training-data specification

The training data for each module is maintained in the training database that is related to a given module. The training data is specified in a DYNASTY data-specification language. Each module's control program parses the training data and loads the training patterns into a network architecture. The data-specification language provides a unified view of the training data for each different module, and makes it easier to maintain and modify the training data. The data-specification language has a simple phrase structure grammar with 4 major key words: IF, AND, FOLLOWS and THEN.

Below is listed a phrasal structure grammar for DYNASTY's training data specification language. The *italics* designate non-terminal symbols, and the capitals designate key-words. The ⋆ is a Kleene-star which means repetitive applications including zero application. The *inteachlayer* specification is for auto-associative networks and has a slightly different semantics (see Triple-Encoder in section 3.3.3).

> ***tdata*** ⟶ ***inputlayer teachlayer*** | ***inteachlayer***
> ***inputlayer*** ⟶ IF ***singlelayer*** (FOLLOWS ***singlelayer***)⋆
> ***teachlayer*** ⟶ THEN ***singlelayer*** (FOLLOWS ***singlelayer***)⋆
> ***singlelayer*** ⟶ ***terminal*** (AND ***terminal***)⋆
> ***terminal*** ⟶ ***triple*** (***triple***)⋆ | yes | no | ***digit*** | nil
> ***inteachlayer*** ⟶ ***triple*** (***triple***)⋆
> ***triple*** ⟶ ***triplenumber caserole word***
> ***triplenumber*** ⟶ ev***digit*** | p***digit*** | g***digit***
> ***caserole*** ⟶ STATE | ACT | AGENT | OBJECT | OBJ-ATTR | INSTRUMENT | COOBJ | FROM | TO | LOCATION | TIME | MODE | GOAL | PLAN
> ***word*** ⟶ vocabularies in the GD
> ***digit*** ⟶ any digit between 1 to 100.

The semantics of each key word is as follows:

IF: input pattern start
AND: bank delimiter
FOLLOWS: hidden pattern recirculation to the input layer
THEN: teaching pattern start

Using this specification language, the training data format can also define the network module's architecture. For example, consider the following training data t1 and t2 for the GP-Associator (section 3.4.1) and the Plan-Selector (section 3.4.2) module respectively.

(t1)
IF 0
AND [g1 GOAL s-hunger], [g1 AGENT ?person]
THEN [p1 PLAN pb-restaurant], [p1 AGENT ?person], [p1 OBJECT ?food], [p1 LOCATION ?restaurant]

(t2)
IF [ev11 ACT entered] [ev11 AGENT ?person] [ev11 LOCATION kitchen]
FOLLOWS [ev12 ACT turned-on] [ev12 AGENT ?person] [ev12 OBJECT gas-stove]
FOLLOWS [ev13 ACT cooked] [ev13 AGENT ?person] [ev13 OBJECT ?raw-food]
THEN [p2 PLAN pb-cook], [p2 AGENT ?person], [p2 OBJECT ?raw-food], [p2 LOCATION kitchen]
AND yes

In the training data, each triple group with the same triple numbers is converted into the 12-unit distributed representations by the Triple-Encoder module. The words with ?-mark are variables which have randomly-varying IDs during training. All other words have null (00) IDs during training. Training data t1 defines a HF architecture; the input layer has two banks, with the number 0 and g1 representation loaded, and the output layer has one bank with p1 representation loaded. The HF architecture that can be defined by t1 is depicted in figure 3.6. Similarly, training data t2 defines an HR architecture; the input layer has two banks: context bank and another bank with ev11, ev12 and ev13 representations being sequentially loaded. The output layer has two banks with p2 representation and "yes" (actually binary 1) flag being loaded. The hidden layer pattern will be copied to the context bank while ev11, ev12 and ev13 representations are sequentially processed. The p2 representation and "yes" flag are maintained the same until ev11, ev12 and ev13 representations are processed in the data t2. The HR architecture that can be defined by t2 is depicted in figure 3.7.

3.3 Representation subsystem

3.3.1 DSR-Learner

DSR-Learner automatically develops the DSR representations for all the words used in DYNASTY (see section 3.3.2 for entire DYNASTY vocabularies) except the function words (e.g. at, to, etc.). The function words have random representations since there is currently no way to define these function words using case-role semantics. The DSR-Learner architecture consists of two XRAAMs and was described in chapter 2. The DSR representations for all DYNASTY vocabularies are developed using the propositions which are collected from all the DYNASTY training data (for all modules). As a result, the union of all the propositions in the training data (for all DYNASTY modules) form a proposition space to develop DSRs. These propositions are case analyzed to be loaded into the two XRAAM modules (concept-encoding network and proposition-encoding network). The two different forms of case triples are constructed from these propositions as described in chapter 2. All the propositions involved in learning DSRs for DYNASTY vocabularies are listed in appendix A in two different forms of case triples which can be directly loaded into the two XRAAM modules. The variables in the propositions are randomly replaced with their instances in each presentation during DSR learning. In other words, the DSR-Learner uses randomly *instantiated* forms of propositions as training data. By this random instantiation, DSR-Learner can develop almost identical DSRs for *all the instances in the same conceptual category* (variable). After the DSR-Learner develops all the DSR representations, each word representation is attached with 2-unit ID patterns. The variables are assigned null (00) ID patterns, and all the instances in the same conceptual category are assigned orthogonal ID patterns (to each other). Throughout the thesis, the distributed representations for words designate these ID-plus-DSR representations.

3.3.2 Global dictionary network

The Global-Dictionary (GD) performs the two-way translation from a symbol string to its corresponding distributed representation (ID+DSR), and vice versa. The GD consists of two HF (Hetero-associative Feed-forward) modules: an ASCII-to-DSR module and a DSR-to-ASCII module. The ASCII-to-DSR module selects a corresponding distributed representation for a symbol in the form of a numeric representation of an ASCII string. The DSR-to-ASCII module performs the opposite mapping, that is, from a distributed representation into the numeric representation of an ASCII symbol string. Figure 3.2 shows the GD architecture. These two HF networks in the GD are trained after the correct DSRs have been formed for all words via the recirculation of propositions described in chapter 2. The GD depicted as a box in figures 2.1, 2.3, 2.4 in section 2.5 is actually implemented as two HF modules in DYNASTY after all the DSRs are learned.

Figure 3.3 shows parts of GD training data which consists of ASCII/distributed representation pairs. The ASCII part is a 10 unit continuous representation. Each unit comes from a normalized ASCII code of a single character. For example, the ASCII representation for a word "John" is obtained by normalizing ASCII code for each character J, o, h, n and 6 blank characters into 0 to 1 range, so that "John" is really represented by the vector, e.g. (0.4, 0.6, 0.3, 0.5, 0, 0, 0, 0, 0, 0). The distributed representation part consists of a 2 unit ID part plus a 10 unit DSR part (see section 2.9). The DSR part is obtained from the DSR-learning technique. The ID parts are orthogonal to each other and are chosen arbitrarily for each instance word for the same concept.

In the ASCII-to-DSR module, the input layer is assigned a character string in ASCII

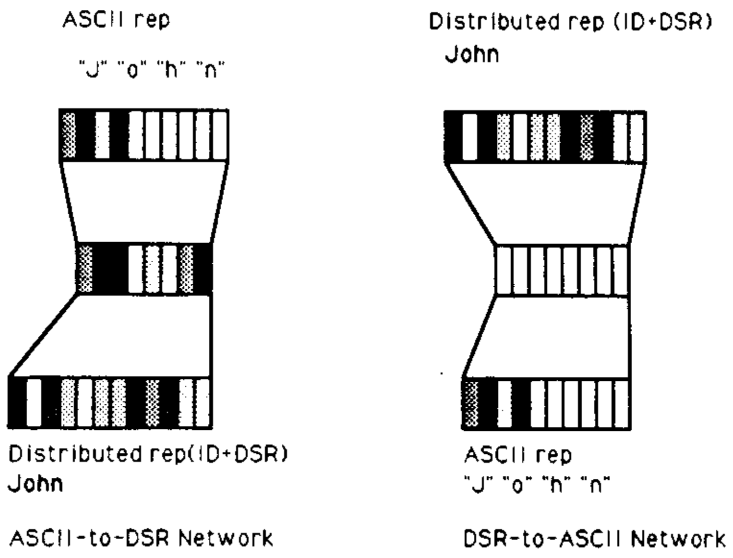


Figure 3.2: **Global-dictionary network architecture.** The architecture employs two HF networks. An ASCII representation is 10 units, representing at maximum a 10 character word, while a distributed representation is 12 units (2 unit ID plus 10 unit DSR).

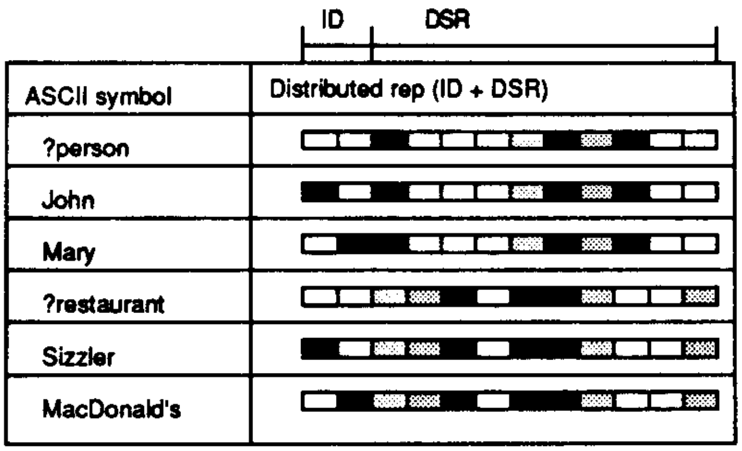

ID
DSR
ASCII symbol
Distributed rep (ID + DSR)
?person
John
Mary
?restaurant
Sizzler
MacDonald's

Figure 3.3: **GD training data examples.** Each ASCII and distributed representation (ID+DSR) pair is used to train the Global-Dictionary network.

?person	?cook-utensil	?raw-food	?guide-book	?restaurant
?food	?market	?class	?professor	?doctor
?hospital				
John	Mary	pan	micro-waveo	fish
chicken	Michelin-guide	Yellow-page	Sizzler	Macdonald's
steak	lobster	Vons	Lucky	computer
biology	Smith	Alan	Dr-Kim	Dr-Park
UCLA-hospital	USC-hospital			
asked	ate	bank	bill	borrowed
bought	brought	called-up	car	cart
cashier	checked	checked-in	coin	comm-link
cooked	d-cont	d-know	d-link	d-prox
drove	entered	examined	exam-room	friend
gasoline	got	got-into	had	home
hospital	hungry	inside	kitchen	knew
left	left-for	line	listened	location
menu	money	near	needed	not
notebook	nurse	ordered	p-health	paid
pay-phone	pb-ask	pb-borrow	pb-cook	pb-doctor
pb-drive	pb-eat	pb-grasp	pb-lecture	pb-letter
pb-phone	pb-read	pb-restaurant	pb-shopping	pb-steal
pb-walk	pb-withdraw	phone-number	picked-up	read
receptionist	s-hunger	sat-down	seat	seated
sick	stole	stopped	tested	time
tip	took-note	took-out	waited	waiter
walked	wanted	went	withdrew	was
from	at	to	book-store	no
for	on	with		

Table 3.1: **Vocabularies in DYNASTY** The three groups of words are respectively variables, instances and non-variables. Every word which starts with "d-" is a goal name, and every word which starts with "pb-" is a plan (or script) name.

representation, while the teaching layer is assigned its corresponding distributed representation. During training, at each step, each word is fed to the network with a corresponding distributed representation and BP training is performed until every association is learned. The DSR-to-ASCII module is trained to learn the inverse association, namely, from a distributed representation to its ASCII string. The program and data format for the GD is listed in appendix E.3.

Table 3.1 shows all the vocabularies used in DYNASTY in 3 groups: variables, instances, and non-variable words. The words with ?-mark are variables. All the words in table 3.1 have 12 unit ID + DSR representations, except the function words which have random 12 unit representations. The DSR representations for the goal/plan names are learned using the goal/plan triple structures in the same manner as the DSR representations for other action/state words are learned using the event triple structures (all the event/goal/plan triple structures to learn DSRs are listed in appendix A). In the GD training, variables are treated equally with their instances. However, for other modules (all the 6 modules in DYNASTY except GD and DSR-Learner), the variables have randomly varying IDs (between 0 and 1) during training. The instances are not used during training but only used during performance phase for the generalizations. All the non-variable words have fixed IDs during training.

3.3.3 Triple-Encoder

Even though the GD provides distributed representations at the word-level, DYNASTY still needs distributed representations for high-level structures (such as event, script and goal/plan knowledge) for each connectionist network module to learn the relations between these high-level structures. The Triple-Encoder module converts event, script, goal and plan triples into distributed vector representations, producing similarity-based representations for each high-level structure from word representations and case-triples. The Triple-Encoder module supports the high-level pattern transformation modules such as Plan-Selector, GP-Associator, and Action-Generator, and parsing and generation modules (see next section for these modules). The Triple-Encoder can encode 4 different kinds of high-level representations: event, script, goal, and plan representations. The encoding knowledge is stored in the weights, but the representations themselves are not pre-stored. Instead, each representation is formed on the fly during the performance phase. During the training of the goal/plan analysis subsystem, the Triple-Encoder is also used to encode the triples to produce the 12-unit representations for event, script, and goals/plans. These 12-unit representations are loaded into each module in the goal/plan analysis subsystem (see section 3.4). An event triple consists of a set of [event-number case-role concept-filler] triples. The event-number is an arbitrary unique number provided in the training data and used to group different triples into one event representation. The concept-filler can be a word concept or another event-number when the triple is embedded within another triple.

Below are shown both simple and embedded event-triple examples. The variables (with ?-mark; e.g. ?person) have randomly varying IDs during training, while the non-variables (e.g. read) have fixed IDs. No instances (e.g. John) are used during training. The specific instances are only used *during the performance phase.* In other words, during training, the Triple-Encoder learns to encode triples with variables (e.g. ?person) with any ID value within a range 0 to 1. During the performance phase, the triples with specific instances (e.g. John) can be encoded since the ID values of specific instances are within the range 0 to 1 and the network knows how to process the specific ID approximately. This procedure of training with random IDs and testing with specific IDs is applied to all the other modules of DYNASTY. As a result, DYNASTY is never trained with specific stories, but only trained with story skeletons with random IDs. Later in the performance phase, DYNASTY can generalize the trained knowledge to the specific stories.

simple event triple: ?person read a ?guide-book.
[ev20 ACT read], [ev20 AGENT ?person], [ev20 OBJECT ?guide-book]

embedded event triple: ?person wanted to read a ?guide-book.
[ev19 STATE wanted], [ev10 AGENT ?person], [ev19 OBJECT ev34]
[ev34 ACT read], [ev34 AGENT ?person], [ev34 OBJECT ?guide-book]

A goal triple consists of a set of [goal-number case-role concept-filler] triples. Likewise, the plan triples consists of [plan-number case-role concept-filler] triples. The goal/plan number is arbitrary and the case-role is extended to contain special case-roles such as GOAL and PLAN in the goal/plan triples.

goal triple: ?person has the goal of taking control of the ?guide-book.
[g23 GOAL d-cont], [g23 AGENT ?person], [g23 OBJECT ?guide-book]

plan triple: ?person has the plan of stealing the ?guide-book from a friend.
[p33 PLAN pb-steal], [p33 AGENT ?person], [p33 OBJECT ?guide-book], [p33 FROM friend]

Script triples are included in the plan triples because a script is a special type of plan.

script triple: ?person has the plan of shopping a ?guide-book at the ?book-store.
[p55 PLAN pb-shopping], [p55 AGENT ?person], [p55 OBJECT ?guide-book], [p55 LOCATION ?book-store]

These kinds of triples provide the training data for the Triple-Encoder module. The Triple-Encoder converts these triples into 12-unit vector representations using an AR (Auto-associative Recurrent) architecture. Figure 3.4 shows the Triple-Encoder module. Some of the event, script, and goal/plan triple examples are listed in the appendix E.2 with the Triple-Encoder program. All the event/goal/plan triples are listed in the appendix A since these triples are the same ones used to learn DSRs in chapter 2, except that the Triple-Encoder does not use instances in the training data. In other words, the Triple-Encoder uses *uninstantiated* forms of triples, while DSR-Learner uses *instantiated* forms of triples as training data.

Basically, this module functions in the same way as the proposition-encoding network in the DSR-Learner module and the training procedure is identical to the one previously described (see the proposition-encoding cycle in section 2.5). But we cannot use the proposition-encoding network in the DSR-learner directly for this purpose, because the representations for the propositions keep getting affected by continuously changing DSRs in the DSR-learner. Here we need to build the triple representations using the *stabilized* DSRs (for every vocabularies in the GD after DSR learning is finished) so that we can decode the triple representations back into the constituent distributed representations[1] for words and case-role representations. One can think that the Triple-Encoder is the same as the proposition-encoding network in section 2.5 using the stabilized ID+DSR representations in the GD, instead of using the continuously evolving DSR representations. Below is the Triple-Encoder training algorithm which is formally described here for clarity. The simple triples that don't have any embedded events as case-fillers are first encoded, and the same procedure is applied to the embedded triples, using previously encoded triple representations as a constituent component (as in figure 3.4). Here the algorithm is described for event triples, but the same algorithm is applied for the script and goal/plan triples.

Triple-encoder training procedure:

1. Load "don't-know" pattern (all 0.5) into bank1.

2. Select the first event triple to be represented, say ev20 (as listed above).

[1]Note that what we mean by distributed representation for word is ID+DSR representation.

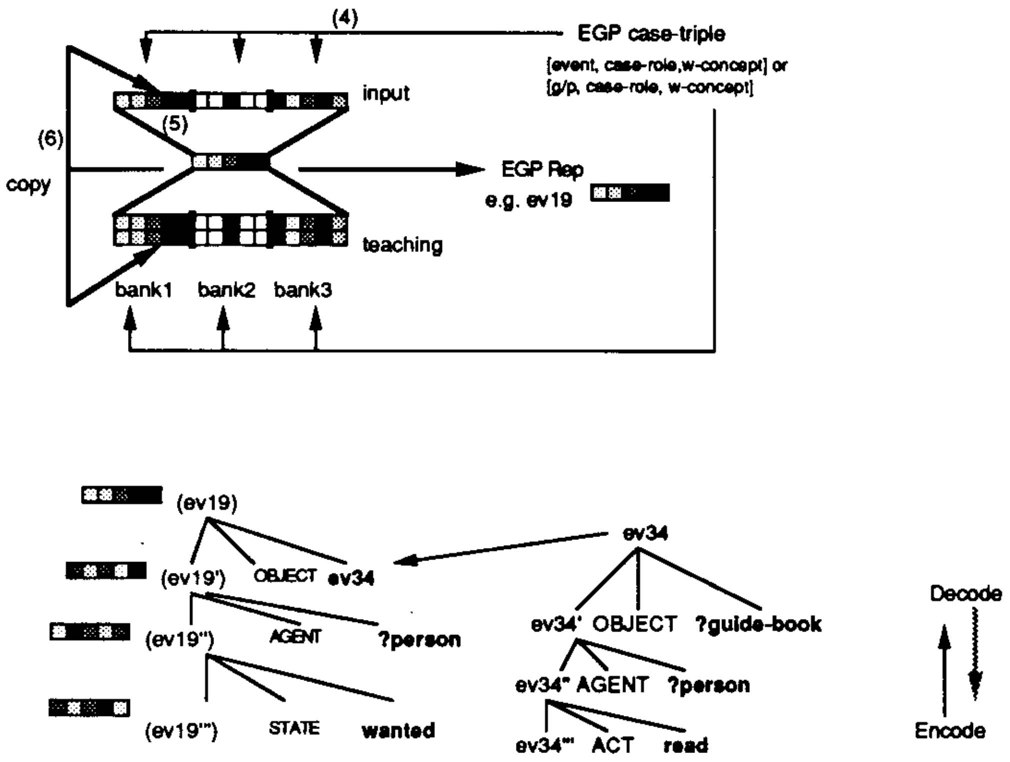


Figure 3.4: **Triple-encoder module.** This example shows the encoding of an embedded event: "?person wanted to read the ?guide-book". The ev34 representation is used as an OBJECT role in ev19. The primes in each triple number (in ev19) indicate that triple representations keep changing until stabilized to reflect the accumulation of encoded case structures. Each bank is drawn to have only 5 units due to the space limitation while it has 12 units (2 unit ID and 10 unit DSR) in the actual experiment.

3. Choose the first triple for the event, ev20, such as [ev20 ACT **read**].

4. Load bank2 and bank3 with the activation pattern of the case-role (e.g. ACT) and, the DSR for the w-concept (e.g. **read**), respectively.

5. Do auto-associative BP, where the input and output layers have the same bit patterns.

6. Recirculate the hidden layer pattern to bank1.

7. Pick up the next triple for ev20 such as [ev20 AGENT **?person**] and repeat step 4 to 7 for all the related triples for this event.

At every epoch[2] during training, the ID values of each variable are set up as random fractions between 0 and 1. These random ID numbers are newly generated for each epoch, but are maintained the same within the epoch so that each variable has consistent ID values. By random, we mean that all the variables have different arbitrary, but consistent IDs during the training. Once the encoder is trained, the encoding and decoding is performed by keeping the weights frozen during the performance phase. The triple decoding algorithm is used to decode the constituent triples from the event/script/goal/plan representations. For example, from the ev20 representation, the decoding procedure extracts the constituent triples [ev20, OBJECT, **?guide-book**], [ev20, AGENT, **?person**] and [ev20, ACT, **read**] in the reverse order of encoding. Below is the triple-decoding algorithm formally described.

Triple-decoding procedure (no training):

1. Load the event representation to the hidden layer: e.g. ev20.

2. Do value propagation (without weight change) from the hidden layer to the output layer.

3. Get the case-role representation from bank2 (e.g. OBJECT) and the DSR for the concept (e.g. **?guide-book**) from bank3.

4. Copy the intermediate representation for the event (e.g. ev20') in bank1 (in the output layer) to the hidden layer.

5. Repeat steps 2 to 4 until we have extracted all the originally encoded triples.

The tree in the figure 3.4 shows the encoding/decoding sequence which consists of stack pushing/popping operations. Encoding is performed bottom-up from the tree leaves to the root, but decoding is performed top-down from the tree top to the leaves.

[2]One epoch is a period for presenting every training data.

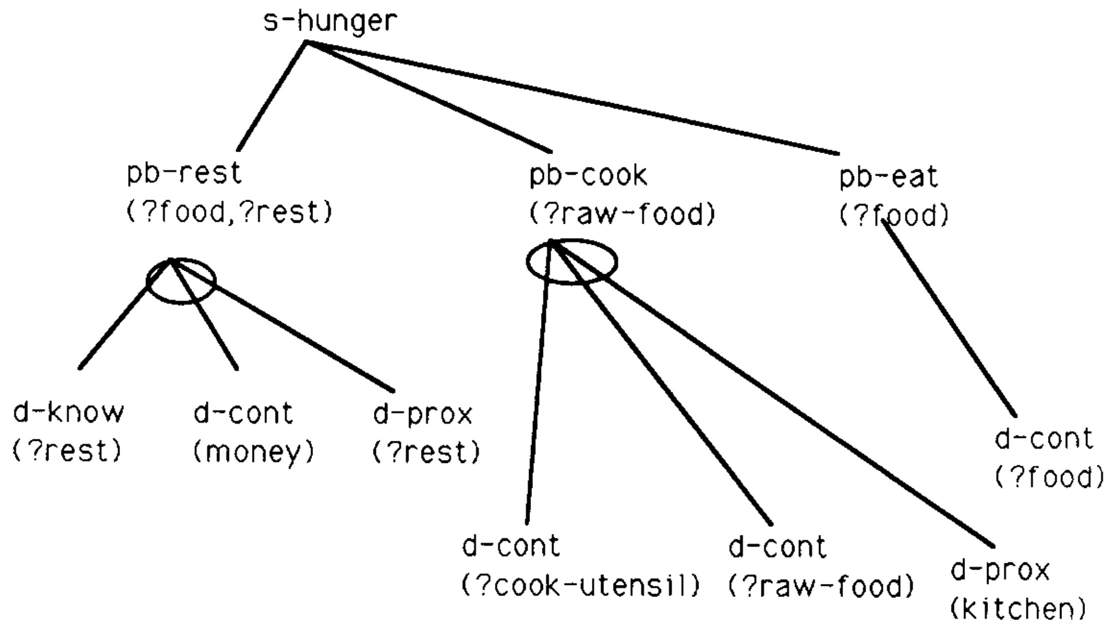


Figure 3.5: **Single s-hunger goal/plan tree.** Words starting with ?-mark indicate variables. The circle designates an AND node in which every child node must be satisfied to satisfy the node. Each goal/plan structure is defined in appendix C.1. In this figure, the AGENT role ?person (planner) is omitted for every goal/plan structure.

3.4 Goal/plan analysis subsystem

3.4.1 GP-Associator

DYNASTY's goal/plan knowledge is represented using a set of gp(goal/plan)-trees. A single gp-tree is an OR/AND tree where an OR node represents a goal and an AND node represents a plan including script. Figure 3.5 shows part of DYNASTY goal/plan-trees. The whole DYNASTY gp-trees are depicted in appendix C.2.

The link from the goal node to the plan node represents the "plan-for" relation. For example, "s-hunger"[3] goal has a plan "restaurant script" OR "cooking plan" OR "direct eating plan". The link from a plan node to a goal node represents the "sub-goal" relation. So the "restaurant script" has a precondition "d-know (restaurant location)" AND "d-cont (money)" AND "d-prox (restaurant)". These preconditions are instrumental goals that should be satisfied before the plan can be executed. The goal node is an OR node because if we execute just one plan among the many possible plans then the goal is satisfied. However, the plan node is an AND node because we need to satisfy all of the precondition goals in order to execute the plan. Each single gp-tree consists of this depth-two OR-AND tree. DYNASTY represents goal/plan knowledge using a set of these OR-AND trees, so DYNASTY's goal/plan

[3]The goal/plan names follow the conventions in the symbolic goal/plan theory [Schank and Abelson, 1977]. All the goal/plan names used in DYNASTY are defined in appendix C.1.

space consists of forests of OR-AND trees (see appendix C.2 for the full DYNASTY gp-trees).

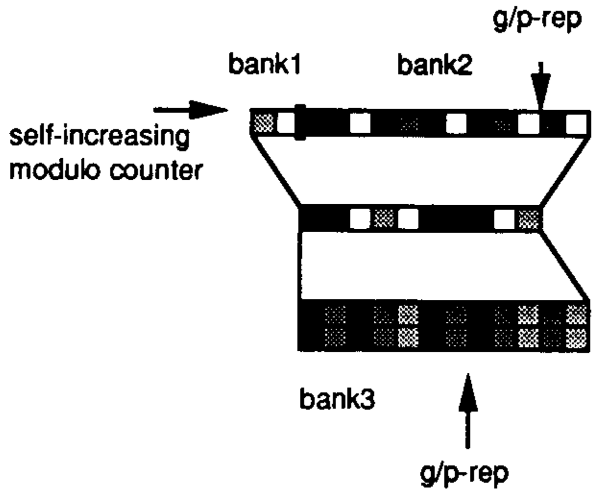
The GP(goal/plan)-associator's function is to encode the gp-tree structures, thereby encoding all of the goal/plan relations. When a goal is given, the GP-Associator can produce all of the possible alternate plans for the goal. Similarly, when a plan is given, the GP-Associator can produce all of the necessary preconditions to execute the plan. All of these preconditions are instrumental goals to the plan and connected to other gp-trees.

Figure 3.6 shows the GP-Associator architecture, which is a HF module with a self increasing counter bank. The input layer has two banks: a 2-unit counter bank and a 12-unit gp-bank. The counter bank is used to distinguish the several child nodes in the gp-tree. When the counter is n, the goal node is associated with the n-th plan available for the goal, and the plan node is associated with the n-th precondition. The 2-unit counter bank can represent maximum 4 different plans or preconditions. Gp-bank holds the 12-unit goal/plan representations obtained from the Triple-Encoder. The output layer has one gp-bank which holds the associated goal/plan representations (with the goal/plan in the input layer). For example, consider encodings of the gp-tree in figure 3.5. First the plan-for relation between s-hunger (OR node) and several possible plans (pb-restaurant, pb-cook, pb-eat) is to be encoded (see figure 3.6 for specific data loading into the bank). When the counter 0 (binary 00) is loaded in the bank1, and the s-hunger goal representation is loaded in the bank2, the training output pb-restaurant plan representation is loaded in the bank3 to be associated as a first plan for s-hunger goal. When the counter is increased to 1 (binary 01), the pb-cook plan representation is loaded in the bank3 as a second plan, and so forth while the bank2 has the same s-hunger representation. Next, the sub-goal relation between pb-restaurant plan (AND node) and several preconditions (d-know, d-cont, d-prox) is to be encoded. The same procedure is repeated. When the pb-restaurant representation is loaded in the bank2 and counter is 0, the d-know representation is loaded in the bank3 as a first precondition. With the counter 1, the d-cont representation is loaded as a second precondition, and so on (figure 3.6).

The training data are specified in DYNASTY specification language in section 3.2. The triples in the training data are converted into the 12 unit goal/plan representation using the Triple-Encoder. In other words, during GP-Associator training, the Triple-Encoder module is called to do the necessary triple to representation encoding on the fly. Below is the training procedure of GP-Associator.

Training procedure:

1. Pick a gp-tree to be encoded.
2. Load the root node (goal representation) (e.g. s-hunger) to the bank2.
3. Initialize bank1 with the initial counter value (00).
4. Load its first child node (plan or goal representation) (e.g. pb-restaurant) into bank3.
5. Do hetero-associative BP.
6. Increase the counter value by one and load the next child node (plan or goal representation) (e.g. pb-cook) into bank 3.



input		training-output
bank1	bank2	bank3
00	s-hunger	pb-restaurant
01	s-hunger	pb-cook
10	s-hunger	pb-eat
00	pb-restaurant	d-know
01	pb-restaurant	d-cont
11	pb-restaurant	d-prox

Figure 3.6: **Goal/Plan-Associator module.** Bank1 acts as a self-increasing modulo counter, which is initially loaded with the binary representation of 0. Each g/p representation in bank2 and bank3 is obtained from the Triple-Encoder during training. See text for loading each representation into the bank.

7. Repeat steps 5 and 6 until all the children nodes are associated.

8. Load the first child node (plan representation) (e.g. pb-restaurant) into the bank2 and repeat steps 3 to steps 7 until all of its children nodes are associated.

9. Pick another gp-tree to be encoded and repeat the whole procedure.

For example, to encode the gp-tree in figure 3.5, the goal representation of s-hunger:

[g1 GOAL s-hunger], [g1 AGENT ?person]

is first associated with the first plan available (pb-restaurant) when the counter is 0, such as:

[p1 PLAN pb-restaurant], [p1 AGENT ?person], [p1 OBJECT ?food], [p1 LOCATION ?restaurant]

When the counter increased to 1, the same goal representation is associated with the second plan in the tree such as:

[p2 PLAN pb-cook], [p2 AGENT ?person], [p2 OBJECT ?raw-food], [p2 LOCATION kitchen]

This training continues until all possible solution plans are associated with the s-hunger goal. So each goal is associated with all the different possible plans according to the gp-tree with different counter values. Similarly, each plan is associated with all the different preconditions (instrumental goals) with different counter values. In the training, the IDs of each variable (?-marked word) are set up as random across all the training data, but are maintained same within an epoch so that consistent IDs can be propagated through the same instance word[4]. These random settings of IDs are performed only when the same variables appear both in the input layer and in the output layer to be propagated (see example below). The variables in DYNASTY are different from the content-less variables (such as ?x, ?y) in symbolic systems. ***In DYNASTY, variables have the function of restricting the possible bindings, so they are more similar to conceptual categories than pure variables.*** All variables in DYNASTY have semantic representations using DSRs. Content-less variables cannot have meaningful distributed representations in a connectionist system [Dolan, 1989].

The GP-Associator program and data format is listed in appendix E.5. The above training example can be represented using DYNASTY's data-specification language as follows:

IF 0
AND [g1 GOAL s-hunger], [g1 AGENT ?person]
THEN [p1 PLAN pb-restaurant], [p1 AGENT ?person], [p1 OBJECT ?food], [p1 LOCATION ?restaurant]

[4]Note the 3 groups of vocabularies in the DYNASTY's GD, that is, variables, instances and non-variables (section 3.3.2).

IF 1
AND [g1 GOAL s-hunger], [g1 AGENT ?person]
THEN [p2 PLAN pb-cook], [p2 AGENT ?person], [p2 OBJECT ?food], [p2 LOCATION kitchen]

In this training data, the ID for the variable ?person is set up randomly during training, but the IDs for the variables ?food and ?restaurant are fixed (as 00). When the variables in the output layer (THEN part) do not have the same variables in the input layer (IF part), there is no need to propagate the correct IDs for variable binding. Random setting of these particular variables would be a waste of computational resources. When the IDs for the variables which *only appear in the output layer* are setup as random, BP algorithm cannot converge since it is computing one-to-many mappings in these particular cases. The same input pattern must be mapped to randomly varying output patterns.

3.4.2 Plan-Selector

The Plan-Selector's function is to select a single goal or plan from specific events or event sequences. This module performs mapping functions from the event (state and action) space into the goal/plan space. These mappings vary from simple association to recurrent mappings according to the plan structures. The semantics of this mapping is that when DYNASTY sees certain actions or states, DYNASTY infers that the planner has a certain goal or that the planner is trying to use a certain plan.

Figure 3.7 shows the Plan-Selector architecture. The network has 2 banks in the input layer and 2 banks in the output layer. The first bank in the input layer consists of contexts copied from the hidden layer. The second bank will be loaded with an event representation. The first bank in the output layer will hold the goal or plan representation selected, and the second bank holds the flag telling whether the goal or plan has been accomplished or not. This flag guides the search process in the goal/plan trees (see section 4.3.2).

In the training phase, the inputs are event representations, and the teaching inputs are the selected goal/plan representations. The training algorithm follows the steps of a HR architecture training.

Training algorithm:

1. Load bank1 with initial "don't-know" pattern.

2. Choose one event (sequence) to be a goal or plan and load bank2 with the first event representation.

3. Load bank3 with the goal or plan representation to be selected for the chosen event (sequence).

4. Load bank4 with the indicator that the goal or plan is a success or not.

5. Do hetero-associative BP.

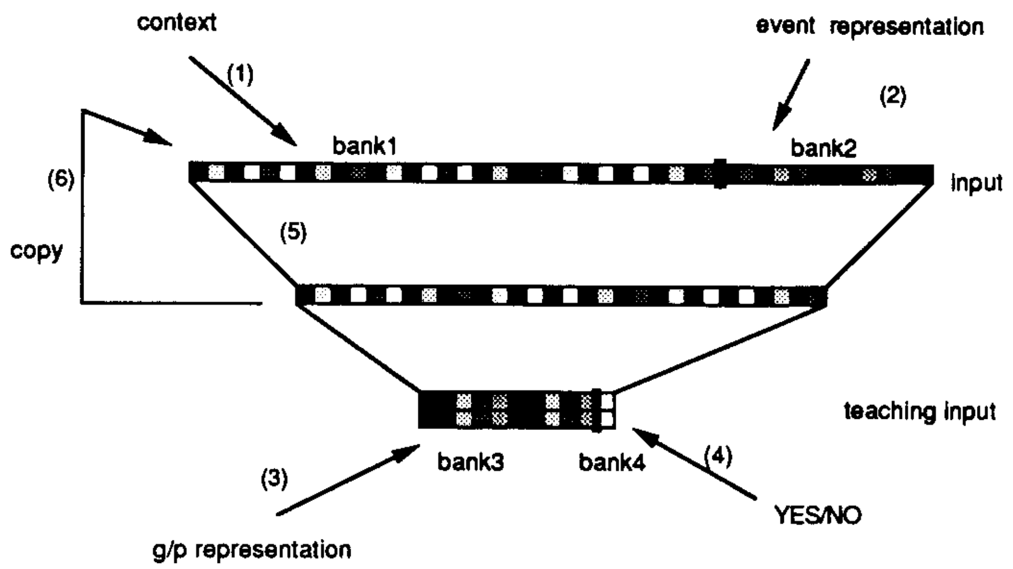


Figure 3.7: **Plan-selector module.**The event and goal/plan representations come from the Triple-Encoder. The yes/no bank contains a single unit flag designating whether the goal/plan has been accomplished or not. Value 1 is YES, and value 0 is NO flag.

6. Recirculate the hidden layer pattern to bank1.
7. Load bank2 with the next event representation in the sequence.
8. Repeat steps 5 to 7 for all of the events in the chosen sequence.

The loaded training patterns are event and goal/plan representations obtained using the Triple-Encoder module. The training data represents the associations between the events (or event sequences) and goals/plans (in the DYNASTY gp-tree). The IDs of variables are set up randomly during training. All possible events that can be associated with the goal or plan in the gp-tree can be listed in the training data. For example, for the "s-hunger goal" in the tree (figure 3.5), the following training data consists of event to goal/plan mapping.

IF [ev30 STATE hungry], [ev30 AGENT ?person]
THEN [g1 GOAL s-hunger], [g1 AGENT ?person]
AND [NO]

The event and goal/plan numbers are just for reference and do not affect the processing. From the training data, the Triple-Encoder produces the event and goal/plan representations and feeds them to the Plan-Selector network. The [NO] flag means this s-hunger goal is not yet accomplished given the input hungry event. The above training data can be interpreted as follows (in plain English):

IF person is hungry
THEN person has a goal not to be hungry
AND the goal is not yet satisfied

The whole purpose of plan selection is to find the most feasible goals and plans of the planner when DYNASTY sees a given event (action/state) or event sequence. Those goals and plans should reside in the goal/plan trees that DYNASTY is trained on (see appendix C.2 for the goal/plan trees in DYNASTY). The most feasible goal/plan designates the one that does not need further goal/plan inferencing. For example, the event "John borrowed a pan from a friend" can be mapped into the two different levels of plans: (1) John is using the borrowing-plan to take control of a pan, or (2) John is using the cooking-plan. The Plan-Selector will choose (1) for this event since the given event directly mentions this plan, so it is the first one to successfully match. Inferring (2) from this event depends upon the context and should be done by dynamically searching the goal/plan trees. This search is done by the GP-Associator module (section 3.4.1). When the event does not directly mention the plan used, the Plan-Selector chooses a goal behind the event. For example, the event "John entered a kitchen" is mapped into the goal "John has a goal of getting inside kitchen" since DYNASTY does not know the plan of entering a kitchen (e.g walking-plan or running-plan) yet. When the event mentions the state of a planner (and not an action), then the Plan-Selector maps the event into a goal (such as s-hunger goal described in the above training data example). Plan selection in DYNASTY is different from the general plan-recognition process [Schmidt et al., 1978] in that DYNASTY infers only the most feasible goal (or plan) and leaves the top-level goal recognition problem to the GP-Associator module.

The above training data are examples of simple association in which the context bank do not play a role. If the plan structure is large and contains several steps of events such as a "pb-restaurant" plan[5], then the mapping needs a context bank. For example, if the plan is a complex script such as restaurant-going, the training data can be read as follows (in plain English):

IF person entered restaurant
FOLLOWS waiter seated person
FOLLOWS waiter brought menu
FOLLOWS person read menu
FOLLOWS person ordered food
FOLLOWS person ate food
FOLLOWS person paid bill
FOLLOWS person left a tip
FOLLOWS person left a restaurant for home
THEN person had a plan of executing restaurant script
AND the plan is accomplished

In the above training data, each event representation is fed to the network and the restaurant plan representation is gradually formed when the network recirculates the hidden

[5]This kind of long complex plan is an example of a script.

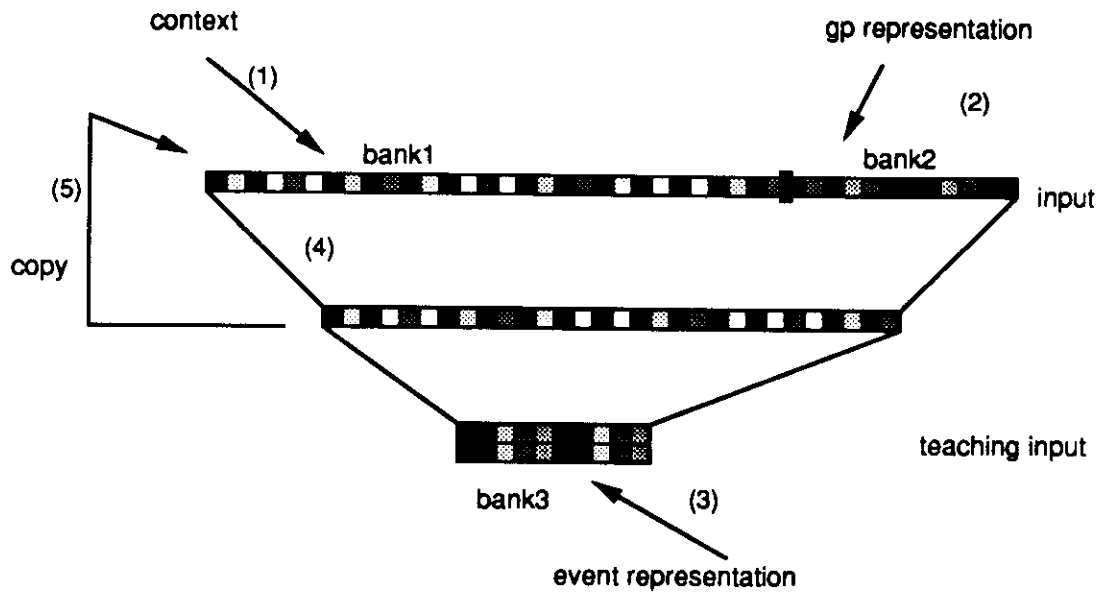


Figure 3.8: **Action-Generator module.**

layer representation to the context layer [Lee et al., 1990; Miikkulainen and Dyer, 1989]. During the performance phase, any combination of input events can select a restaurant plan representation. For example, from the partial input event sequence such as "John entered the Chart-House. John ate steak. John left a tip", the Plan-Selector produces the restaurant plan representation with the correct bindings during the performance phase (see section 4.6). The program and data format for the Plan-Selector is listed in appendix E.4.

3.4.3 Action-Generator

The Action-Generator's function is to produce the event (action or state) representations when a goal/plan representation is given as an input. The output event designates the event or event sequences when the goal or plan has been already accomplished. For example, when the s-hunger goal is accomplished, the Action-Generator can produce the state that the planner is not hungry anymore. The module performs the mappings from goal/plan space to event space which is the reverse of the Plan-Selector's mappings. The semantics of this mapping is that when the planner has a certain goal or plan and if the goal or plan has already accomplished, then the planner should be in a certain state or the planner should have done a certain sequence of actions. The network has to find the regularities between event and goal/plan representations in order to retrieve the state or action representations from the goal or plan representations.

Figure 3.8 shows the architecture, which uses the same HR (Hetero-associative Recurrent) architecture as used in the Plan-Selector module, but with different inputs and outputs. The input layer has two banks. The first bank holds the context representation that is copied from the hidden layer and the second bank holds the goal/plan representations. The output layer has one bank that holds the event representations generated.

In the training data, the inputs are goal/plan representations and the training outputs are the associated event or event sequences. The training procedure is similar to the Plan-Selector module which is a HR module training.

Training procedure:

1. Load bank1 with the "don't-know" pattern.
2. Pick-up a goal/plan representation and load it into bank2.
3. Load bank3 with the first event representation for the chosen goal/plan.
4. Do hetero-associative BP.
5. Recirculate the hidden layer pattern to bank1.
6. Load bank3 with the next event representation in the goal/plan.
7. Repeat steps 4 to 6 for all the events associated in the chosen goal/plan.

For example, the following verbatim training data consists of goal/plan to event mapping examples. The IDs of each variable varies randomly during training.

(td1)
IF [g1 GOAL s-hunger], [g1 AGENT ?person]
THEN [ev101 STATE hungry], [ev101 AGENT ?person], [ev101 MODE not]

(td2)
IF [p2 PLAN pb-cook], [p2 AGENT ?person], [p2 OBJECT ?raw-food], [p2 LOCATION kitchen]
THEN [ev110 ACT cooked], [ev110 AGENT ?person], [ev110 OBJECT ?raw-food], [ev110 LOCATION kitchen]

These training data can be interpreted in plain English as follows:

(td1)
IF person accomplishes a goal of not to be hungry
THEN person is not hungry anymore

(td2)
IF person executed a cooking plan
THEN person cooked a raw-food in the kitchen

The triples are converted into 12 unit event/goal/plan representations to be loaded into the banks of Action-Generator. The IDs for variables are set up randomly within a epoch. The non-variable words have fixed IDs during training.

When the plan is a complicated script such as a pb-restaurant, then the event sequence associated with the script is produced [Lee et al., 1990]. For example, the following interpretation of training data explains script generation using the Action-Generator.

IF person executed a plan of restaurant script
THEN person entered restaurant
FOLLOWS waiter seated person
FOLLOWS waiter brought menu
FOLLOWS person read menu
FOLLOWS person ordered food
FOLLOWS person ate food
FOLLOWS person paid bill
FOLLOWS person left tip
FOLLOWS person left restaurant for home

From the pb-restaurant representation, the Action-Generator produces this full event sequence during the performance phase (section 4.6). The program and data format for the Action-Generator is listed in appendix E.6.

3.5 Linguistic subsystem

3.5.1 Sentence to triple parser

The Sentence-to-Triple (ST) parser converts input natural language sentences into event triples. DYNASTY ST-Parser was modeled after case-role assignment networks in [McClelland and Kawamoto, 1986]. The sequential inputting of words was modeled after the DISCERN sentence-parser [Miikkulainen, 1990a]. Figure 3.9 shows the architecture, which is a HR network.

The input to the network is a distributed representation of each word in the sentence, word by word. The training output is a case-role assignment representations of the input sentence. The teaching input has 12 banks that correspond to each case-role in DYNASTY. The hidden layer representation is recirculated into the context bank in the input layer. Below we describe a training algorithm for the ST-Parser.

Training ST-Parser

1. Pick up one input sentence, and load the first word's distributed representation into the bank2 (input).
2. Load don't know pattern (all 0.5s) into bankl (context).
3. Load the sentences' case-role assignment into the teaching input banks. If there is no word for a specific case-role, load an all-zero pattern for that case-role.
4. Do hetero-associative BP.
5. Recirculate the hidden layer into bankl.
6. Load the next word's distributed representation (in the input sentence) into the bank2.
7. Repeat steps 4 to 6 for all the words in the sentence.

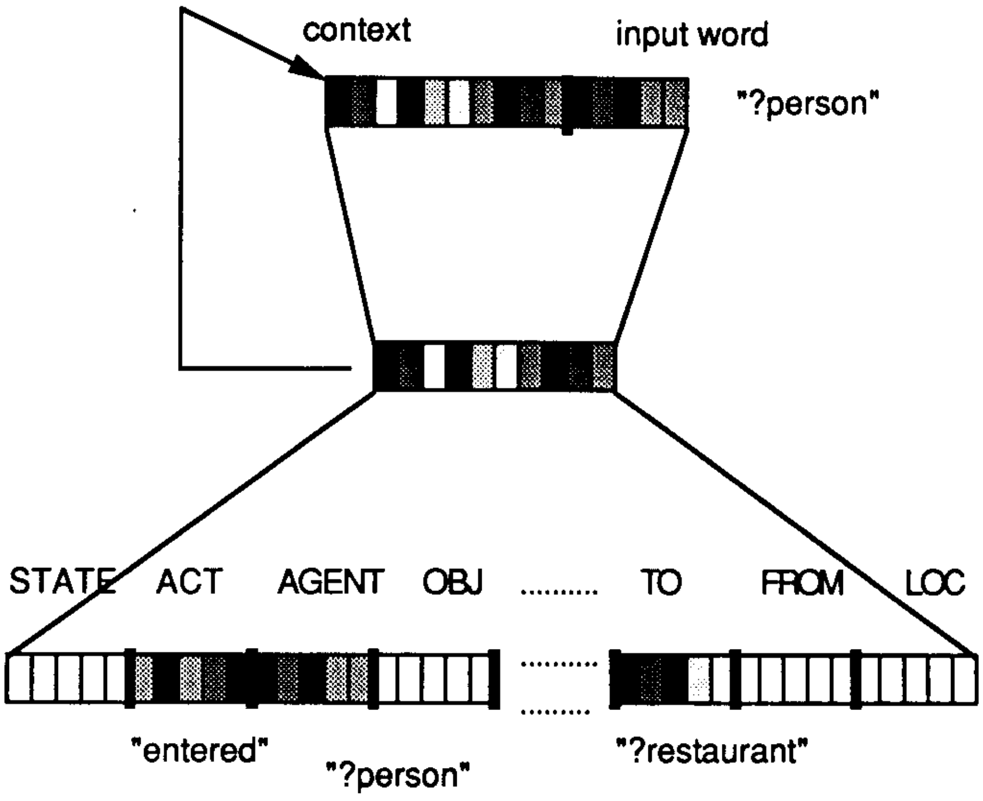


Figure 3.9: **ST-Parser network architecture.** The input layer has two banks; bankl for context and bank2 for each word representation. The output layer has 12 banks for each case-role assignment used in DYNASTY: STATE, ACT, AGENT, OBJECT, CO-OBJ, OBJ-ATTR, INSTRU, FROM, TO, LOC, TIME, MODE. Each word in one sentence is sequentially fed to the input word bank. Blank banks in the output layer designate that the null pattern (all zeros) are assigned for those banks. The number of units in each bank is arbitrarily drawn in this figure due to space limitations. In the actual network, the context bank has 144 units and the other banks have 12 units.

For example, during training, for the input sentence "?person entered ?restaurant", the teaching input has the representation for word **?person** in the AGENT role, **entered** in the ACT, and **?restaurant** in the TO case-role bank. For the input layer, the representation for **?person, entered, ?restaurant** are fed to the bank 2 sequentially, one word at a time. The parser generates arbitrary event numbers to form event triples.

ST-Parser can handle embedded sentences and produce an event triple for each embedded event. For example, consider the following sentence: "?person wanted to know the ?market location". Via pre-processing, the sentence is slightly modified into: ?person wanted (?person knew ?market location) to feed to the ST-Parser. The sentence inside parenthesis is processed first. The training data for this embedded sentence is as follows:

```
IF ?person
FOLLOWS knew
FOLLOWS ?market
FOLLOWS location
THEN knew AND - AND ?person AND location AND - AND ?market AND -
AND - AND - AND - AND - AND -

IF ?person
FOLLOWS wanted
FOLLOWS ev30
THEN wanted AND - AND ?person AND ev30 AND - AND - AND - AND -
AND - AND - AND - AND - AND -
```

The output layer after THEN has 12 banks for the case-role STATE, ACT, AGENT, OBJECT, CO-OBJECT, OBJ-ATTR, INSTRU, FROM, TO, LOCATION, TIME, MODE in this sequence. The "-" designates that the null (all zero) pattern is to be loaded. The ev30 designates the representation of the inner event: ?person knew ?market location. After the inner event is parsed, the resulting event triples are converted into 12 unit event representations using the Triple-Encoder. The converted event representation (ev30, here) is fed to the outer event parsing data. The parsing output is as follows:

```
[ev31 STATE wanted] [ev31 AGENT ?person] [ev31 OBJECT ev30]
[ev30 STATE knew] [ev30 AGENT ?person] [ev30 OBJECT location] [ev30 OBJ-
ATTR ?market]
```

The program and training data for the ST-Parser are listed in appendix E.7.

3.5.2 Triple to sentence generator

The TS-Generator converts event triple forms into an output sentence, which is a reverse process of ST-Parsing. The architecture (shown in Figure 3.10) is similar to the ST-Parser's except that, this time, the 12 case-role banks form the input layer and there is one output bank for the generated word.

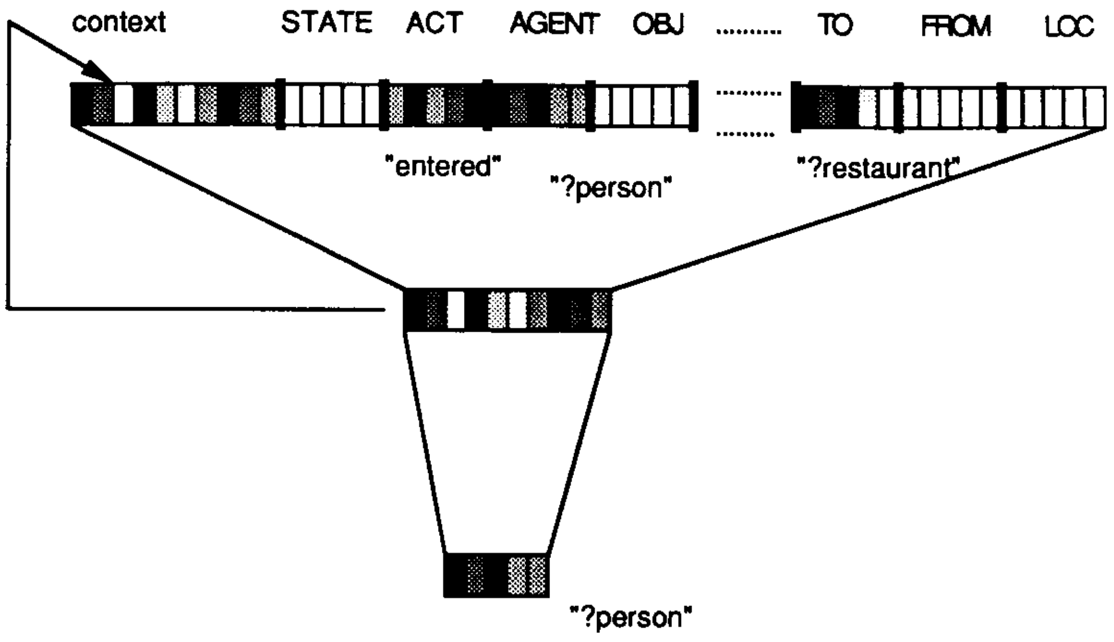


Figure 3.10: **TS-Generator network architecture.** Again, the number of units in each bank is arbitrarily drawn here. The actual number of units are 144 (12 × 12) units for the context bank, and 12 units for the other banks.

The training algorithm is similar to that of ST-Parser. The input layer has 13 banks (one for the context, and others for the 12 case-roles), and the teaching input layer has one bank for the output word representations. For example, the event triples ([evl ACT **entered**], [evl AGENT **?person**], [evl TO **?restaurant**]) are assigned to the input layer such as: **entered** goes to the ACT, **?person** goes to the AGENT, and **?restaurant** goes to the TO case-role bank (using symbolic pattern copy operations). The output representation for the word **?person**, **entered**, and **?restaurant** are produced sequentially, word by word, in the output layer. During generation (performance phase), the network produces some canned words to help the user interpret output inference chains. For example, from the event: [ev30 STATE hungry], [ev30 AGENT John][6], the TS-Generator produces the sentence "John *was* hungry", not "John hungry". Some of the functional words such as "at", "to", etc are also produced. In other words, some of the functional words are also part of the training data. The code for TS-Generator and training data are listed in appendix E.8.

The GD (Global-Dictionary), ST-Parser and TS-Generator provide linguistic front-ends for the DYNASTY system and are the only language-dependent parts in the system. The GD connects the symbolic world to the connectionist world by providing a means for accessing connectionist representations from symbolic ASCII strings and vice versa. By means of the GD, symbolic input can be processed in the connectionist network, and connectionist output can be shown as symbolic forms to the users. The ST-Parser and TS-Generator map sentences to/from event triples, on which DYNASTY's reasoning subsystems for the

[6]During performance phase, specific instances (e.g. John) are used instead of variables (e.g. ?person) to make up specific story instances.

script/goal/plan analysis operate. The ST-Parser and TS-Generator are not a full-fledged natural language parser and generator, but just translate between natural language inputs/outputs and event triples. They do not deal with any complicated syntax and are designed only to handle simple word orderings. Nor do they deal with any kind of discourse analysis in natural language generation either.

Chapter 4

Goal/plan processing in DYNASTY

4.1 Introduction

This chapter describes how DYNASTY integrates several modules to process script and goal/ plan-based stories. DYNASTY uses symbolic algorithmic controls to connect each connectionist module sequentially in order to generate inferences behind the narratives. Figure 4.1 shows DYNASTY's top-level performance configuration. The DYNASTY performance program and data format is listed in appendix F.

The following steps show how DYNASTY processes script/goal/plan-based stories. In the performance phase, DYNASTY uses a symbolic working memory to store the goal/plan representations previously inferred by the system during the story processing. DYNASTY stores a top-level goal into this memory as a default goal so that the system can assume that this goal is already inferred. This top-level goal gives the necessary bootstrap to the system, as in [Wilensky, 1978].

Script/goal/plan-based story processing step (for each sentence):

1. Access distributed representation (ID+DSR) for each symbol in the input sentence using the Global-Dictionary.
2. Get event-triples using ST-Parser for the input sentence.
3. Build event representations using the Triple-Encoder.
4. Select a proper goal/plan for the input event using the Plan-Selector.
5. Store selected goals/plans into the working memory.
6. Expand gp-tree in a breadth-first way using the GP-Associator to generate candidate goals or plans until the selected goal or plan is generated.
7. Build up a goal/plan inference chain by following in reverse the expanded gp-tree in the working memory.
8. Generate actions or states from the goal/plan in the chain using the Action-Generator.
9. Decode the generated action/state representations into their triples using the Triple-Encoder.
10. Generate output paraphrase from the triples using the TS-Generator.

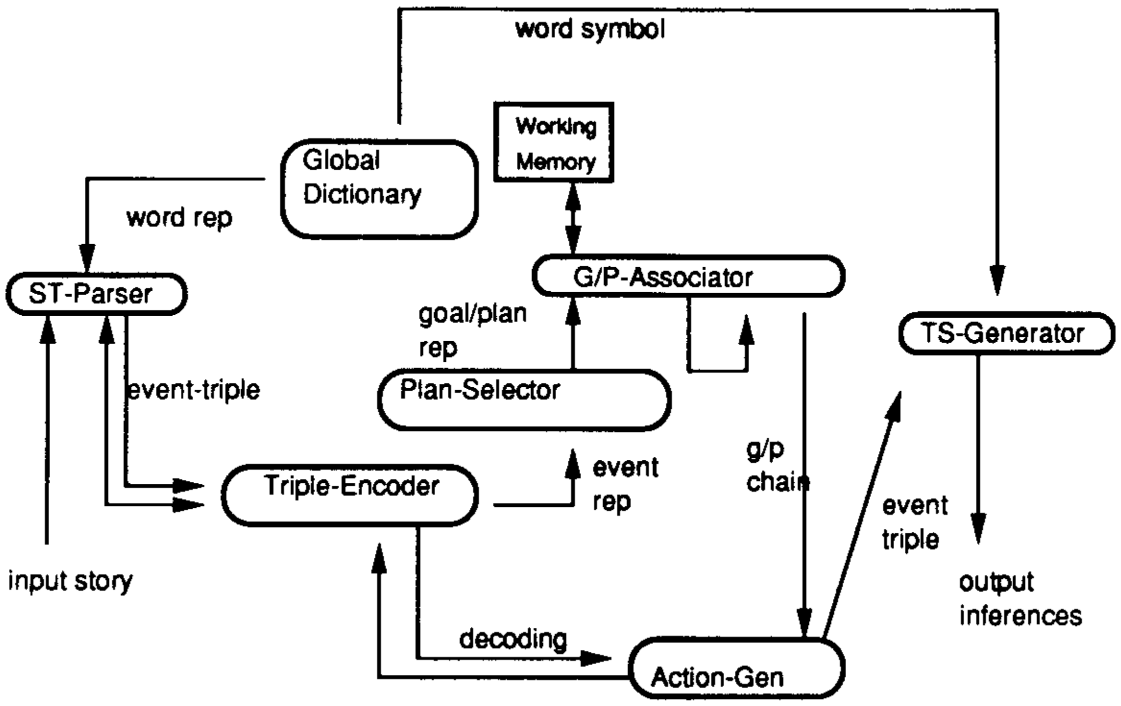


Figure 4.1: **Top-level architecture in performance phase.** The entire processing architecture is divided into 3 parts: surface processing (ST-Parser, Global-Dictionary, Triple-Encoder), goal/plan analysis (Plan-Selector, GP-Associator, Action-Generator) and surface generation (TS-Generator, Triple-Encoder, Global-Dictionary). The Triple-Encoder during surface input processing is used in encoding mode, while the Triple-Encoder during surface generation is used in decoding mode. The arrows designate data flow between each module.

11. Look-up in the Global-Dictionary and change the distributed representations for each word into its corresponding symbolic form.

Goal/plan processing consists of three big steps: surface processing, inferencing through goal/plan analysis, and surface generation. At the top level, the entire goal/plan processing resembles that of symbolic systems [Wilensky, 1978]. The processing starts with the story parsing into the internal representations that can be used by the connectionist modules (surface processing). Plan-recognition, goal/plan searching, and inference generation processes follow in order to generate the inference chains (goal/plan analysis). Finally, the internal representations of the inference chains are realized into surface language forms (surface generation).

4.2 Surface processing

Surface processing consists of step 1 to step 3 in the top-level algorithm. Suppose the input sentence reads:

John wanted (John called-up friend).

The articles, pronouns and other syntactic elements are ignored in the actual natural language input. The most important syntactic component considered in ST-parsing is the word order. According to the word order, the ST-Parser network assigns proper case-roles for each word, and builds up the case-role triples. When the sentences have embedded structure such as the example here, we group the inside constituents using parentheses. The ST-Parser scans the story and first processes the sentence inside the parentheses. Before parsing, the ST-Parser fetches the distributed representations (ID+DSR) for each symbolic word in the sentence from the GD and feeds those representations into the ST-Parser network sequentially, word by word. Then the ST-Parser builds the event triples for each sentence. For example, for the above sentence, ST-Parser builds up the following event triples:

[ev30 ACT called-up] [ev30 AGENT John] [ev30 TO friend]
[ev31 STATE wanted] [ev31 AGENT John] [ev31 OBJECT ev30]

The event numbers are arbitrarily assigned from DYNASTY's internal number counter. Other non-embedded events are parsed in the same manner.

The final step in surface processing is performed by the Triple-Encoder. The Triple-Encoder fetches each triple and encodes the triple structures into 12-unit distributed representations for each event. The final 12 unit representation (e.g. for the ev31) is passed to the goal/plan analysis process as the internal representation of the sentence in the story.

4.3 Goal/plan analysis

Goal/plan analysis corresponds to step 4 to step 8 in the top-level algorithm. The goal/plan analysis processes are performed by chains of 3 network components: Plan-Selector, GP-Associator, and Action-Generator. From the input event representations, the

Plan-Selector selects a goal or plan on the gp-tree and GP-Associator starts expanding the gp-tree to create the goal/plan inference chain. The Action-Generator produces events (or event sequences) from the goals/plans in the inference chain.

Suppose the input story reads (story 5):

> (story 5) John was hungry. John read Michelin-guide. John borrowed money from friend.

The surface-processing steps produce the following event representations from this input story. Actually, each event triple was already encoded into 12 unit vector representations using the Triple-Encoder at this point. The original triples are listed here for reference:

> [ev31 STATE hungry] [ev31 AGENT John]
> [ev33 ACT read] [ev33 AGENT John] [ev33 OBJECT michelin-guide]
> [ev39 ACT borrowed] [ev39 AGENT John] [ev39 OBJECT money] [ev39 FROM friend]

4.3.1 Plan selecting process

The Plan-Selector searches the goals or plans for these events in the gp-tree and changes the event representations into the goal/plan representations with accomplishment flags (YES/NO). The selected goals/plans for the input events are as follows:

> [g32 GOAL s-hunger] [g32 AGENT John] [NO]
> [p34 PLAN pb-read] [p34 AGENT John] [p34 OBJECT michelin-guide] [YES]
> [p40 PLAN pb-borrow] [p40 AGENT John] [p40 OBJECT money] [p40 FROM friend] [YES]

The accomplishment flags designate whether the goal/plan was accomplished or not, and are used later for searching the gp-tree.

4.3.2 Expanding the goal/plan tree

From the goal/plan representations with accomplishment flags, the GP-Associator expands the gp-tree in a breadth-first manner and builds up the inference chain. Figure 4.2 shows the expanded gp-tree from the s-hunger (g32) goal to pb-read (p34) plan.

From the starting node (g32), the GP-Associator generates every possible plan for the s-hunger goal, and for each plan, it generates every precondition, until it matches the target node (p34). Once the GP-Associator matches the target node, it builds-up the goal/plan inference chain by following the reverse pointer (thick black lines in figure 4.2) in the symbolic working memory. In other words, the working memory consists of symbolic linked lists in which each node has a pointer to the parent node. Traversing through the parent pointer produces a single chain since each node has only one parent in the gp-tree. (The matching process will be discussed in section 4.3.4.)

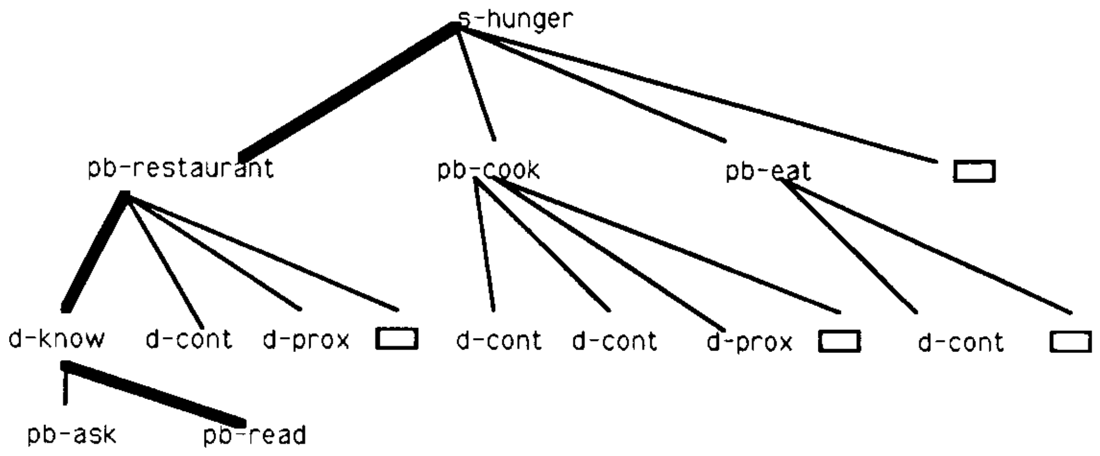


Figure 4.2: **First gp-tree expansion.** Breadth-first expansion of goal/plan tree from s-hunger goal to pb-read plan. This expansion is represented by thin lines. The thick lines show the built-up inference chain that links pb-read to s-hunger. The boxes represent null pattern (all 0's) as an expansion terminator.

Even though both DYNASTY and PAM [Wilensky, 1978] use breadth-first search for the goal/plan-based inferencing, there is a essential difference in the search direction. PAM uses backward search from the input event to the stored goal/plan knowledge. PAM matches the input event to one of the rules, and continues to activate the applicable rules until one of the stored goal/plan knowledge structures is produced. Once the inference chain from the input event to the stored knowledge is acquired, PAM assumes that the input event is explained, and stops the rule application. DYNASTY uses forward search from the previously stored knowledge to the input event (new facts) which is a reverse direction of PAM's. From the proper stored knowledge (start-node), DYNASTY expands the goal/plan trees until the node which can be matched to the input event (target-node) is produced.

Two nodes must be defined for the GP-Associator to expand the gp-tree for building up an inference chain: the start-node and the target-node. The start-node defines where the search begins in the gp-tree, and the target-node defines where the search ends. In PAM's case, the target-node is previous goal/plan knowledge that PAM has in its episodic memory, and the start-node is a new fact stated in the input sentence [Wilensky, 1978]. However, since DYNASTY uses forward search, the role of start-node and target-node is reversed. The target-node always comes from the goal/plan representations that are selected from the current event in the story, that is, a new fact. The start-node has two different cases according to the accomplishment flag produced from the Plan-Selector, and these two cases correspond to the vertical and horizontal reasoning respectively (see section 1.1.3): (1) when the target-node is an unaccomplished goal or plan (the accomplishment flag is NO), then the target-node becomes the next start-node in the next inference cycle (vertical

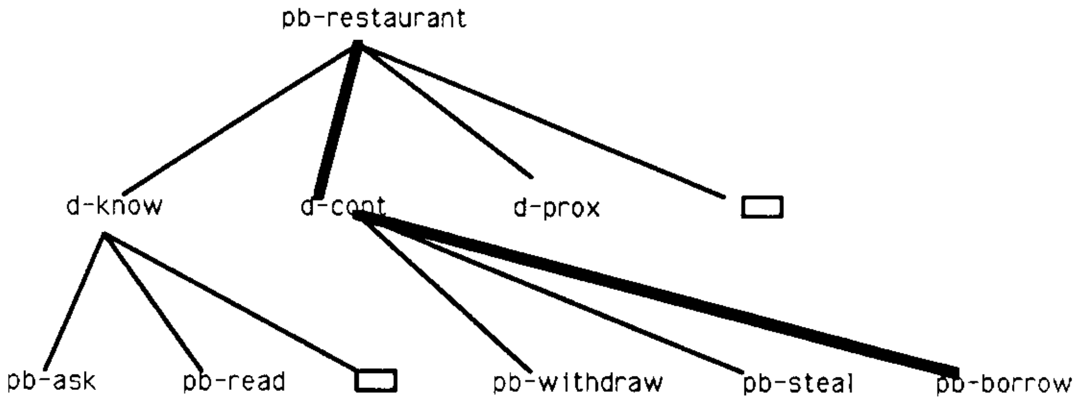


Figure 4.3: **Second gp-tree expansion.** Expansion from pb-restaurant plan to pb-borrow plan. The target-node (pb-borrow) is obtained from the input event, and the start-node (pb-restaurant) is determined using the start-node selection heuristic (see text for explanation).

reasoning), (2) when the target-node is an already accomplished goal or plan, then the first *unaccomplished* plan node in the inference chain becomes the next start-node in the next cycle (horizontal reasoning). The intuition behind this start-node selection heuristic is to focus the search process on the most relevant node in the gp-tree. When the current goal/plan is not accomplished, the planner should try to accomplish the current goal/plan (vertical reasoning), so the story continues to describe how the planner accomplishes the current goal/plan. When the current goal/plan is already accomplished, the planner should try to accomplish the first *unsuccessful* plan to which the current goal/plan is instrumental (horizontal reasoning), so the story continues to describe how the planner accomplishes another precondition for the pending goal/plan. These two cases define the two different types of goal/plan-based stories described in section 1.1.3.

For the above example (example story in section 4.3), when the current event (John read michelin-guide) selects pb-read (p34) as a plan, the next start-node will be pb-restaurant node since pb-read plan (p34) is already accomplished (flag YES) and pb-restaurant is the first unaccomplished plan in the output chain. The next target-node will be pb-borrow (p40) which is converted from the next sentence "John borrowed money from friend". Figure 4.3 shows the working memory after gp-tree expansion from pb-restaurant to the pb-borrow (p40) in the next inference cycle.

These expanded goals/plans are candidates for the inference chains. The symbolic working memory holds these goals/plans expanded, and the reverse pointers are followed to build up goal/plan inference chains. The reverse pointer of each node points to the parent node, and enables DYNASTY to form a single chain from the target-node to the start-node. The reason DYNASTY has these reverse pointers is that it implements the working memory using symbolic linked lists. DYNASTY does not have connectionist episodic memory model yet, and constructing it is one direction of the future research (see section 7.3.2). The resulting goal/plan inference chain for the first inference cycle is as follows:

[g35 GOAL s-hunger] [g35 AGENT John]
[p36 PLAN pb-restaurant] [p36 AGENT John] [p36 OBJECT ?food] [p36 LOCATION ?restaurant]
[g37 GOAL d-know] [g37 AGENT John] [g37 OBJECT location] [g37 OBJ-ATTR ?restaurant]
[p38 PLAN pb-read] [p38 AGENT John] [p38 OBJECT michelin-guide]

The resulting goal/plan inference chain for the second inference cycle is as follows:

[p41 PLAN pb-restaurant] [p41 AGENT John] [p41 OBJECT ?food] [p41 LOCATION ?restaurant]
[g42 GOAL d-cont] [g42 AGENT John] [g42 OBJECT money]
[p43 PLAN pb-borrow] [p43 AGENT John] [p43 OBJECT money] [p43 FROM friend]

4.3.3 Action-generating process

From the goal/plan inference chain, the Action-Generator produces actions/states when the goals/plans have been accomplished. For the first goal/plan chain in the above section, the result of the action-generating process is as follows:

[ev10 STATE hungry] [ev10 AGENT John] [ev10 MODE not]
[ev11 ACT ate] [ev11 AGENT John] [ev11 OBJECT ?food] [ev11 LOCATION ?restaurant]
[ev12 STATE knew] [ev12 AGENT John] [ev12 OBJECT location] [ev12 OBJ-ATTR ?restaurant]
[ev13 ACT read] [ev13 AGENT John] [ev13 OBJECT michelin-guide]

Note that some of the variables are instantiated (e.g. John, Michelin-guide) and some of them are not (e.g. ?food, ?restaurant). Since the instances like John and Michelin-guide are provided in the input sentence (story 5 in section 4.3), they can be propagated to instantiate the variables ?person and ?guide-book. The mechanism of this binding propagation lies in propagation of the specific ID value without alteration (see section 4.5). However, in the input sentence, there is no knowledge to instantiate the variables ?food and ?restaurant. More specifically, the variables ?food and ?restaurant were introduced by the GP-Associator module during the gp-tree expansion from the s-hunger goal to the pb-restaurant plan (for this particular input story). Since there were no instances for ?food and ?restaurant during the gp-tree expansion, these variables were not instantiated. This variable binding mechanism is similar to that of symbolic systems. The bindings are exact, and there are no guesses in the bindings. When the input sentences do not provide the instances, the particular variables cannot be instantiated. This is because DYNASTY is trained with only variables which vary randomly during training, and *is not trained with the specific instances for statistically biased guesses during bindings.* However, DYNASTY can be trained with the specific instances to do the approximate bindings. For example, during the GP-Associator training, we can set up the training data with specific instances for ?food (e.g. steak), then DYNASTY can instantiate the ?food with "steak" when no particular instance is given in the input story. These statistically-biased generalizations are discussed in section 5.3.1.

4.3.4 Goal/plan structure matching

Goal/plan structure matching in the gp-tree searches (for GP-Associator module) is performed by decoding each goal/plan representation into triples and accessing GDs for symbolic comparison. Since the GP-Associator module produces the structure with both variables and instances, the goal/plan structure matching needs unifying variables with the instances. The matching is two step process: (1) matching the goal/plan name and (2) matching the rest of the case-roles.

Goal/plan matching algorithm:

```
IF target-node (gp1) and generated-node (gp2) has the same gp-name
AND all case-role fillers are unifiable
THEN both goal/plan nodes are same and stop
ELSE continue gp-expanding
```

The unifiability of each filler is defined as follows:
filler1 (generated-node) and filler2 (target-node) are unifiable if and only if:

- filler1 and filler2 are variables and have the same representations (e.g. ?person == ?person)
- filler1 and filler2 are instances and have the same representations (e.g. John == John)
- filler1 is a variable and filler2 is an instance and both have the same DSR parts (versus ID parts) in the representations (e.g. ?person == John)

For example, suppose the target-node is pb-read (p34) and the newly generated node for one expansion (see figure 4.2) is pb-read (p44) as follows:

```
target-node
[p34 PLAN pb-read] [p34 AGENT John] [p34 OBJECT michelin-guide]
generated-node
[p44 PLAN pb-read] [p44 AGENT John] [p44 OBJECT ?guide-book]
```

First the goal/plan name is matched (control of matching process is done symbolically) and since they are same (accessing the same symbol pb-read in the GD), the rest of the case-role fillers can be matched according to the algorithm above. The michelin-guide and ?guide-book are unified according to the unification criteria above. The word-level matching is performed by accessing the GD; two word-level representations are the same if they access the same symbol. In the above example, the two goal/plan nodes p34 and p44 are considered to be the same and the goal/plan expansion stops at this point.

4.4 Surface generation

The surface generation process corresponds to steps 9 to 11 (see section 4.1). It first decodes the output action sequence into triples, and feeds those triples to the TS-Generator network. The TS-Generator produces word sequences according to the triples, and converts the word representations into the corresponding symbols. Suppose the Action-Generator's decoded output is as follows:

[ev10 STATE hungry] [ev10 AGENT John] [ev10 MODE not]
[ev11 ACT ate] [ev11 AGENT John] [ev11 OBJECT ?food] [ev11 LOCATION ?restaurant]
[ev12 STATE knew] [ev12 AGENT John] [ev12 OBJECT location] [ev12 OBJ-ATTR ?restaurant]
[ev13 ACT read] [ev13 AGENT John] [ev13 OBJECT michelin-guide]

The TS-Generator produces output paraphrases and replaces each representation for a word with its symbolic form. PC designates the pre-condition (the instrumental goal) and P designates the plan (see section 1.1 for the use of P and PC).

PC: John was not hungry.
P: John ate food at restaurant.
PC: John knew restaurant location.
P: John read michelin-guide.

The TS-Generator produces the necessary functional words such as "was" and "at" to help readability. The network is trained with the necessary functional words using the random representations in the GD in order to produce the correct English sentences.

4.5 Variable binding during goal/plan analysis

Variable binding is a difficult problem in connectionist models [Dyer, 1990b] since bindings must be done statistically, rather than symbolically. If DYNASTY handles only scripts such as DISPAR [Miikkulainen and Dyer, in press] or DISCERN [Miikkulainen, 1990a] which maps all the sentences in the particular input story to only one structure (particular script), then DYNASTY can use only statistical binding propagations using IDs. However, DYNASTY's input sentences (in one story) should be mapped to several goal/plan structures, and some of the variables are only instantiated later when DYNASTY reads in more sentences. So DYNASTY needs to backward propagate the bindings from the current sentence to the previously read sentences. This is why DYNASTY's binding propagation must be performed *using the ID value propagation [Miikkulainen, 1990a] for the forward-binding propagation and symbolic operations for the backward-binding propagation.*

4.5.1 Binding during plan-selection and action-generation

Variable binding during plan-selection and action-generation needs only forward propagation because the event structures and goal/plan structures have the same set of variables, and the two modules perform the mappings between events and goals/plans. As a result, variable binding during plan-selection and action-generation is performed through ID value propagation. The ID values in the input structure are passed without alteration to the output structures. Since the ID parts are set up randomly during training for all the variables, any kind of instance can be propagated without alteration during the performance phase. Here the BP-based networks perform the an identity function as far as the ID part is concerned. Consider the following training data for the Plan-Selector.

> IF [ev20 ACT asked] [ev20 AGENT ?person] [ev20 OBJECT location] [ev20 OBJ-ATTR ?restaurant]
> THEN [p30 PLAN pb-ask] [p30 AGENT ?person] [p30 OBJECT location] [p30 OBJ-ATTR ?restaurant]
> AND [YES]

For this training data, DYNASTY was trained to correctly pass the random IDs for ?person and ?restaurant variables from the "asked" event structure to the "pb-ask" plan structure. Suppose DYNASTY reads in the following input sentence:

> John asked about Sizzler location

The ST-Parser produces the following event structures:

> [ev20 ACT asked] [ev20 AGENT John] [ev20 OBJECT location] [ev20 OBJ-ATTR Sizzler]

Now this event structure should be an input to the Plan-Selector to select the pb-ask plan structure with **John** bound to ?person, and **Sizzler** bound to ?restaurant. The representation of **John** is specific ID plus DSR, where the DSR part is almost identical with that of ?person. Since the Plan-Selector was trained to pass random IDs of ?person without alteration, it can pass the specific ID for **John** also without alteration (approximately). So the instance **John** can be safely bound to ?person in the pb-ask plan structure. Similarly, when the Action-Generator generates the asking event (ev20) from the pb-ask plan (p30), the correct ID values can be propagated from the plan to the event structure.

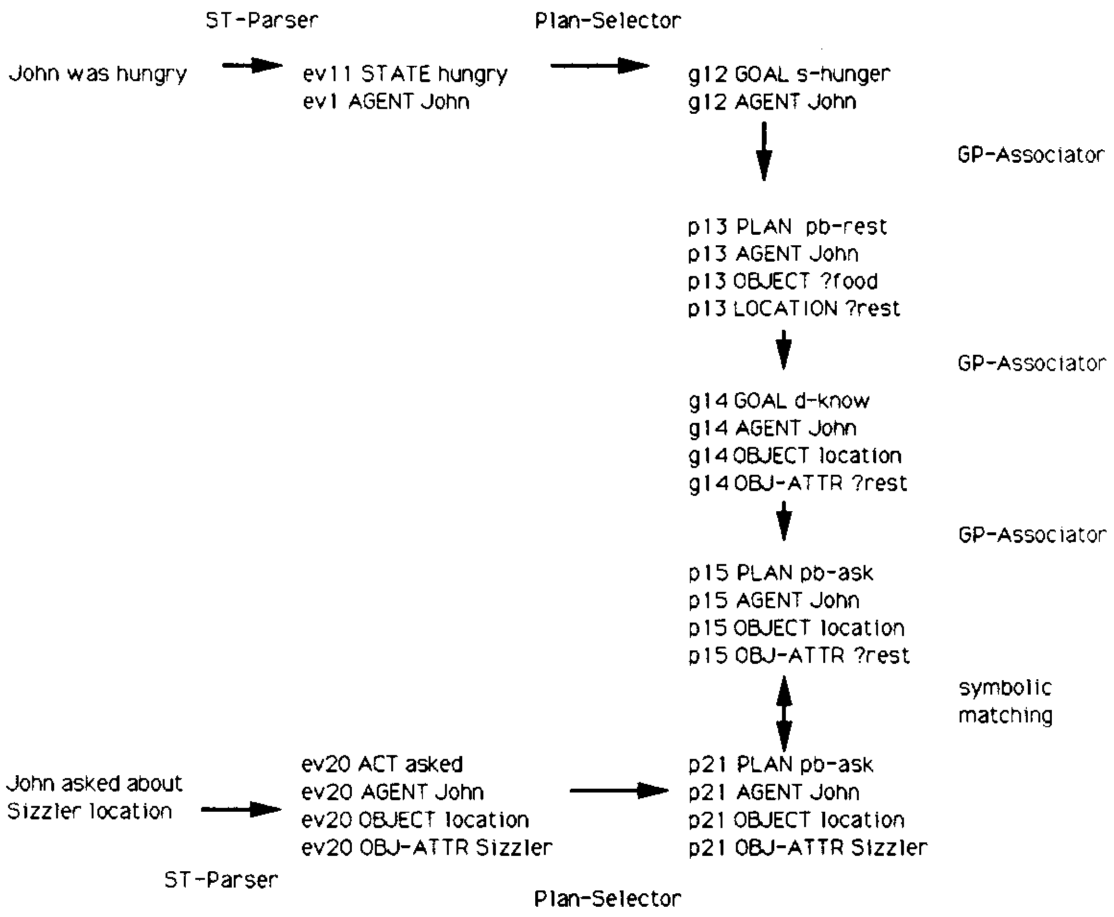


Figure 4.4: **Input to binding data flow.** The figure shows the mappings of two input sentences to the binding process. The first, second, and third column shows the input sentence, output of ST-Parser and output of Plan-Selector respectively. The down arrows in the third column shows the constructed inference chain by the GP-Associator.

4.5.2 Binding during gp-tree expansion

The goal/plan structures that should be associated by the GP-Associator usually do not have the same variable sets. Figure 4.4 shows the input story with 2 sentences, and the structure mappings from input to the binding process by each module in the DYNASTY.

The ST-Parser converts input sentences into the event structures, which are again converted into the goal/plan structures by the Plan-Selector. The variable ?restaurant cannot be instantiated until the second sentence provides the necessary instance **Sizzler**. If the two sentences can be mapped into one goal/plan structure (script), then the ?restaurant variable can be instantiated with the forward ID propagation.

Figure 4.5 shows variable binding propagation during gp-tree expansion with the start-node being s-hunger goal and the target-node being pb-ask plan. During the gp-tree expansion, the ID propagation is not enough to process all the bindings. For example, when the GP-Associator generates pb-restaurant from s-hunger, the instance **John** can be ID-propagated since both structure share ?person variable, but the instance for the variable ?restaurant cannot be determined until the next event is read. This is reasonable since there is no way of knowing at which restaurant John ate until we have a clue such as the second sentence "John asked about Sizzler location".[1] When the target-node matches the generated-node with variables, the instance in the target-node (e.g. Sizzler) should be backward propagated to replace the variable (e.g. ?restaurant) in the inference chain because this instance is a new valuable information that must be added to the inference chain. When John asked about Sizzler location, we can safely infer that John ate at the Sizzler, but we still don't know what John ate (for ?food). Hence, the gp-tree expansion has two different kinds of binding processes; forward and backward binding propagation.

Binding propagation during gp-tree expansion

- forward propagation : from start-node to target-node through ID value propagation (e.g. John).
- backward propagation: from target-node to start-node using symbolic copy operations. When the generated-node has uninstantiated variables and the target-node has their instances, replace every variable in the inference chain with the instances after unification matching (e.g. Sizzler).

4.6 Script processing

Script processing in DYNASTY [Lee et al., 1989; Lee et al., 1990] does not need a separate architecture since DYNASTY considers a script as a special type of plan. Suppose the input story is script-based as follows:

John was hungry. John entered Chart-House. John ate steak. John left a tip

[1]However, when DYNASTY is trained with the specific instances for statistically biased generalizations (section 5.3.1), DYNASTY can produce plausible, but sometimes incorrect guesses for the ?restaurant variable before the second sentence is read.

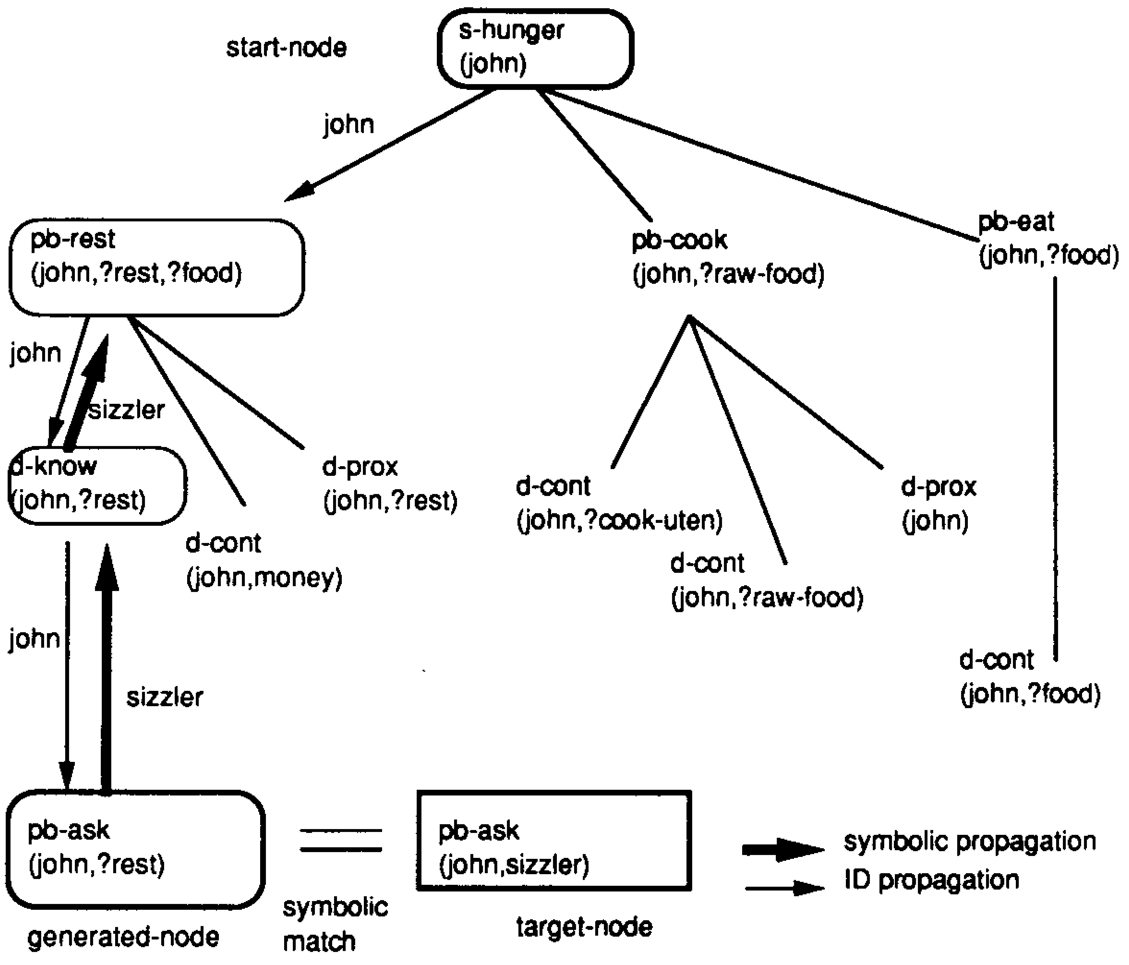


Figure 4.5: **Forward and backward binding propagation during gp-tree expansion.** The ?-mark shows that the word is a variable in the goal/plan structure. The instance word in the start-node (e.g. John) is forward propagated using ID value propagation while the instance word in the target-node (e.g. Sizzler) is backward propagated by a symbolic pattern copy. The node in the oval forms the resulting goal/plan inference chain.

Surface processing produces the event representations for each sentence:

[ev11 STATE hungry] [ev11 AGENT John]
[ev12 ACT entered] [ev12 AGENT John] [ev12 LOCATION Chart-House]
[ev13 ACT ate] [ev13 AGENT John] [ev13 OBJECT steak]
[ev14 ACT left] [ev14 AGENT John] [ev14 OBJECT tip]

All the script-related actions are grouped in one block to be recognized as single script by the user. DYNASTY currently cannot decide on its own when to group certain event sequences into a single script. For example, in the above events, after ev11 is processed, the event group ev12, ev13 and ev14 are fed to the Plan-Selector to select an appropriate script as a plan with the correct role-bindings. Script role-binding is a special case of variable binding and performed using only ID value propagation (forward propagation). In the above example, the ID value of John, Chart-house, and steak is propagated to the restaurant script representation. The Plan-Selector selects the s-hunger goal for the first event, and selects the pb-restaurant plan with customer bound to John, restaurant bound to Chart-House and food bound to steak for the group of events – ev12, ev13, and ev14. Since the script representation has all the necessary instances, no backward binding propagation is needed. The Plan-Selector's output is as follows:

[g30 GOAL s-hunger] [g30 AGENT John] [NO]
[p40 PLAN pb-restaurant] [p40 AGENT John] [p40 OBJECT steak] [p40 LOCATION Chart-House][YES]

The GP-Associator simply expands the gp-tree with start-node being g30 and target-node being p40, and produces a single inference chain that John has a plan of executing the restaurant script to satisfy his hunger. The produced inference chain goes to the Action-Generator to produce the state/actions. From the script representation, the Action-Generator expands all the events for the script, filling-in the omitted events. The output of Action-Generator is as follows:

[ev20 STATE hungry] [ev20 AGENT John] [ev20 MODE not]
[ev11 ACT entered] [ev11 AGENT John] [ev11 LOCATION Chart-House]
[ev12 ACT seated] [ev12 AGENT waiter] [ev12 OBJECT John]
[ev13 ACT brought] [ev13 AGENT waiter] [ev13 OBJECT menu]
[ev14 ACT read] [ev14 AGENT John] [ev14 OBJECT menu]
[ev15 ACT ordered] [ev15 AGENT John] [ev15 OBJECT steak]
[ev16 ACT ate] [ev16 AGENT John] [ev16 OBJECT steak]
[ev17 ACT paid] [ev17 AGENT John] [ev17 OBJECT bill]
[ev18 ACT left] [ev18 AGENT John] [ev18 OBJECT tip]
[ev19 ACT left-for] [ev19 AGENT John] [ev19 FROM Chart-House] [ev19 TO home]

These event representations are decoded using the Triple-Encoder, producing event triples and these event triples go to the TS-Generator to produce the script-based paraphrase. G means top-level goal, and P means selected plan in the paraphrase.

G: John was not hungry
P: John entered Chart-house
Waiter seated John
Waiter brought menu
John read menu
John ordered steak
John ate steak
John paid bill
John left tip
John left chart-house for home

Currently, DYNASTY can process 4 different types of script knowledge: going to restaurant, attending a lecture, going shopping and visiting a doctor, all taken from psychological experiments [Bower et al., 1979]. DYNASTY's script knowledge is defined in appendix C.3.

Part III

Evaluation

Chapter 5

DYNASTY learning and performance analysis

In the previous chapters, the training and process models of each component in DYNASTY have been presented. In this chapter, the DSRs that DYNASTY has learned from actual script/goal/plan-based stories are analyzed and compared to each other. The DYNASTY output traces for script/goal/plan-based stories are presented with high-level explanations along with each component's performance statistics. DYNASTY's generalization and fault-tolerance performances are also evaluated, with various story data.

5.1 Learning analysis

In this section, the DSRs learned from different proposition spaces are analyzed and compared. A proposition space designates a set of propositions which are drawn from the different sets of stories (see section 2.3). Our hypothesis is that the DSRs learned from a rich proposition space show more clear similarity properties compared to the DSRs from a poor proposition space. When we say that a proposition space A is richer than a proposition space B, we mean that the same words are associated with more propositions in space A than in space B. For example, the proposition space which is drawn from the whole stories (script/goal/plan-based) is richer than the one drawn from only goal/plan-based stories or only script-based stories. In this analysis, we excluded ID-units so each representation consists of 10 units. We use the same DSR-Learner program which was used in section 2.7. The program and data format is listed in appendix E.1.

5.1.1 DSRs in goal/plan processing

We have selected 6 goal/plan-based story skeletons to analyze the DSRs learned (see appendix B.1 for selected story skeletons). Each variable in the story skeletons is randomly replaced with their instances to produce actual propositions during learning. Figure 5.1 shows the DSR representations for the selected word-concepts. To compare the DSRs that are learned from the different proposition spaces (in the next two sections), we grouped all the word representations into 6 instance-words (instances for the variables), 9 act/state-words, and 4 general concept-words including physical-object, locations etc. This configuration was used for all of the next studies.

The figure shows that the words which have similar usages (similar semantics) in the given propositions develop similar representations. The similar usages designate what kind of semantic case-roles the words played in the given sentence. The instance-words (e.g. John, Mary) for the same variables (e.g. person) developed almost identical representations. Actually, these instance words played the same semantic case-roles in each proposition since we

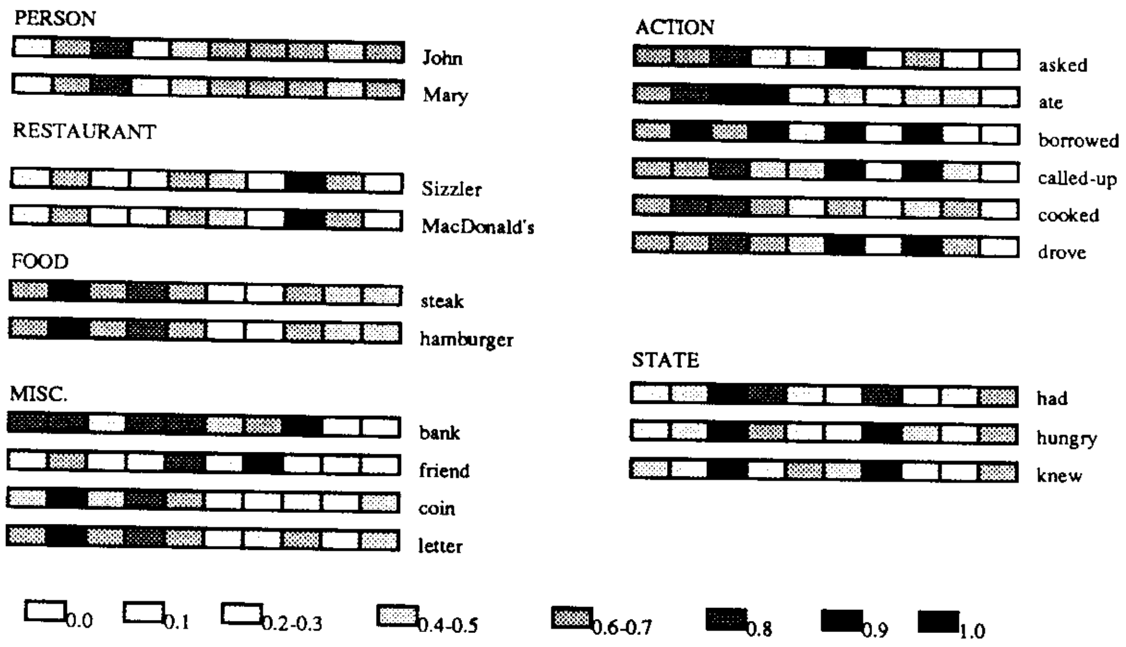


Figure 5.1: **Learned DSRs of concepts for the selected goal/plan-based stories.** Learning rate = 0.07 to 0.02; momentum factor = 0.5 to 0.9; 150 epochs for each concept and propositions; one epoch = 300 cycles of auto-associative backprop. The learning parameters are the same through out all of the DSR learning experiments to get consistent results.

replaced each variable with the instance word randomly for each epoch during training. All the act (or state) words (e.g. asked, ate) developed similar representations since they played ACT (or STATE) case-roles in every proposition. However since each word is associated with different propositions, their representations are not identical. The state-word group is more similar because their case structures are more uniform than the act-word group. (Most of the state-word group has only AGENT and OBJECT in the case structures.) The general concept-word group (under MISC in the figure 5.1) shows variations according to their semantic usage. **Coin** and **letter** have almost identical DSRs since they are mainly used in OBJECT roles. **Bank** and **friend** are mainly used in LOCATION or TO/FROM roles respectively, so their representations are not similar. The reason that **Sizzler** and **MacDonald's** are similar to **bank** is clear, because they are used mainly in LOCATION roles. **John** and **Mary** are not similar to any other word representations since they are the only words which are used in AGENT roles. Figure 5.2 shows hierarchical clustering analysis results for the developed DSRs, using the same program as in section 2.7.

The figure shows the global similarity structures for the developed DSRs. The instance-words are merged first (step 1, 2, 3) since they have the most similar representations. The act-word groups and state-word groups are merged next, and the general concept-word groups are merged with instance-word groups according to their case-roles. Globally, there are 5 merging groups: AGENT-words (step 9), OBJECT-words (step 14), LOCATION-words (step 5), ACT-words (step 13) and STATE-words (step 12). There are some anomalies shown in the figure 5.2. For example, **asked** should not be merged to **Mary**, and there is no reason that **borrowed** should be merged with **bank**. **MacDonald's** and **friend** are other anomalous examples. However, their merge step is not small, which means they are not that similar in actual representations (see figure 5.1) and it is not difficult for DYNASTY to distinguish them as different words. Conceptually, these anomalous examples can be a signal that the proposition space is not rich enough for the specific words in the experiment. Most of the anomalous merge cases disappeared when we ran the DSR learning experiments with script and goal/plan-based stories combined (see section 5.1.3).

5.1.2 DSRs in script processing

We selected 4 script-based story skeletons (appendix B.2) to analyze learned DSRs. Each variable (script-role) is replaced with their instances randomly during training. The 4 scripts are knowledge of *going to a restaurant, attending a lecture, grocery shopping,* and *visiting a doctor* taken from [Bower et al., 1979] (see appendix C.3 for the script knowledge used in DYNASTY). Figure 5.3 shows some of the learned DSRs for selected words to compare with the figure 5.1. The same group of words was selected: 6 instance-words, 9 act-words, and 4 general concept-words. Since script-based stories do not have state-words, we selected 9 act-words instead.

In figure 5.3, we can see the same kind of similarities as in the previous section, even though the representations themselves are different for the same word in the two experiments (for example, compare **Sizzler** in figure 5.1 and figure 5.3). The instance-words (e.g. **John, Mary**) for the same script role (e.g. **person**) still develop the most similar representations. The 9 act-word group has more similar representations than the goal/plan-based

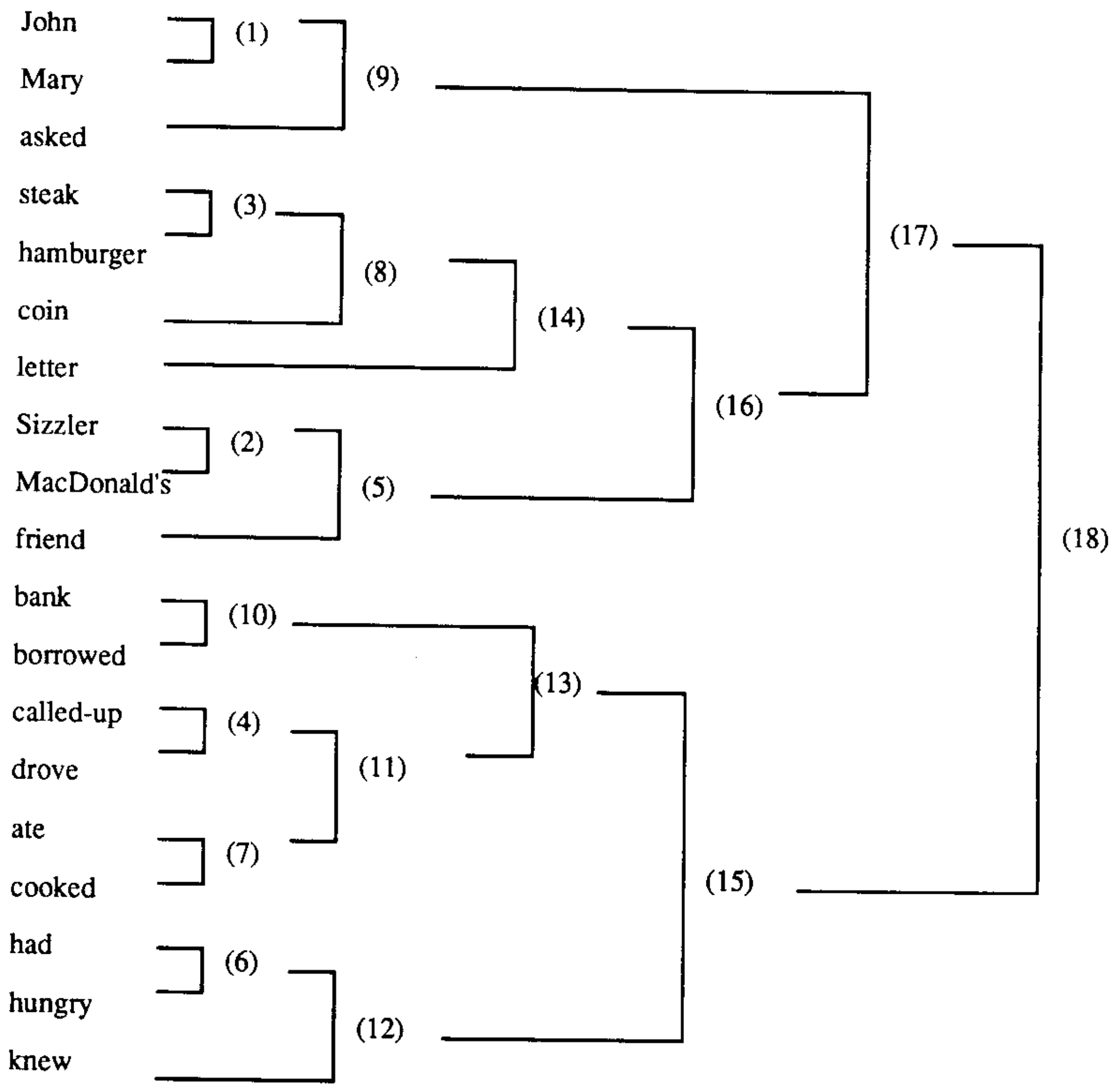


Figure 5.2: **Merge clustering results for DSRs in the goal/plan-based stories** The number in the parenthesis shows each merge step.

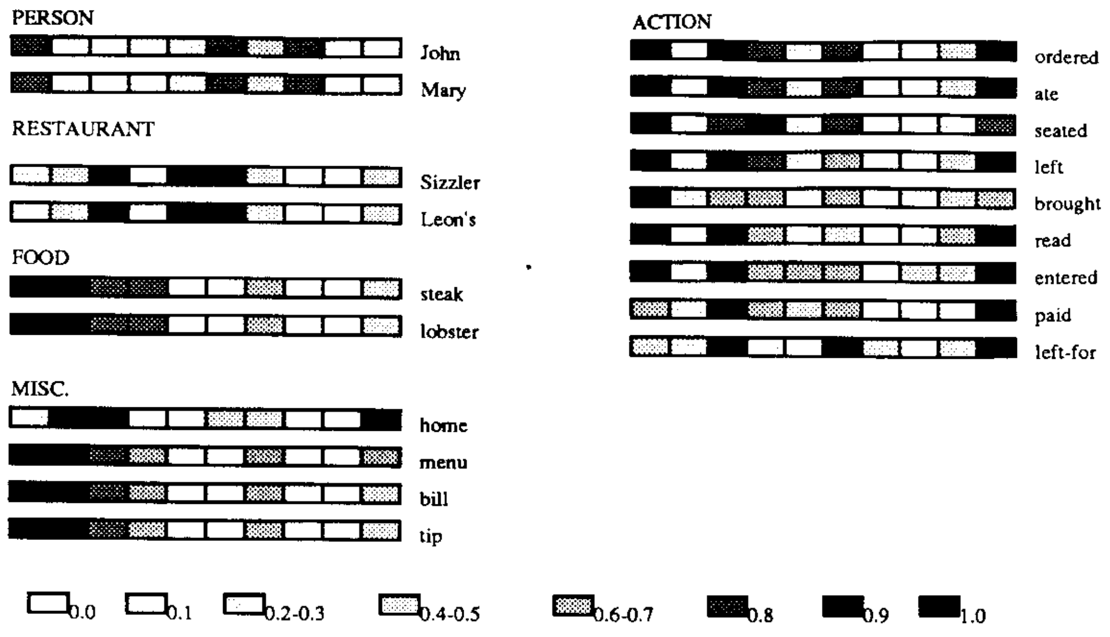


Figure 5.3: **Learned DSRs of concepts for script-based stories.** Experiments are run under the same conditions as in figure 5.1.

experiments, because script-based stories are more uniform than the goal/plan-based stories. In the script-based stories, each action is always carried out in the same sequence, which forces more regularities in the act-word group. In the general concept-word group, the 3 words **menu**, **bill** and **tip** are more similar than **home**, as expected. The words in the **food** group, which are mainly used in the OBJECT role, also develop similar representations, as the 3 general concept-words stated above.

Figure 5.4 shows the similarity structures by using the same merge clustering technique as before. This figure shows 3 major categories of words: (1) instance-words in the script roles (e.g. John, Mary , etc), (2) script actions (e.g. entered, paid, etc), and (3) the remaining concepts used in the script (e.g. menu, bill, etc). In each category, the most similar words (e.g. Sizzler and Leon's) in the script context started to merge together. The reason non-intuitive clustering occurs, such as **left-for** and **home**, is that they are under-defined in the proposition space. In this case, the two words are merged at step 11, so the anomalous clustering does not affect the system's performance. The words **steak** and **lobster** are also unintuitively merged with the word **home** (at step 14), for the same reason. In general, the clusters in the script experiment are more natural than the goal/plan-based experiments since script-based stories force each word to be used stereotypically.

5.1.3 DSRs in combined script/goal/plan processing

We have run the same experiments with a proposition space which is drawn from both script-based and goal/plan-based story skeletons. In this experiment, the stories in appen-

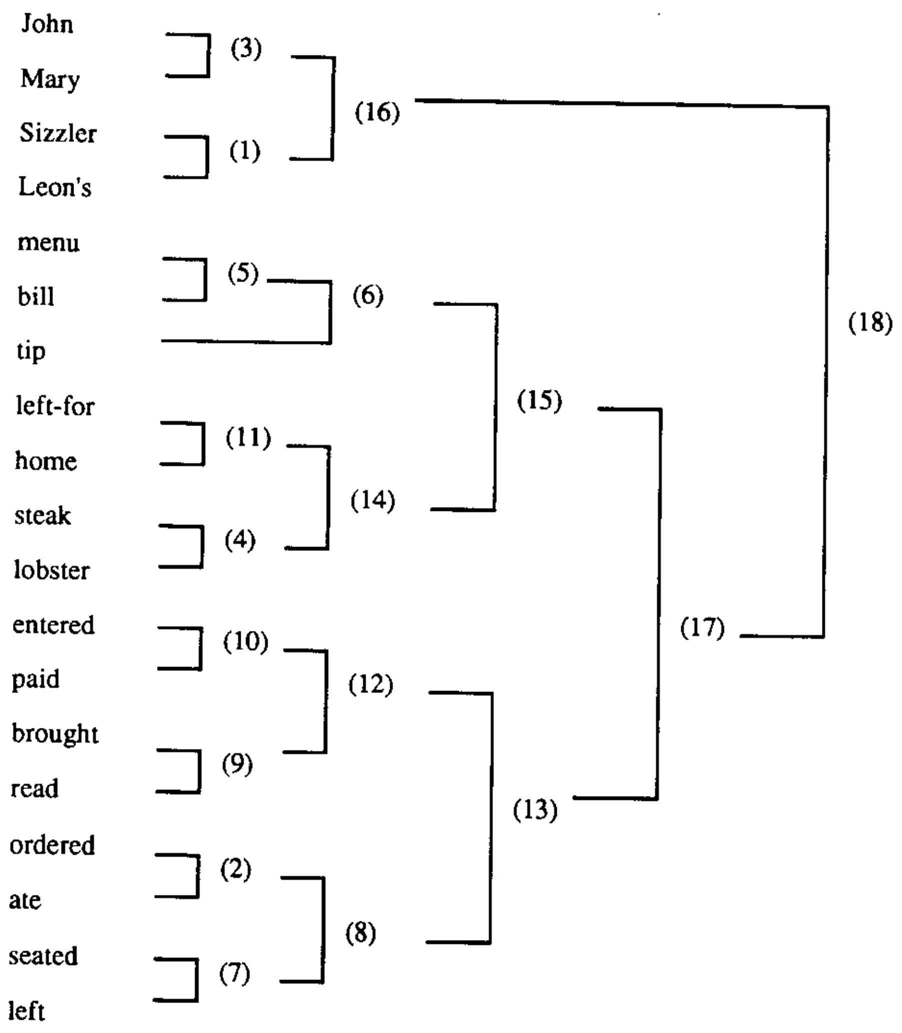


Figure 5.4: **Merge clustering of the learned DSRs for the script stories.**

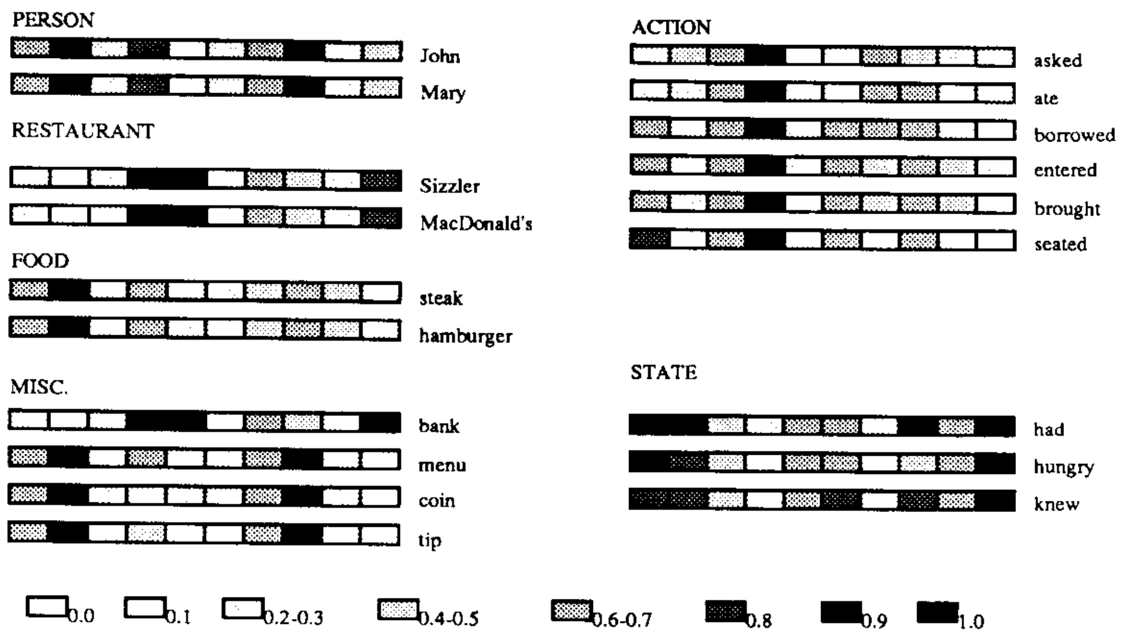


Figure 5.5: **Learned DSRs for the combined proposition space.**

dix B.1 and appendix B.2 are combined to learn DSRs. Our objective is to show that this proposition space is richer than the previous two experiments, so the anomalous clusterings will hopefully disappear and DSRs will show more clear similarity structures. Figure 5.5 shows the resulting DSRs. We selected the same instance-word group, such as **person**, **restaurant** and **food** group as in the previous two experiments. Half of the words in the act, state, and general concept-word group are from the goal/plan-based experiment (figure 5.1) and half of them are from the script-based experiment (figure 5.3) to clearly compare them.

The results are positive. The act-word groups show more similar representations than the previous two experiments, and so do the 3 words **menu**, **coin**, and **tip** in the general concept-word group (under the MISC in the figure). In the clustering analysis result (figure 5.6), the anomalous merging of **Mary** to **asked**, and **bank** to **borrowed** have disappeared. In this figure, the merge of **MacDonald's** and **bank** makes sense since both words are mainly used as LOCATION-roles in the propositions.

Our conclusion is that combined stories give richer proposition spaces than solely script-based or goal/plan-based stories, since they have more propositions associated with each single word during DSR learning. Script-based stories seem to have richer proposition spaces than do goal/plan-based stories because the same word is repeatedly associated with the same sequence of actions.

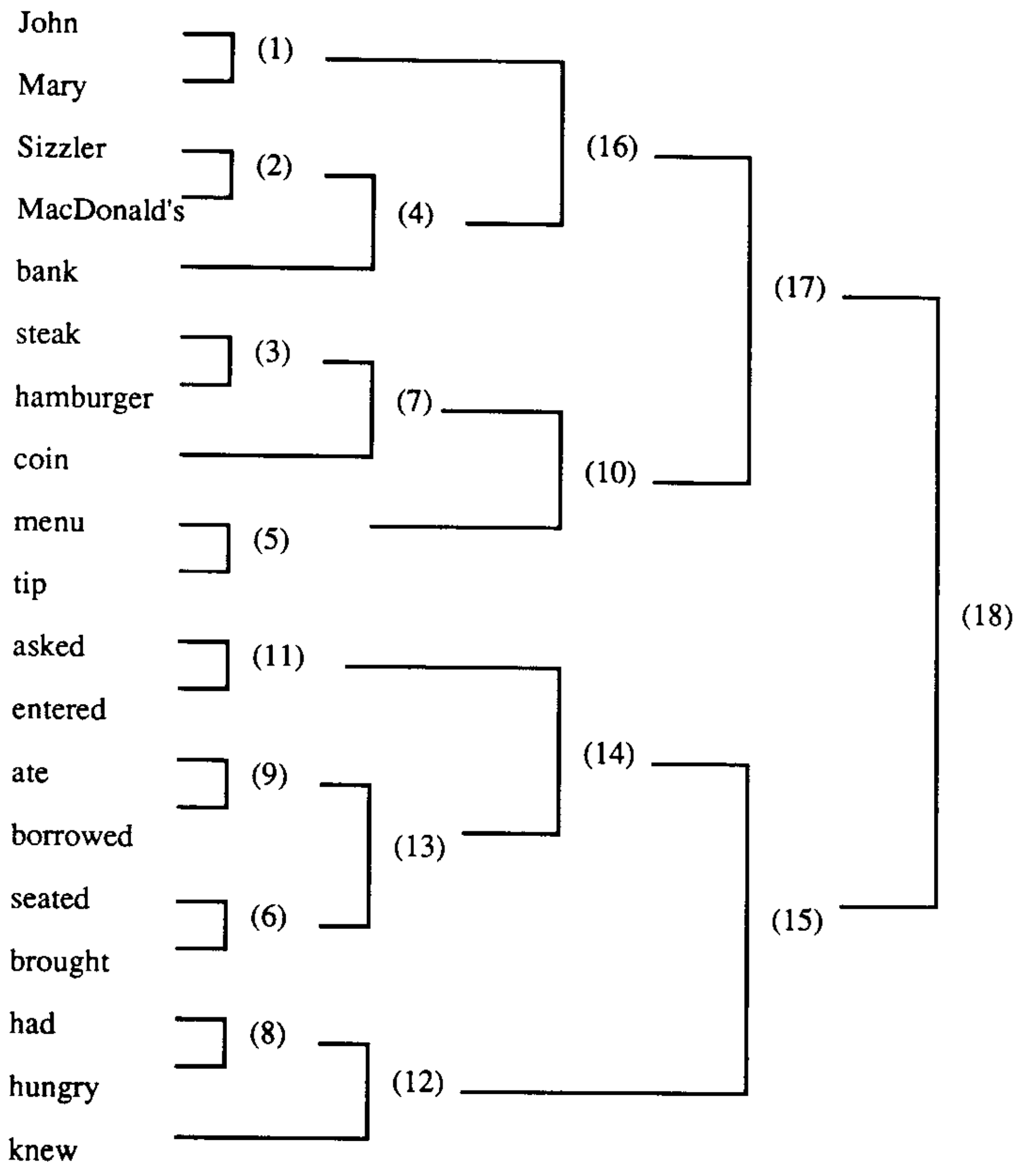


Figure 5.6: **Merge clustering of DSRs for the script/goal/plan combined stories.**

5.2 Performance analysis

In the last section, we analyzed DSRs learned from script/goal/plan-based stories. DYNASTY's word representation consists of DSRs and IDs (see section 2.9). DSRs give similarity-based generalization and fault-tolerant abilities to the system since similar words have similar representations. IDs [Miikkulainen and Dyer, in press] enable the system to keep the variable bindings straight. In this section, we show DYNASTY's output traces with edited explanations (using the program and data format listed in appendix F). Output traces are shown for both goal/plan-based stories and script-based stories. At the end of each trace, we show performance statistics in table form. The performance statistics for processing whole stories are also shown. In the example traces, the system's output is in type-font, and in lower case. The events, goals, and plans are arbitrarily numbered when they are decoded, and the purpose of numbering is to group the relevant triples into one event (or goal/plan). At each pattern transformation step, DYNASTY decodes the output pattern into triple structures and translates them into the corresponding symbols for the users.

5.2.1 Goal/plan-based story processing

This section shows an example of goal/plan-based story processing traces and performance statistics. DYNASTY reads the following input story:

```
DYNASTY> give me a story
```
John was hungry. John asked friend about Sizzler location. John drove to Sizzler. John had no money. John wanted (John call-up to friend). John borrowed coin from waiter.

DYNASTY accesses the GD to convert each word symbol into its vector representation, and feeds them to the ST-Parser, word by word, producing event triples. If the word is not listed in the GD, DYNASTY simply ignores the word (e.g. some functional words such as **about** are not in the GD). When the sentence has parentheses, DYNASTY reads the sentence inside the parentheses first and uses the representations for this inner sentence as a constituent component for the outer sentence. At this point, the ST-Parser is interacting with the Triple-Encoder to get the inner event representations. The event numbers are arbitrarily assigned by the DYNASTY main program. Now here is the trace of st-parsing for the first sentence in the story:

```
input event:  john was hungry
st-parser:
st-parsing...
[ev1 STATE hungry], [ev1 AGENT john]
```

The Triple-Encoder builds-up an event representation for this event. The Plan-Selector chooses a goal/plan for this event with the correct bindings (e.g John as an AGENT). The binding information is propagated through the ID units. In this example, since **John** is the AGENT in ev1, the same ID for **John** is propagated to the goal/plan representation, selecting g2 with **John** as an AGENT (planner). The selected goal/plan representations are from the goals/plans which are in the DYNASTY gp-tree (see appendix C.2 for entire DYNASTY gp-tree). The YES/NO flag designates the success status of the goal/plan at the current point. Now follows the trace for goal/plan selection for the first input sentence:

```
triple-encoder:
event representations formed for hungry
plan-selector:
selecting g/p and decoding....
[g2 GOAL s-hunger] [g2 AGENT john] [NO]
```

For the first goal (top-level goal), the GP-Associator simply passes it to the Action-Generator and marks it as a start-node for the next gp-tree expansion cycle. The Action-Generator produces corresponding event for this goal. The decoded event goes to the TS-generator and output sentence is produced as follows:

```
gp-associator:
first start-node s-hunger
[g2 GOAL s-hunger] [g2 AGENT john]

action-generator:
generating actions and decoding ...
[ev3 STATE hungry] [ev3 AGENT john] [ev3 MODE not]
ts-generator:
ts-generating...
john was not hungry
```

DYNASTY continues to process the second sentence. First, DYNASTY st-parses the sentence, builds up the event representations, and selects a proper goal/plan for this event with correct bindings. In this example, the ID-units in the AGENT **John** and OBJ-ATTR **Sizzler** are propagated to the goal/plan representations to select p5.

```
input event:  john asked friend about sizzler location
st-parser:
st-parsing...
[ev4 ACT asked] [ev4 AGENT john] [ev4 TO friend] [ev4 OBJECT location]
[ev4 OBJ-ATTR sizzler]
triple-encoder:
event representation formed for asked
plan-selector:
selecting gp and decoding...
[p5 PLAN pb-ask] [p5 AGENT john] [p5 TO friend] [p5 OBJECT location]
[p5 OBJ-ATTR sizzler] [YES]
```

Taking this goal/plan as a target-node, DYNASTY starts to expand the encoded gp-tree using the GP-Associator. The start-node was defined as gl (s-hunger) in the previous cycle since "s-hunger" was unaccomplished in the previous cycle. The gp-tree expansion continues until DYNASTY generates the target-node ("pb-ask" for this cycle). After the gp-tree is expanded, DYNASTY builds up the inference chain by following the reverse pointers in the working memory (symbolic link traversing). When a variable-instance match occurs, DYNASTY copies the ID part of the instance (in the target-node) to the variable (in the matched generated node). This copied ID is propagated backward to the start-node for binding variables which cannot be forward propagated from the start-node (see section 4.5).

```
gp-associator:
expanding gp-tree
start-node:  s-hunger
target-node :  pb-ask
expanding ...

s-hunger pb-restaurant pb-cook pb-eat d-know d-cont d-prox d-cont d-cont
d-prox d-cont pb-ask

building inference chain and backward binding propagation...
[g6 GOAL s-hunger] [g6 AGENT john]
[p7 PLAN pb-restaurant] [p7 AGENT john] [p7 OBJECT ?food] [p7 LOCATION
sizzler]
[g8 GOAL d-know] [g8 AGENT john] [g8 OBJECT location] [g8 OBJ-ATTR sizzler]

[p9 PLAN pb-ask] [p9 AGENT john] [p9 TO friend] [p9 OBJECT location]
[p9 OBJ-ATTR sizzler]
```

DYNASTY selects the start-node for the next inference cycle at this point. Since the pb-ask plan is successful (flag is YES), so is the d-know goal. The first unaccomplished plan node is pb-restaurant, so according to the start-node selection heuristics (in section 4.3.2), it becomes the next start-node. Next, DYNASTY generates events for the goals and plans in the inference chain and feeds them to the TS-Generator. The TS-Generator produces the explanation of John's action in the story as a goal/plan chain (some of the functional words are added for the readability):

```
action-generator:
generating actions and decoding...
[ev10 STATE hungry] [ev10 AGENT john] [ev10 MODE not]
[ev11 ACT ate] [ev11 AGENT john] [ev11 OBJECT ?food] [ev11 LOCATION
sizzler]
[ev12 STATE knew] [ev12 AGENT john] [ev12 OBJECT location] [ev12 OBJ-ATTR
sizzler]
[ev13 ACT asked] [ev13 AGENT john] [ev13 TO friend] [ev13 OBJECT location]
[ev13 OBJ-ATTR sizzler]
```

```
ts-generator:
ts-generating...
john was not hungry
john ate ?food at sizzler
john knew sizzler location
john asked friend about sizzler location
```

The remaining sentences are processed in the same manner, and the trace is listed below. The third sentence is st-parsed, and "pb-drive" is selected as a relevant plan.

```
input event:  john drove to sizzler
st-parser:
st-parsing...
[ev14 ACT drove] [ev14 AGENT john] [ev14 TO sizzler]
triple-encoder:
event representation formed for drove
plan-selector:
selecting gp and decoding...
[p15 PLAN pb-drive] [p15 AGENT john] [p15 TO sizzler] [YES]
```

Since "pb-restaurant" was determined as a start-node in the previous cycle, DYNASTY expands gp-tree from "pb-restaurant" to "pb-drive". Since "pb-restaurant" already instantiated all the variables, the forward ID propagation does all the necessary binding. And no backward binding propagation occurs for this goal/plan chain. We cannot know what food John ate, so the variable ?food is not instantiated. The start-node for the next inference cycle is determined as the same "pb-restaurant" because this plan is not yet satisfied.

```
gp-associator:
expanding gp-tree
start-node:  pb-restaurant
target-node :  pb-drive
expanding ...

pb-restaurant d-know d-cont d-prox pb-ask pb-read pb-withdraw pb-steal
pb-borrow pb-walk pb-drive

building inference chain and backward binding propagation...
[p16 PLAN pb-restaurant] [p16 AGENT john] [p16 OBJECT ?food] [p16 LOCATION
sizzler]
[g17 GOAL d-prox] [g17 AGENT john] [g17 TO sizzler]
[p18 PLAN pb-drive] [p18 AGENT john] [p18 TO sizzler]


action-generator:
generating actions and decoding...
[ev19 ACT ate] [ev19 AGENT john] [ev19 OBJECT ?food] [ev19 LOCATION
sizzler]
[ev20 STATE inside] [ev20 AGENT john] [ev20 LOCATION sizzler]
[ev21 ACT drove] [ev21 AGENT john] [ev21 TO sizzler]
ts-generator:
ts-generating...
john ate ?food at sizzler
john was inside sizzler
john drove to sizzler
```

The fourth sentence is turned into a "d-cont (money)" goal, and the goal/plan chain is constructed from the "pb-restaurant" plan to "d-cont (money)" goal. The start-node for next inference cycle is determined as "d-cont (money)" goal because the goal is not accomplished yet (NO flag).

```
input event:  john had no money
st-parser:
st-parsing...
[ev22 STATE had] [ev22 AGENT john] [ev22 OBJECT money] [ev22 MODE not]

triple-encoder:
event representation formed for had
plan-selector:
selecting gp and decoding...
[g23 GOAL d-cont] [g23 AGENT john] [g23 OBJECT money] [NO]

gp-associator:
expanding gp-tree
start-node:  pb-restaurant
target-node :  d-cont
expanding ...
pb-restaurant d-know d-cont d-prox pb-ask pb-read d-cont
building inference chain and backward binding propagation...
[p24 PLAN pb-restaurant] [p24 AGENT john] [p24 OBJECT ?food] [p24 LOCATION
sizzler]
[g25 GOAL d-cont] [g25 AGENT john] [g25 OBJECT money]


action-generator:
generating actions and decoding...
[ev26 ACT ate] [ev26 AGENT john] [ev26 OBJECT ?food] [ev26 LOCATION
sizzler]
[ev27 STATE had] [ev27 AGENT john] [ev27 OBJECT money]
```

```
ts-generator:
ts-generating...
john ate ?food at sizzler
john had money
```

Until the fourth sentence, the story is about how John fulfills the various preconditions to the "pb-restaurant" plan (horizontal reasoning). From the fifth sentence on, the story is about how John fulfills the series of preconditions for the unaccomplished goal "d-cont (money)" (vertical reasoning). The fifth and sixth sentence explain how John can have money to dine at the restaurant by meeting a series of preconditions. The fifth sentence is an embedded sentence. DYNASTY st-parses the inner sentence first, and uses the representation to fill the OBJECT role for the outer sentence. DYNASTY selects the pb-phone plan for this sentence, and makes this plan the start-node for the next cycle since this plan is not accomplished yet. The output inference chain explains that the reason for the pb-phone plan is to borrow money from a friend, and the pb-borrow plan is for d-cont (money) in order to dine at the restaurant.

```
input event:  john wanted (john called-up friend)
st-parser:
st-parsing...
[ev28 ACT called-up] [ev28 AGENT john] [ev28 TO friend]
[ev29 STATE wanted] [ev29 AGENT john] [ev29 OBJECT ev28]
triple-encoder:
event representation formed for wanted
plan-selector:
selecting gp and decoding...
[p30 PLAN pb-phone] [p30 AGENT john] [p30 TO friend] [NO]

gp-associator:
expanding gp-tree
start-node:  d-cont
target-node :  pb-phone
expanding ...
d-cont pb-withdraw pb-steal pb-borrow d-prox d-link pb-walk pb-drive
pb-phone
building inference chain and backward binding propagation...
[g31 GOAL d-cont] [g31 AGENT john] [g31 OBJECT money]
[p32 PLAN pb-borrow] [p32 AGENT john] [p32 OBJECT money] [p7 FROM friend]

[g33 GOAL d-link] [g33 AGENT john] [g33 TO friend]
[p34 PLAN pb-phone] [p34 AGENT john] [p34 TO friend]

action-generator:
generating actions and decoding...
```

```
[ev35 STATE had] [ev35 AGENT john] [ev35 OBJECT money]
[ev36 ACT borrowed] [ev36 AGENT john] [ev36 OBJECT money] [ev36 FROM
friend]
[ev37 STATE had] [ev37 AGENT john] [ev37 OBJECT comm-link] [ev38 TO
friend]
[ev38 ACT called-up] [ev38 AGENT john] [ev38 TO friend]
ts-generator:
ts-generating...
john had money
john borrowed money from friend
john had comm-link to friend
john called-up friend
```

For the sixth sentence, the output inference chain explains that the reason for the pb-borrow (coin) plan is to get a coin for the pb-phone plan.

```
input event:  john borrowed coin from waiter
st-parser:
st-parsing...
[ev39 ACT borrowed] [ev39 AGENT john] [ev39 OBJECT coin] [ev39 FROM
friend]
triple-encoder:
event representation formed for borrowed
plan-selector:
selecting gp and decoding...
[p40 PLAN pb-borrow] [p40 AGENT john] [p40 OBJECT coin] [p40 FROM friend]
[YES]

gp-associator:
expanding gp-tree
start-node:  pb-phone
target-node :  pb-borrow
expanding ...
pb-phone d-know d-cont d-prox pb-ask pb-read pb-borrow
building inference chain and backward binding propagation...
[p41 PLAN pb-phone] [p41 AGENT john] [p41 TO friend]
[g42 GOAL d-cont] [g42 AGENT john] [g42 OBJECT coin]
[p43 PLAN pb-borrow] [p43 AGENT john] [p43 OBJECT coin] [p43 FROM waiter]

action-generator:
generating actions and decoding...
[ev44 ACT called-up] [ev44 AGENT john] [ev44 TO friend]
```

```
[ev45 STATE had] [ev45 AGENT john] [ev45 OBJECT coin]
[ev46 ACT borrowed] [ev46 AGENT john] [ev46 OBJECT coin] [ev46 FROM
waiter]
ts-generator:
ts-generating...
john called-up friend
john had coin
john borrowed coin from waiter

processing completed.
DYNASTY> give me a story
```

Now we can return to the unaccomplished top-level goal, and make it successful at this point. Since the pb-borrow (coin) plan is successful, the pb-phone plan is also accomplished. This pb-phone makes d-cont (money) and pb-restaurant successful, which results in the accomplished s-hunger goal. The entire I/O for this story is listed below. Each sentence with a □ mark is an input sentence, and output inferences follow each input sentence.

□ john was hungry
john was not hungry

□ john asked friend about sizzler location
john was not hungry
john ate ?food at sizzler
john knew sizzler location
john asked friend about sizzler location

□ john drove to sizzler
john ate ?food at sizzler
john was inside sizzler
john drove to sizzler

□ john had no money
john ate ?food at sizzler
john had money

□ john wanted (john called-up friend)
john had money
john borrowed money from friend
john had comm-link to friend
john called-up friend

□ john borrowed coin from waiter
john called-up friend
john had coin
john borrowed coin from waiter

Modules	Correctness	E_{avg}
ASCII-to-DSR-GD	89	0.037
st-parser	84	0.027
triple-encoder	81	0.026
plan-selector	93	0.054
gp-associator	82	0.060
action-generator	100	0.031
ts-generator	85	0.034
DSR-to-ASCII GD	76	0.036

Table 5.1: **Goal/plan analysis performance statistics for the connected modules.** The "Correctness" designates the percentage of the correct outputs compared with the desired outputs. Correct output means that each and every unit in a output vector is within 0.15 of the unit value in the desired output vector. The value range is 0.0 to 1.0. E_{avg} designates an average Euclidean distance between the actual outputs and the desired outputs over all the units.

Note the variable ?food is not still instantiated in this inference chain since there is no way of knowing what John ate at Sizzler from the input story. During training, DYNASTY experiences all kinds of ?food that John ate at Sizzler because the ID for the ?food variable is set up as random in the training data. In other words, DYNASTY training with ID was not statistically biased for certain instances. So in the output inference chain, the ID pattern for ?food is simply a blending of all kinds of possible foods. However, statistically biased generalization is possible in DYNASTY (see section 5.3.1). If we train DYNASTY with **John** always eating **steak** at the restaurant by *setting IDs for ?person and ?food together*, then we can get a default **steak** for ?food in the above story (as in section 5.3.1).

Other DYNASTY I/O examples can be obtained from the story skeletons in appendix B.1 by replacing variables with suitable instance words. Table 5.1 shows the performance data for each module when they are connected to process the 12 goal/plan-based stories randomly produced from the story skeletons in the appendix B.1. To get the performance data, each skeleton in the appendix B.1 is instantiated as two different stories by replacing different instance words for the same variables. Table 5.1 shows over 76% correctness for all modules, which is good for the chains of 8 modules connected to do the complex tasks. The average Euclidean distance errors are less than 0.06 for all of the modules. The performance statistics programs are listed in appendix G.1.

5.2.2 Script-based story processing

This section shows output traces for the script-based stories. DYNASTY processes the script-based stories in the same manner as the goal/plan-based stories. The full expansion of the script-based stories, including unmentioned events in the input sentence, are produced as an output. DYNASTY reads the following shopping story:

```
DYNASTY> give me a story.
```

Mary needed milk. Mary entered Vons. Mary picked-up milk. Mary paid the cashier.

The first event is processed to set up the top level goal as in the previous section.

```
input event:  mary needed milk
st-parser:
st-parsing..
[ev1 STATE needed] [ev1 AGENT mary] [ev1 OBJECT milk]


triple-encoder:
event representations formed for needed
plan-selector:
selecting g/p and decoding...
[g2 GOAL d-cont] [g2 AGENT mary] [ev2 OBJECT milk] [NO]

action-generator:
generating actions and decoding...
[ev3 STATE had] [ev3 AGENT mary] [ev3 OBJECT milk]
ts-generator:
ts-generating...
mary had milk
```

DYNASTY processes the second block of events. DYNASTY processes a group of events to select one plan (script). The grouping of events into a block is designated in the input story by the user. DYNASTY knows in advance whether the stories are script-based or goal/plan-based. If more than one sentence are grouped into one processing block, then the story is script-based. This is because DYNASTY assumes that a non-scriptal plan has only one act associated with it in the current implementation. For this block of events, DYNASTY selects "pb-shopping" plan.

```
input event:  mary entered vons.  mary picked-up milk.  mary paid to
cashier.
st-parser:
st-parsing...
[ev4 ACT entered], [ev4 AGENT mary], [ev4 TO vons]
```

```
[ev5 ACT picked-up], [ev5 AGENT mary], [ev5 OBJECT milk]
[ev6 ACT paid], [ev6 AGENT mary], [ev6 OBJECT money], [ev6 TO cashier]
triple-encoder:
event representation formed for entered, picked-up, paid
plan-selector:
selecting gp and decoding...
[p7 plan pb-shopping] [p7 AGENT mary] [p7 OBJECT milk] [p7 LOCATION
vons] [YES]
```

From the top-level goal, DYNASTY expands gp-tree to find the "pb-shopping" plan. This search is simple because script-based stories are centered around a single plan.

```
gp-associator:
expanding gp-tree
start-node:  d-cont
target-node:  pb-shopping
expanding ...
d-cont pb-shopping
building inference chain and backward binding propagation...
[g8 GOAL d-cont] [g8 AGENT mary] [g8 OBJECT milk] [p9 PLAN pb-shopping]
[p9 AGENT mary] [p9 OBJECT milk] [p9 LOCATION vons]
```

From the selected plan, DYNASTY expands the full sequence of events for this plan, filling-in unmentioned events in the input story.

```
action-generator:
generating actions and decoding...
[ev10 STATE had] [ev10 AGENT mary] [ev10 OBJECT milk]
[ev11 ACT entered] [ev11 AGENT mary] [ev11 LOCATION vons]
[ev12 ACT got] [ev12 AGENT mary] [ev12 OBJECT cart]
[ev13 ACT picked-up] [ev13 AGENT mary] [ev13 OBJECT milk]
[ev14 ACT waited] [ev14 AGENT mary] [ev14 LOCATION line]
[ev15 ACT paid] [ev15 AGENT mary] [ev15 OBJECT money] [ev14 TO cashier]

[ev16 ACT left-for] [ev16 AGENT mary] [ev16 FROM vons] [ev14 TO home]
ts-generator:
ts-generating...
mary had milk
mary entered vons
mary got cart
mary picked-up milk
mary waited in line
mary paid to cashier
mary left vons for home
```

```
processing completed.
DYNASTY> give me a story
```

The entire I/O for this story is listed below. The input sentences start with the □ mark as before. The script-based paraphrases follow the second sentence block.

```
□ mary needed milk
mary had milk

□ mary entered vons
□ mary picked-up milk
□ mary paid to cashier
mary had milk
mary entered vons
mary got cart
mary picked-up milk
mary waited in line
mary paid to cashier
mary left vons for home
```

The other script-based I/O examples can be obtained from the story skeletons in appendix B.2 by replacing variables with the suitable instance words. Table 5.2 shows the performance data for each module when they are connected to process the 12 script-based stories generated from these story skeletons. To get the performance data, each skeleton in the appendix B.2 produced 3 different script-based stories by replacing variables with different instance-words. Table 5.2 shows slightly better performance than the goal/plan based story performance. All of the modules have over 77% correctness and under 0.035 average Euclidean distance error.

Table 5.3 shows the performance data when DYNASTY processes the 12 goal/plan-based stories and 12 script-based stories together (generated from the skeletons in appendix B.1 and appendix B.2).

5.2.3 Role-binding errors

DYNASTY's role-binding errors occur when word concepts are confused with similar concepts, or word instances are confused with different instances in the same conceptual category. Consider the following example (role-binding errors are surrounded with asterisks, with correct outputs in the parentheses):

```
input event:  john read michelin-guide
st-parser:
stparsing...
[ev14 ACT read] [ev14 AGENT john] [ev14 OBJECT michelin-guide]
triple-encoder:
event representation formed for read
```

Modules	Correctness	E_{avg}
ASCII-to-DSR-GD	90	0.026
ST-parser	86	0.027
triple-encoder	82	0.028
plan-selector	93	0.030
gp-associator	100	0.021
action-gen	89	0.035
TS-gen	87	0.034
DSR-to-ASCII-GD	77	0.029

Table 5.2: **Script processing performance for the connected modules.** The definition of Correctness and E_{avg} is the same as in the table 5.1.

Modules	Correctness	E_{avg}
ASCII-to-DSR-GD	85	0.038
ST-parser	82	0.031
triple-encoder	80	0.027
plan-selector	91	0.044
gp-associator	82	0.061
action-gen	99	0.032
TS-gen	84	0.035
DSR-to-ASCII-GD	75	0.032

Table 5.3: **Script/goal/plan combined processing performance.**

```
plan-selector:
selecting gp and decoding...
[p15 PLAN pb-read] [p15 AGENT john] [p15 OBJECT *yellow-page* (michelin-guide)]
[YES]

gp-associator:
expanding gp-tree
start-node:  s-hunger
target-node :  pb-read
expanding ...

s-hunger pb-restaurant pb-cook pb-eat d-know d-cont d-prox d-cont d-cont
d-prox d-cont pb-ask pb-read

building inference chain and backward binding propagation...
[g15 GOAL s-hunger] [g15 AGENT john]
[p16 PLAN pb-restaurant] [p16 AGENT john] [p16 OBJECT ?food] [p16 LOCATION
?restaurant]
[g17 GOAL d-know] [g17 AGENT john] [g17 OBJECT location] [g17 OBJ-ATTR
*?market* (?restaurant)]
[p18 PLAN pb-read] [p18 AGENT john] [p18 OBJECT *yellow-page* (michelin-guide)]

action-generator:
generating actions and decoding...
[ev19 STATE hungry] [ev19 AGENT john] [ev19 MODE not]
[ev20 ACT ate] [ev20 AGENT john] [ev20 OBJECT ?food] [ev20 LOCATION
?restaurant]
[ev21 STATE knew] [ev21 AGENT john] [ev21 OBJECT location] [ev21 OBJ-ATTR
?restaurant
[ev22 ACT read] [ev22 AGENT john] [ev22 OBJECT *yellow-page* (michelin-guide)]

ts-generator:
ts-generating...
john was not hungry
john ate ?food at ?restaurant
john knew ?restaurant location
john read *yellow-page* (michelin-guide)
```

In this example, DYNASTY introduced a role-binding error at the plan-selection process, and produced wrong instance (yellow-page) instead of correct instance (michelin-guide) for the variable (?guide-book). This type of instance confusion occurs because DYNASTY sometimes confuses ID patterns of different instances in the same conceptual category. When ID patterns are confused, they are not corrected through out the whole process which conforms to the previous results using ID schemes [Miikkulainen, 1990a; Miikkulainen and Dyer, in press]. However, when concept categories are confused, they can be corrected at the next module. In the above example, the misproduced ?market concept by the GP-Associator was corrected later by the Action-Generator. The instances are sometimes confused but the concept categories themselves are rarely confused because DYNASTY experiences all kinds of different instances for the same concept category during training.

5.3 Generalizations in DYNASTY

One of the advantages of DYNASTY, compared to symbolic systems, is its automatic generalization abilities. Based on the training data, DYNASTY can generalize previous experiences to new cases, which is one form of similarity-based learning. DYNASTY's generalizations are supported at three levels: (1) The DSRs support generalization at the representation level since similarly used words already have similar representations. The experiment for DSR-based generalizations are described in section 5.3.2. (2) The ID unit supports generalizations at the processing level. The IDs are set up as random during training. Every story is *new* to DYNASTY during processing because DYNASTY is trained using *only story skeletons with random IDs.* For example, DYNASTY never saw the specific story instances in the test data (used in sections 5.2.1 and 5.2.2 for the performance statistics) during the training stage. However since DYNASTY learns the identity function to propagate any ID pattern without alteration, it can propagate the specific ID pattern for the specific instance-word by interpolation. So the performance tables in section 5.2.1 and 5.2.2 are the result of generalizations to new stories. (3) Based on the statistical biases in the training data, DYNASTY can generalize the previous experiences to specific situations, and *guess the default instances* when the input stories do not provide necessary specific knowledge. This kind of generalization is called statistically-biased generalization (next section).

5.3.1 Statistically-biased generalizations: encoding experience-based goal/plan preferences/defaults

In general, it is very difficult to encode statistically-biased knowledge using symbolic rule-based systems, because symbolic rule-based knowledge must be uniformly applied to all the relevant situations. However, humans can process statistically-biased knowledge easily, such as personal preferences acquired through past experience. DYNASTY can process statistical knowledge by biasing the training data. The following statistical knowledge was embedded into the DYNASTY's goal/plan trees to test statistically-biased generalizations.

John always eats steak, while Mary always eats shrimp at the restaurant.
John always cooks fish, while Mary always cooks chicken when he/she is hungry.
John always asks a friend to know the restaurant location.
Mary always reads the Michelin-guide to know the restaurant location.
John always shops at Vons, while Mary shops at Lucky's.
John always withdraws money from BOA, while Mary from Security-Pac.
John always drives when he goes to the restaurant.
Mary always walks when she goes to the restaurant.

Note that this knowledge is purely based on statistics from past experience. There are no natural rules behind John's eating steak and Mary's eating shrimp. They are just personal preferences, and this statistical knowledge is difficult to be encoded in symbolic system's rule-bases, such as PAM [Wilensky, 1978]. Symbolic systems would require a lot of knowledge engineering with very ad-hoc rules to encode such statistical knowledge, such as:

> IF John is a diner
> THEN John eats steak at the restaurant
>
> IF John is a diner and John goes to the restaurant
> THEN John drives to the restaurant

However, when the rules are too specific, those rules become very ad-hoc, i.e. they cannot be applied to several different situations through instantiation and bindings. Connectionist systems can handle these statistical data in natural way, since connectionist systems' rule-following behaviors emerge by extracting statistical regularities underlying the training data.

Appendix D.1 shows the statically-biased goal/plan trees which encode the personal preferences about plan selection and default role fillers as described above. What happens when DYNASTY is trained with these statistical biases? Consider the following two stories (story 6 and 7).

> (story 6) John was hungry. John wanted to know the Sizzler location. John called up a friend.
> (story 7) Mary was hungry. Mary wanted to know the Sizzler location. Mary called up a friend.

When DYNASTY is trained with the goal/plan trees in appendix D.1, *DYNASTY builds up different goal/plan inference chains even though the two stories are different instantiations of the same story skeleton.* This is because DYNASTY has different default roles and planning preferences for John and Mary.

The following is an output trace for the second sentence in story 6. Note that DYNASTY infers that the food John might eat at Sizzler is **steak** because John always eats steak at that restaurant.

```
input event:  john wanted (john knew sizzler location)
st-parser:
st-parsing...
[ev38 STATE knew] [ev38 AGENT john] [ev38 OBJECT location] [ev38 OBJ-ATR
sizzler]
[ev39 STATE wanted] [ev39 AGENT john] [ev39 OBJECT ev38]
triple-encoder:
event representation formed for wanted
plan-selector:
selecting gp and decoding...
[g40 GOAL d-know] [g40 AGENT john] [g40 OBJECT location] [g40 OBJ-ATTR
sizzler] [NO]
```

```
gp-associator:
expanding gp-tree
start-node:  s-hunger
target-node :  d-know
expanding ...

s-hunger pb-restaurant pb-cook pb-eat d-know

building inference chain and backward binding propagation...
[g41 GOAL s-hunger] [g41 AGENT john] [p42 PLAN pb-restaurant] [p42 AGENT
john] [p42 OBJECT steak] [p42 LOCATION sizzler] [g43 GOAL d-know] [g43
AGENT john] [g43 OBJECT location] [g43 OBJ-ATTR sizzler]


action-generator:
generating actions and decoding...
[ev44 STATE hungry] [ev44 AGENT john] [ev44 MODE not]
[ev45 ACT ate] [ev45 AGENT john] [ev45 OBJECT steak] [ev45 LOCATION
sizzler
[ev46 STATE knew] [ev46 AGENT john] [ev46 OBJECT location] [ev46 OBJ-ATTR
sizzler]

ts-generator:
ts-generating...
john was not hungry
john ate steak at sizzler
john knew sizzler location
```

The following is trace for the third sentence in the same story (story 6). DYNASTY infers "pb-ask" plan for John because that is John's preference for the d-know goal.

```
input event:  john called up friend
st-parser:
st-parsing...
[ev49 ACT called-up] [ev49 AGENT john] [ev49 TO friend]
triple-encoder:
event representation formed for called-up
plan-selector:
selecting gp and decoding...
[g50 PLAN pb-phone] [g50 AGENT john] [g50 TO friend] [YES]


gp-associator:
expanding gp-tree
start-node:  d-know
target-node :  pb-phone
expanding ...
```

```
d-know pb-ask d-link pb-phone building inference chain and backward
binding propagation...
[g51 GOAL d-know] [g51 AGENT john] [g51 OBJECT location] [g51 OBJ-ATTR
sizzler]
[p52 PLAN pb-ask] [p52 AGENT john] [p52 OBJECT location] [p52 OBJ-ATTR
sizzler] [p52 TO friend]
[g53 GOAL d-link] [g53 AGENT john] [g53 TO friend]
[p54 PLAN pb-phone] [p54 AGENT john] [p54 TO friend]

action-generator:
generating actions and decoding...
[ev55 STATE knew] [ev55 AGENT john] [ev55 OBJECT location] [ev55 OBJ-ATTR
sizzler]
[ev56 ACT asked] [ev56 AGENT john] [ev56 OBJECT location] [ev56 OBJ-ATTR
sizzler] [ev56 TO friend]
[ev57 STATE had] [ev57 AGENT john] [ev57 OBJECT comm-link] [ev57 TO
friend]
[ev58 ACT called-up] [ev58 AGENT john] [ev58 TO friend]

ts-generator:
ts-generating...
john knew sizzler location
john asked friend about sizzler location
john had comm-link to friend
john called-up friend
```

For the story 7, DYNASTY infers a different kind of food and plan for Mary according to the given statistical biases. The following traces are for the second sentence in the Mary's story (story 7). Only the traces of GP-Associator are shown here. We can see that DYNASTY infers **shrimp** for the food that Mary ate at the Sizzler.

```
gp-associator:
expanding gp-tree
start-node:  s-hunger
target-node :  d-know
expanding ...

s-hunger pb-restaurant pb-cook pb-eat d-know

building inference chain and backward binding propagation...
[g41 GOAL s-hunger] [g41 AGENT mary] [p42 PLAN pb-restaurant] [p42 AGENT
mary] [p42 OBJECT shrimp] [p42 LOCATION sizzler] [g43 GOAL d-know] [g43
AGENT mary] [g43 OBJECT location] [g43 OBJ-ATTR sizzler]
```

The following traces are for the third sentence in story 7. The traces show that DYNASTY infers pb-read plan for d-know goal, instead of pb-ask plan. So the reason that

Mary called up a friend is different from that of story 6. For story 6, DYNASTY infers that John called up friend because he wanted to ask the Sizzler location. However, for story 7, DYNASTY infers that Mary called up friend to borrow the Michelin-guide to read it. Even though this inference chain is longer than the one for story 6, this inference is reasonable because DYNASTY knows that Mary always reads Michelin-guide to know the restaurant location.

```
gp-associator:
expanding gp-tree
start-node:  d-know
target-node :  pb-phone
expanding ...
d-know pb-read d-cont pb-shopping pb-borrow pb-grasp d-prox d-cont d-know
d-link pb-walk pb-withdraw pb-steal pb-borrow pb-read pb-phone

building inference chain and backward binding propagation...
[g51 GOAL d-know] [g51 AGENT mary] [g51 OBJECT location] [g51 OBJ-ATTR
sizzler]
[p52 PLAN pb-read] [p52 AGENT mary] [p52 OBJECT michelin-guide]
[g53 GOAL d-cont] [g53 AGENT mary] [g53 OBJECT michelin-guide]
[p54 PLAN pb-borrow] [p54 AGENT mary] [p54 OBJECT michelin-guide] [p54
FROM friend]
[g55 GOAL d-link] [g55 AGENT mary] [g55 TO friend]
[p56 PLAN pb-phone] [p56 AGENT mary] [p56 TO friend]
```

Complete I/O's for the story 6 and 7 with two more example stories are listed in appendix D.2. These examples show that DYNASTY can incorporate statistical knowledge by biasing the training data. In other words, *DYNASTY can produce different conclusions for the same input with different past experiences.* These abilities are very difficult for pure symbolic systems, such as SAM [Cullingford, 1978] and PAM [Wilensky, 1978], and such "subjective understanding" requires that the knowledge be pre-encoded by hand, as in the POLITICS system [Carbonell, 1979].

5.3.2 Generalization experiments without IDs

When we do not use IDs in the word representations, DYNASTY's generalization abilities can be measured by standard experiments [Lee et al., 1990]. Without IDs, generalizations occur solely based on the similarity properties in the event, goal, and representations, which are originated from the word level similarity properties in the DSRs. We tested DYNASTY's generalization abilities without ID-units by training DYNASTY on some portion of the stories and by testing it with the remaining portion of the stories. In the following experiment, we changed the ratio between trained and tested script-based stories. We used 2 different script-based story skeletons (restaurant and shopping script-based story skeletons in appendix B.2) and generated 16 script-based stories (8 for each script). Among these 16 stories,

Trained	Tested	E_a Scr	Correct Scr	E_a Ev	Correct Ev
12	4	0.003	100	0.021	99
8	8	0.004	100	0.022	99
4	12	0.007	100	0.023	99
2	14	0.012	100	0.044	98

Table 5.4: **Generalization performance for the full story input.** Trained (tested) designates the number of trained (tested) script-based stories. E_a Scr (E_a Ev) designates the average Euclidean distance over all the units for the plan-selector (action-generator) module. Correct Scr (Correct Ev) designates the percentage of correctly recognized scripts (events), where "correct" means that each and every unit in a script (event) representation is within 0.1 of the correct unit value (0.0 to 1.0 range). There are 10 units in the script and event representations in this experiments. No ID units are included.

Trained	Tested	E_a Scr	Correct Scr	E_a Ev	Correct Ev
12	4	0.005	100	0.021	99
8	8	0.008	100	0.021	99
4	12	0.012	100	0.023	99
2	14	0.023	100	0.067	98

Table 5.5: **Generalization performance for the partial script story input.**

we used from 2 up to 12 stories for training, and 14 down to 4 stories for testing the generalization abilities. In table 5.4, the sum of the number of stories under the first two columns are always 16, the total number of stories. The first row designates the results of having 12 stories for training, and 4 stories for testing out of total 16 stories. The last row designates the results of having 2 stories for training and 14 stories for testing. Table 5.4 shows DYNASTY's generalization ability with the full story inputs, where all the events in the script knowledge (see appendix C.3) are shown in the input stories.

Table 5.4 shows excellent generalization performance. The percentage of correctly recognized scripts and correctly produced events are not reduced when the number of trained instances decrease (the number of testing instances increase). Table 5.5 shows DYNASTY's generalization ability for partial story input, where only the events star-marked among the entire script events in the appendix C.3 appeared in the input story.

modules	1 unit	2 unit	3 unit	4 unit	5 unit
TE	74 (0.031)	69 (0.037)	65 (0.043)	61 (0.048)	54 (0.054)
PS	87 (0.059)	82 (0.064)	76 (0.069)	71 (0.072)	62 (0.080)
GP	75 (0.067)	69 (0.073)	64 (0.077)	57 (0.084)	49 (0.094)
AG	92 (0.037)	86 (0.044)	82 (0.049)	75 (0.057)	66 (0.067)

Table 5.6: **Unit damage resistance for each module.** The table shows the result of 1 to 5 unit damaged among 10 units in the DSR representations. Each number designates the correctness measure: percentage of correct outputs over all the outputs for each module. The correct output means that each unit in the output is within 0.15 value from the desired output (value range 0 to 1). The numbers in the parentheses designate the average Euclidean distance over all the units. (TE: Triple-Encoder, PS: Plan-Selector, GP: GP-Associator, AG: Action-Generator)

The generalization performance is almost the same as the full story case, since the plan-selector robustly recognizes the correct script when only partial input stories are given.

5.4 Damage resistance

5.4.1 Lesioning units

The robustness of DSR representations against damage was tested by eliminating the last n units from each word representation in the GD while DYNASTY is processing the stories. These damaged units were fixed at 0.5, which are the "don't know" values. Table 5.6 shows performance declines for each module when more and more units are damaged. The performance decline is approximately linear to the number of unit damaged. The performance decline is about 4 to 9 percent drop at each step when each of the last units is damaged. The average Euclidean distance errors increase 0.004 to 0.009 range for each unit's damage. This table is the result of processing the same 12 goal/plan-based stories as used in section 5.2.1. These stories were produced from the story skeletons in appendix B.1 by randomly instantiating the variables. The unit damage analysis programs are listed in appendix G.3.

modules	1%	2%	5%	10%
TE	80 (0.027)	75 (0.029)	62 (0.045)	40 (0.129)
PS	92 (0.058)	83 (0.060)	66 (0.079)	42 (0.105)
GP	82 (0.061)	77 (0.074)	65 (0.094)	49 (0.118)
AG	100 (0.040)	91 (0.058)	76 (0.089)	52 (0.110)

Table 5.7: **Weight damage resistance for each module.** The table shows from 1 to 10 percent of random weight damage resistances. The numbers designate the correctness measure, and the numbers in the parentheses designate the average Euclidean distance errors as before. The module name abbreviations are the same as in the table 5.6.

5.4.2 Lesioning weights

We have tested DYNASTY's robustness against weight damage by randomly killing some amount of the network's weights while DYNASTY is processing the stories. These weights were randomly fixed at 0, the middle value of the entire weight value range. Table 5.7 shows the performance decline as more and more weights are damaged for each module. The random array position of weight matrices were fixed at 0 according to the damage percentage. For example, when damage percentage is 1%, then 1 percent of weights are damaged at several positions in the weight matrices (by random number generation). Table 5.7 shows almost linear decrease in performance, and linear increase of Euclidean distance errors as more and more percentage of weights are damaged. This table is a result of processing the same 12 goal/plan-based stories as in the unit damage experiment. Weight damage analysis programs are listed in appendix G.2.

5.5 Discussion

Compared to pure symbolic systems, such as SAM [Cullingford, 1978; DeJong, 1979] and PAM [Wilensky, 1978], DYNASTY demonstrates many desirable features of distributed connectionist systems, i.e. automatic knowledge encodings by training, automatic generalization with similarity-based representations and ID units, statistically-biased generalization, fault-tolerance, and graceful performance degradation. DYNASTY is able to encode the statistical knowledge (such as personal preferences), which are very difficult to be encoded using symbolic rules, but are easy for humans to process. This is because DYNASTY's knowledge is formed gradually by extracting statistical regularities from the past experience which is represented by training data. Unlike symbolic systems, DYNASTY is also suitable for the massively parallel hardware implementations using VLSI technology [Mead, 1987].

A modular connectionist architecture with recursive, compositional distributed representations (the DSRs in DYNASTY) opens a new way to building practical symbolic/ connectionist systems that can perform fairly high-level inferencing tasks. This type of neurally inspired cognitive architecture can bridge the gap between logical/symbolic AI and the more numerical/statistical neural network field. Usually symbolic AI systems lack expandability since they are brittle and break easily with larger practical data, and symbolic systems cannot encode statistical knowledge well without manually encoding very ad-hoc rules. But in the DYNASTY case, when the system needs to process larger practical data, all it needs is to increase amount of training data. Connectionist implementation gives DYNASTY the abilities to encode fuzzy statistical knowledge as well as rule-based knowledge.

Chapter 6

Related research

Three areas of previous research influenced the DYNASTY project: symbolic systems, (2) distributed representations, and (3) distributed connectionist systems. In the symbolic systems section, we discuss pure symbolic systems which utilize important knowledge structures such as scripts, plans and goals for story understanding. In the distributed representation section, we discuss connectionist (distributed) knowledge representation schemes used to build high-level cognitive systems. In the distributed connectionist system section, we discuss connectionist implementations of high-level cognitive systems which were similar to DYNASTY at the task level.

6.1 Symbolic systems

6.1.1 Script processing systems

SAM (Script Applier Mechanism) [Cullingford, 1978; Schank and Riesbeck, 1981] is a symbolic system which can read stereotypical stories, such as:

> John went to a restaurant. He ordered a hot dog. The waiter said they didn't have any. He asked for a hamburger. When the hamburger came, it was burnt. He left the restaurant.

SAM processes stories using a knowledge structure for stereotypic action sequences called a *script* [Schank and Abelson, 1977; Dyer et al., 1987]. The above story shows SAM's deviation handling ability. SAM can infer why John left the restaurant without eating the burnt hamburger. It is difficult for DYNASTY to handle this kind of deviation since DYNASTY's restaurant plan has only the normal event sequences that occur in a restaurant.

To demonstrate its performance, SAM can do question-answering and summarization (in English and in Spanish) according to the story it read, such as:

> Q1: Did John sit down in the restaurant?
> A1: Probably.

SAM's scripts are organized as hierarchical trees. For example, the Museum-script is embedded in a more general Trip-script, and the Museum-script has a Bathroom-script and Restaurant-script as its sub-scripts. The knowledge of a Trip-script can be inherited to the Museum-script while SAM is processing the story. Currently DYNASTY does not have a script hierarchy for characteristic inheritance in SAM's sense, but can handle multiple scripts in one story easily because a script is a special complex plan to DYNASTY. DYNASTY's script (plan) knowledge is not organized in a strict hierarchy, but organized into plan-for/sub-goal relations with goal knowledge (see section 3.4.1).

SAM consists of several symbolic modules including a parser, a memory instantiator and a script applier. The parser (ELI) and memory system (PP-Memory) were adopted from the previous system MARGIE [Schank, 1975]. ELI (English Language Interpreter) [Riesbeck, 1975] was one of the first parser modules to convert English input to the language-independent representations called Conceptual Dependency (CD) [Schank, 1973]. PP-Memory is the tokenizer module [Rieger, 1975] which finds the PPs (Picture Producers: objects, e.g. human) in the CD representation and assigns tokens (instances, e.g. John) to them. The SAM's Script-Applier is the module which fetches the script data structure from the memory and make inferences to build story representations. It does this by use of symbolic pattern matching.

Script selection in SAM is performed using script-header patterns. A script-header is a collection of patterns for those events which will invoke or initiate a script. SAM defines 4 types of script-headers and they are applied according to priority rules. But sometimes, it is difficult to recognize a correct script by using just the predefined script headers, so SAM must do the backtracking to correct the mistake in recognizing scripts. For example, consider the following story:

> John went to the restaurant. He mopped the floor. He cleaned the table. He emptied out the trash can.

This story was not actually discussed in the original SAM system. However, for this story, SAM would recognize the restaurant script since the first event is a perfect locale header [Cullingford, 1978] of the restaurant script. When SAM reads the second event, it now must backtrack to select the cleaning script. DYNASTY however is trained with all the events in the input stories, not just with a few headers, to select a single script without backtracking.

Consider how DYNASTY reads the above story. When DYNASTY reads the first event, it appears to select the restaurant script like SAM does, but DYNASTY gradually changes the selected pattern to the cleaning script when it reads the next several events (see section 3.4.2 for the description of the plan-selector). In other words, DYNASTY's plan selection is not performed by a partial pattern matching like SAM, but by a holistic approach. Since DYNASTY's Plan-Selector is trained with all the events that would form a single script using a recurrent network, all the input events cooperate to gradually select one script.

FRUMP [DeJong, 1979] addressed additional problems in script recognition by using discrimination networks (D-Net) [Feigenbaum, 1963]. FRUMP is a script processing system that can read news stories taken directly off of the UPI news wire. Using sketchy scripts and solely top-down processing, FRUMP can skim newspaper stories faster than SAM. But FRUMP's D-Net-based script recognition system was keyed off the first sentence of the stories, and cannot handle the above mopping story either.

SAM and FRUMP have many problems as story understanding systems, and the most serious one is the problem of generality. Since symbolic scripts are rigidly defined according to stereotypic event sequences, SAM and FRUMP are not able to process non-stereotypical, but still understandable stories. But DYNASTY's main inference engine is goal/plan analysis module like PAM [Wilensky, 1978] so DYNASTY can handle non-stereotypical stories. Moreover, since SAM and FRUMP have been implemented as pure symbolic systems, they have many unresolved problems such as (1) smooth script recognition without backtracking and (2) difficult knowledge engineering problems in script design (e.g. what the main-conceptualization is and how many tracks the given script needs, etc.). The connectionist approach does not immediately solve all the knowledge engineering problems since the training data still must be devised manually. However connectionist approach *can reduce the difficult knowledge engineering problem* by designing system by training, not by programming. Since scripts are trained, not hand-crafted in DYNASTY, DYNASTY can reduce difficult knowledge engineering problems.

6.1.2 Goal/plan analysis system

PAM (Plan Applier Mechanism)[Wilensky, 1978] is a symbolic explanation-based story understander which can read non-stereotypical stories such as:

> John wanted money. He got a gun and walked into a liquor store. He told the owner he wanted some money. The owner gave John the money and John left.

PAM can perform question-answering based on the story contents and also can paraphrase the story from a specific character's point of view, using goal/plan analysis (e.g. story paraphrase from John's point of view) such as:

> I needed to get some dough. So I got myself this gun, and I walked down to the liquor store. I told the shopkeeper that if he didn't let me have the money then I would shoot him. So he handed it over. Then I left. [Wilensky, 1978, page 5]

Wilensky's claim is that we cannot have all the necessary scripts for all the possible stories. We need more dynamic structures to understand those non-stereotypical, but understandable stories. Besides the original goal/plan knowledge developed by Schank and his colleagues [Schank and Abelson, 1977], PAM has goal/plan interaction knowledge such as goal-conflicts and goal-subsumption for a single agent level, and goal-concord and goal-competition at the multi-agent level [Wilensky, 1978]. So PAM can handle the stories in which several agents' goals and plans are interrelated, such as:

> John wanted to watch the football game, but Mary said she was going to watch the Bolshoi ballet. John punched Mary in the mouth and put on the ball game.

In the above story, John and Mary have a goal competition. PAM can understand why John punched Mary in the mouth by using goal-competition knowledge. DYNASTY is a connectionist implementation of a *subset* of PAM, including script handling abilities. DYNASTY is restricted to the single planner oriented goal/plan inferencing and does not deal with the goal/plan interactions between single and multi-agents. To process goal/plan interactions, DYNASTY needs to be equipped with the theory of goal interactions and understanding of the story situations that the goal interactions give rise to. The goal interaction theory produces meta-level rules [Wilensky, 1983] which need another level of knowledge encodings in DYNASTY.

PAM uses both bottom-up and top-down processing in story understanding. Since PAM does not have predefined scripts, but dynamically connects the goal/plan structure to build goal/plan inference chains on the fly, PAM can handle non-stereotypical stories, which are impossible for SAM. At the implementation level, PAM uses complex, sometimes ad-hoc rules in the processing, and has difficult knowledge engineering problems such as devising efficient rules for all the knowledge that is necessary for goal/plan processing. So PAM has a scale-up problem which is common to every rule-based symbolic system. When PAM needs to process more stories, even if the stories are similar to the ones that PAM already processed, PAM still needs to be augmented with new rules for these similar stories. So it is difficult to scale up PAM's capability to process new but similar stories unless more rules are devised and added to PAM's rule-base. On the other hand, DYNASTY's goal/plan knowledge is not hand-crafted using ad-hoc rules, but are statistically learned from the goal/plan processing data. The training data itself are hand coded and the rule-based behaviors statistically emerge from these training data. As a result, when DYNASTY needs to process similar but new stories, DYNASTY can automatically generalize the old experiences to these similar stories. Moreover, DYNASTY can perform statistically-biased generalizations which are difficult to encode in PAM's rule-base. It would be too ad-hoc if PAM's rule-base would include statistical data such as personal preferences for the plan selection or for the specific roles (e.g. John always eats steak at Sizzler). However PAM can process a wider range of stories than the current implementation of DYNASTY can do, including goal/plan interactions between several agents. DYNASTY needs to be trained with the goal/plan interaction knowledge to process PAM's input stories, and it is not clear how it can be trained with such a knowledge with the current connectionist theory.

6.1.3 Other symbolic natural language systems

Symbolic natural language understanding models currently go far beyond simple script and plan applications. BORIS [Dyer, 1983] can handle stories based on affects and themes. In BORIS, thematic knowledge is organized around knowledge structures called TAUs (Thematic Abstraction Units) which model planning failures. OpEd [Alvarado et al., 1990] and CM (Correction Machine) [Quilici, 1991] can handle texts involving belief and arguments. OpEd uses AUs (Argumentation Units) to process economic editorial texts, which are complex knowledge structures such as scripts, but involving belief representations. CM does not apply argument units, but dynamically constructs argumentation chains using belief/goal/plan analysis. CM uses a plan-based approach to handle belief and argumentation, and recognizes/responds to user's plan oriented misconceptions. OCCAM [Pazzani, 1988; Pazzani and Dyer, 1989] can do EBL (Explanation-Based Learning) while it is reading texts. That is, OCCAM postulates casual relationships based on causal patterns, and can learn the new facts *without requiring a large number of examples.* Pure SBL (Similarity-Based Learning) systems, such as IPP [Lebowitz, 1980], need large number of examples to learn new facts by generalizations.

Current DC (distributed connectionist) systems do not yet attempt story understanding at this level, i.e. handling affects, belief, argumentation and EBL. Before current DC systems can get to this level, many processing and architectural issues must be resolved, including connectionist episodic memory and connectionist control implementations. Current connectionist systems, including DYNASTY, can only do SBL, and cannot do EBL, as can symbolic systems such as OCCAM.

6.2 Distributed representations

Even though nobody has presented a precise definition of what a distributed representation really is, there have been several techniques for forming distributed representations based on the following general description:

> Each entity is represented by a pattern of activity distributed over many computing elements, and each computing element is involved in representing many different entities [Hinton et al., 1986, page77].

van Gelder gives an interesting philosophical discussion regarding distributed representations in [vanGelder, 1989]. After thoroughly reviewing several schemes for forming distributed representations, he argues that the essence of a distributed representation turns out to be *semantic superposition*, i.e., multiple contents represented over the same space whether the space is over patterns of activation (e.g. coarse codings) or on the pattern of connectivity (e.g. tensor representations). Below, DYNASTY's DSR approach is compared with such previous research on forming distributed representations.

6.2.1 Microfeature-based representations

In the microfeature-based representation scheme, a connectionist knowledge engineer defines several microfeatures in a certain domain to represent an entity in a fixed vector format. Each bit corresponds to one micro-feature (or conjunction of two micro-features [McClelland and Kawamoto, 1986]), and the entity to be represented has "on" values in the bits which correspond to the microfeatures that the entity possess. Figure 6.1 shows a micro-feature based representation of the concept **milk**.

A micro-feature based representation is global, so that one representation can be used in many applications. It is a similarity-based representation, so that similar concepts have similar micro-features, thus supporting generalizations. However, there are some disadvantages to the micro-feature based technique: (1) it is difficult to determine the necessary micro-features in advance to cover every entities in the domain, and (2) there is no automatic way to learn the micro-features for an entity, so the representation must be manually encoded. For example, we cannot assign POINTEDNESS or BREAKABILITY features to **milk** in the above example. Therefore, representations are very sparse and most of the bits become "off". But DSRs are automatically learned using a set of propositions and are similarity-based without predefined microfeatures.

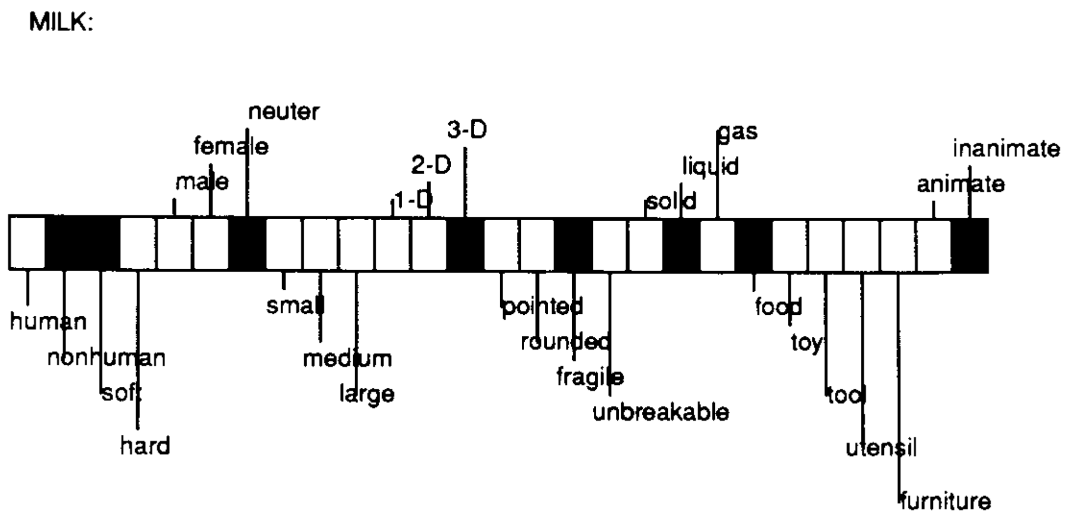


Figure 6.1: **Micro-feature based representation of concept milk.** Each micro-feature name is after [McClelland and Kawamoto, 1986].

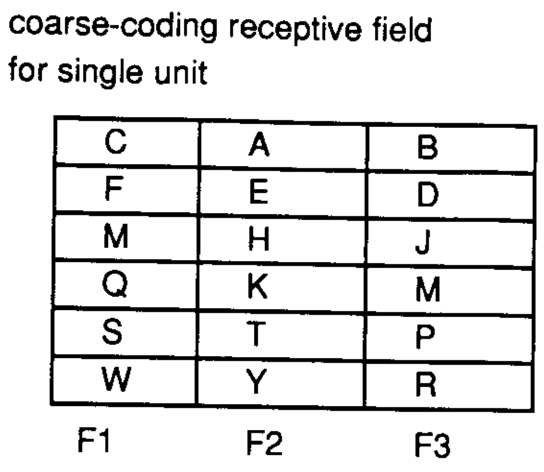
6.2.2 Coarse-codings

In the coarse-coding scheme, each unit has a carefully designed "receptive field" and every unit which has a corresponding value in the receptive field is turned on to represent an entity [Touretzky and Hinton, 1988]. For example, to represent the triple [F A B], each and every unit which has F, A, B in the first, second, third position in its receptive field should be turned on (Figure 6.2).

Carefully designed receptive fields can be used globally for several applications, and representations are coarsely similarity-based. But coarse-coding schemes have the following disadvantages: (1) much human effort is needed to design receptive fields, and the representations are not automatically learned, and (2) coarse-coded memory represents many entities in the same representation space at the same time. To access an entity in a coarse-coded system, a complex access mechanism is needed such as a pullout network [Mozer, 1984] or clause-space [Touretzky and Hinton, 1988] which uses basically a winner-take-all scheme [Feldman and Ballard, 1982] to single out an entity. In contrast, accessing DSRs needs only simple architectures such as AR (Auto-associative Recurrent) networks. However, accessing DSRs is a stack popping operation and therefore is very sequential.

6.2.3 Learning internal representations by BP

This scheme automatically learns the internal representations of an entity/relation in the hidden layer of the network by use of backpropagation (BP) [Rumelhart et al., 1986b; Hinton, 1986]. However these internal representations are locally confined to the network, so they


coarse-coding receptive field
for single unit

F1	F2	F3
C	A	B
F	E	D
M	H	J
Q	K	M
S	T	P
W	Y	R

Figure 6.2: **Randomly generated receptive field table for a working memory.** A receptive field of an unit is defined as the cross-product of the symbols in the three columns. The unit should be turned on to represent [F A B], [M H J] etc, but not turned on to represent, say, [F B B].

cannot be used in other applications. For example, in Hinton's family tree example [Hinton, 1986], the network develops two different representations for the same entity, one for the input and the other for the output layer. The developed internal representations guide the network to correctly retrieve family relationships. However, these internal representations cannot be utilized in other networks. Every network would develop its own internal representations in the hidden layer. Besides feed-forward networks, recurrent networks can also be used to develop internal representations using BP [Pollack, 1988; Elman, 1988; Jordan, 1986]. A key feature of DSRs is that the representations formed in the hidden layer are saved in a global memory called GD, so that they can be used in other applications.

6.2.4 Pollack's RAAM

Our XRAAM architectures are based on Pollack's RAAMs (recursive auto-associative memory) [Pollack, 1988] that is an implementation of Hinton's reduced description idea [Hinton, 1988]. RAAM is a PDP architecture which can devise compositional, similarity-based, and recursive PDP representations. The resulting RDR (Recursive Distributed Representations)[Pollack, 1990] can encode the variable-sized recursive and sequential data structures, such as trees and stacks in fixed resource systems. It has been argued in [Pollack, 1989] that RDRs can form a bridge between the data structures necessary for high-level cognitive tasks and the associative, pattern recognition machinery provided by neural networks, since they combine aspects of features, pointers and structures. Figure 6.3 shows the RAAM architecture used to encode stacks, which is actually a combination of two single layer feed-forward networks.

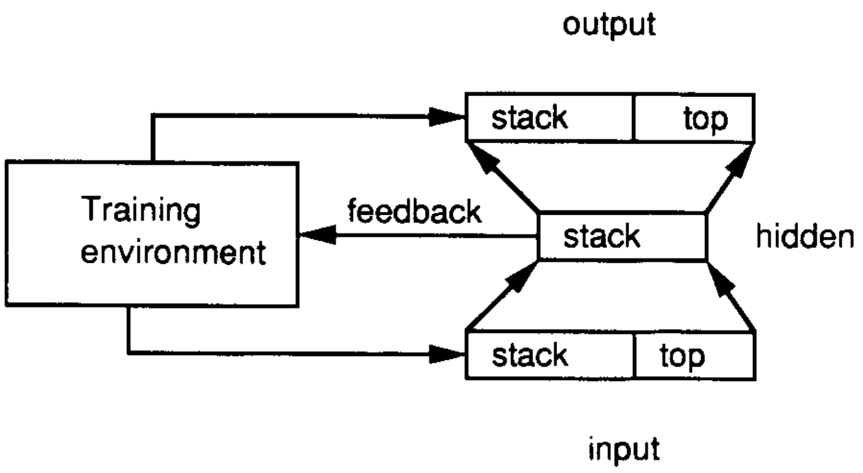


Figure 6.3: **Recursive auto-associative memory** The memory develops compositional distributed representations as the outputs of the hidden layer. These representations are part of the training environment, which therefore evolves with the weights in the network. The lower part of the networks learns to encode these representations while the upper part learns to decode them. The encoder and decoder will ultimately be used as separate mechanisms.

A RAAM can be used to encode parse trees so that it can be applied for syntactic-level processing [Chalmers, 1990]. RAAMs, however, lack an external storage for each representation formed, and thus the resulting RDRs are not global. Pollack's RAAM architecture also lacks semantics at the word level in its representations. Current RAAM's applications have been restricted to syntax processing; there are no conceptual level applications of RAAMs. But DSRs are learned using propositions in XRAAM, so they have word-level semantics and can be applied to conceptual information processing as has been done in DYNASTY. What is needed for RDRs is another level of RAAM style architecture to develop semantic level representations which have all the strong features of RDRs. This approach was taken in the DSR learning architecture.

6.2.5 Miikkulainen's FGREP

FGREP (Forming Global Representations using Extended backPropagation) [Miikkulainen and Dyer, 1988] is a mechanism to develop global distributed representations using the same kinds of symbol recirculation ideas used in the XRAAM architecture. FGREP's representations are optimal for their processing tasks because the system learns the representations at the same time it is trained to perform those tasks. The basic idea of FGREP is to extend the BP error signal to an additional input layer during the backpropagation learning to modify the input representations as if they were weights (Figure 6.4).

The advantage of FGREP representations are that they are optimal for the given tasks and have similarity properties, i.e., the representations which are used similarly in the tasks

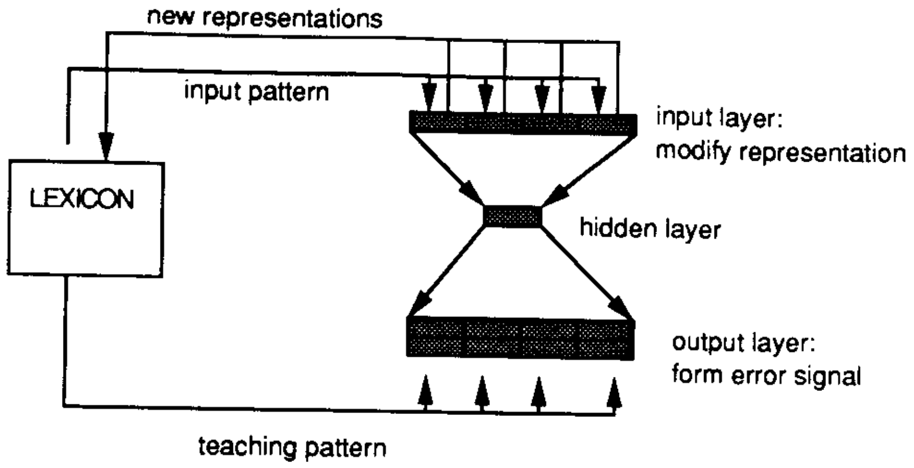


Figure 6.4: **Basic FGREP architecture.** The system consists of a three-layer BP network and an external global lexicon containing the input/output representations. At the end of each BP cycle, the current input representations, as an extra layer of weights, feed into the input layer, and are modified according to the BP extended error signal. These new representations are loaded back into the lexicon, replacing the old representations of the words used.

end up being similar to each other. However, FGREP representations are developed during specific task processing, so performance becomes poor when FGREP is applied to unrelated tasks. When we combine the architectures for different tasks, the single resulting FGREP representations can be developed during the combined processing tasks, so it can provide optimal performances for several different tasks [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press]. DSRs are, however, learned independent from any particular processing task, so the representations should be useful in any task requiring access to the propositional content of word meanings.

6.3 Distributed connectionist systems

This section compares DYNASTY with previous distributed connectionist systems that perform high-level reasoning, such as story understanding.

6.3.1 DISPAR

DISPAR (DIStributed PARaphraser) [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press] is a PDP level system which reads partial script-based stories and paraphrases them as causally complete output stories using FGREP representations [Miikkulainen and Dyer, 1988]. DISPAR's domain is restricted to script processing while DYNASTY concentrates on goal/plan analysis (including script processing). After paraphrasing the stories, DISPAR ends up developing new representations which are optimal for the paraphrasing tasks. DISPAR's representations are developed on-line while it is processing some specific applications. This is different from DYNASTY's DSRs because DSRs are developed off-line independent of any specific application. DISPAR uses a global lexicon which is the same as our global-dictionary, but DISPAR's global lexicon contains not only word concepts but also script and role representations [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press].

DISPAR consists of 4 different modules of HR (Hetero-associative Recurrent) networks: sentence-parser, story-parser, story-generator and sentence-generator. Each module performs simple pattern transformations. The sentence-parser analyzes the sentences into the case-role assignment forms [McClelland and Kawamoto, 1986]. The story-parser builds up script representations from the case-role assignment forms. The other two modules do the exact reverse transformations: from script representations to the case-role assignment forms, and finally to the output sentences. The main architectural difference between DISPAR and DYNASTY is that DYNASTY uses recursively encoded fixed length representations for events and scripts while DISPAR uses flat representations which have separate banks for the case-roles in the representations. Therefore all the event/script representations in DYNASTY have the same number of units as the word representations. However DISPAR has at least 6 times more units in the event representations and 7 times more units in the script representations than the word representations [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press]. This fixed length encoding of high-level event/script representations enable DYNASTY to handle embedded sentences (see section 3.3.3) which cannot be processed in DISPAR since DISPAR's high-level representations cannot be compressed into a fixed length.

The FGREP representations developed in DISPAR are too similar to each other when two words are used similarly in the tasks because there is no way to control the similarity of two words except through the usages in the tasks. DYNASTY also suffers from the same kind of too-similarity problem. Theoretically, DYNASTY's DSRs can be varied by adding/deleting more propositions in the proposition space (see result in section 5.1), but in practice it is very difficult to find the correct set of propositions for the applications. To solve this too-similarity problem, Miikkulainen and Dyer introduced in DISPAR a cloning mechanism to develop instance representations (e.g. John) from the generic concepts (e.g. human) by attaching random, fixed identification (ID) bits to the FGREP representations, and DYNASTY adopts this ID technique (see section 2.9). But the bad thing is that it turns out that 90% of the time is actually used to learn the ID parts of the representations since BP is very inefficient in learning to copy the static patterns to different places [Miikkulainen and Dyer, 1989]. However, Miikkulainen and Dyer [in press] have shown that an exponentially increasing number of ID patterns can be learned by DISPAR with only a polynomial amount of effort. This result is important, because both DISPAR and DYNASTY use IDs to handle the variable binding problem in distributed connectionist systems. While the distributed, similarity-based part of a word representation allows each system to generalize, the ID part gets passed on unaltered and thus allows unique bindings to be propagated.

Since DISPAR's domain is only script-based story processing, DISPAR does not need backward binding propagation like DYNASTY. This is because the selected script structure has the same variable sets as input stories. By the same reason, DISPAR does not need unification matching process either. When input stories do not provide instances for a certain variable, DISPAR guesses plausible bindings using statistically-biases generalizations. DISPAR can automatically learn the representations of function words such as articles, prepositions and conjunctives, even for punctuation, because FGREP is operating on-line while processing the stories. Every function word has its own role in the story processing, even the period. That is why DISPAR can form representations for function words. However, DYNASTY uses off-line DSR-Learner to automatically learn the representations of words using semantic case-role structures. Since the function words do not have necessary semantic case-role structures, DYNASTY cannot automatically learn the representations of function words. As a result, DYNASTY ignores the function words during parsing, while DISPAR does not. However, DSRs for the word semantics has many desirable properties for symbol processing as described in section 2.2.

6.3.2 DISCERN

DISCERN (DIstributed SCript processing and Episodic memoRy Network) [Miikkulainen, 1990a] is an extension to DISPAR. The domain remains the same (script-based story processing), but DISCERN includes episodic and semantic memory components. DISCERN's episodic memory utilizes Kohonen's feature maps [Kohonen, 1984] which have been modified to store hierarchical script representations. DISCERN's semantic memory (global lexicon) also consists of double feature maps which are formed and connected by Hebbian associative training. DISCERN also has a question-answering subsystem which is a slight modification of the original DISPAR networks. Miikkulainen's main concern in extending the DISPAR system is to explore the role of self-organizing memory using Kohonen's feature maps while DYNASTY concentrates on the extension of application domains to cover dynamic goal/plan analysis as well as static script processing.

6.3.3 DCPS

DCPS (Distributed Connectionist Production System) [Touretzky and Hinton, 1988] is a connectionist rule interpreter for a restricted class of production rules based on coarse-coded distributed representations and the Boltzmann machine learning algorithm. In DCPS, the authors implemented pattern matching and variable-binding problems in a connectionist framework. The condition part of each production rule consists of two triples of terms, which are either grounded such as [A B C] or of the form [x A B] where x is a variable and A, B, C are constants. Only two triples are allowed in the condition part of each rule, and only one variable is allowed in each rule. Moreover the variable position is restricted to the first position of each triple. For example, DCPS can interpret rule such as:

```
if (x A A) (x B B)
then add(G A B), delete(x A A), delete (x B B)
```

Figure 6.5 shows DCPS's top-level architecture. The working memory of DCPS is a set of

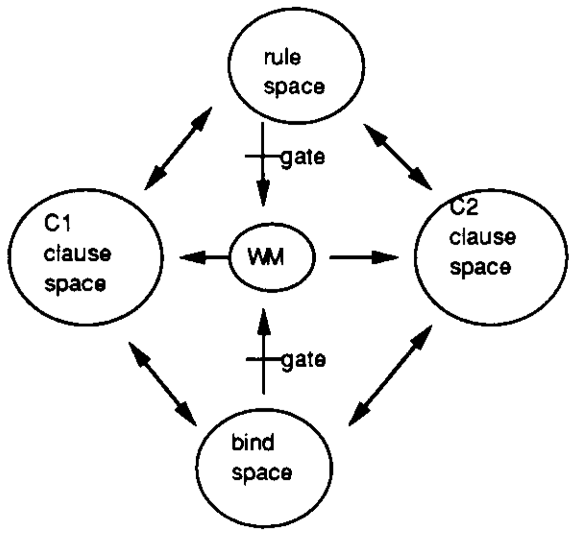


Figure 6.5: **DCPS top-level architecture**

coarse-coded binary state units which represent a set of triples. To find out which condition part of a production rule matches the working memory, a special set of units called the *clause-space* are introduced: one for the first and one for the second triple in the condition part of a production rule. For the variable binding, another set of coarse-coded units called the bind-space is used. In the bind-space, a binding unit representing the constant, say "A", is connected to all units in the clause-space which represent triples whose first element is "A". In other words, *all the possible bindings must already be statically represented in the memory*, which is a different approach from DYNASTY or DISCERN, where the bindings are propagated through the network using ID units.

DCPS is successful for an extremely limited form of rules, and it is not obvious how the technique can be applied if the position of the variables in the rule are changed and/or if the working memory also contains variables. Moreover, their coarse-coded representations require a large amount of human effort, and complex access mechanisms such as clause-spaces. In contrast, DSRs in DYNASTY are automatically learned from a set of propositions and need relatively simple access mechanisms, such as XRAAMs and the GD.

6.3.4 CRAM

CRAM [Dolan, 1989] is a distributed connectionist system which is able to read single paragraph, fable-like stories, and either give a thematically relevant summary or generate planning advice for a character in the story. CRAM is implemented using special connectionist networks called tensor manipulation networks, where the operation of the network is interpreted as manipulations of high rank tensors (generalized vector outer products)

[Dolan and Smolensky, 1989; Smolensky, 1987b]. The operations on tensors are interpreted as operations on symbol structures in the same spirit of DYNASTY in which the associative operations on the triples are interpreted as operations on symbol structures. CRAM and DYNASTY both exploit functional design approaches, called symbolic neuro-engineering [Dyer, 1990b] in which we define functional modules first, and then replace them with connectionist architectures.

CRAM's role-binding architecture uses conjunctive-coding which stores several compositions of [schema role filler] triples in one 3-D cube, later allowing the system to retrieve fillers when the schemata and roles are given. For example, when the long-term memory stores the [restaurant DINER ?person] triple in a rank-3 tensor representation in a 3-D cube, the input triple [restaurant DINER John] in the short-term memory can be matched to the long-term memory triple. The binding of John to ?person can be propagated to the other triples such as [restaurant PAYER John] since the DINER should be the PAYER in the restaurant. All these processes are implemented via a complex architecture between short-term memory and long-term memory using rank-4 exploded bindings [Dolan, 1989].

CRAM uses carefully designed, almost orthogonal micro-features to obtain reasonable performance in the tensor binding/unbinding process [Smolensky, 1987b]. Since tensor manipulation networks cannot handle non-orthogonal similarity-based representations without running into cross-talk, CRAM actually has to have additional circuits to reduce cross-talk for satisfactory performance, even with the micro-feature representations. Actually CRAM could adopt DYNASTY's DSRs as a filler representations in the [schema role filler] triples and make tensor manipulation network work as long as the schema and role representations are kept orthogonal to each other in the triples.

6.3.5 DUCS

DUCS [Touretzky and Geva, 1988] is a distributed connectionist schema processing system which emphasizes a concept and role inheritance mechanism in a connectionist framework. Unlike other distributed connectionist schema processing systems (such as CRAM), DUCS uses a micro-feature-based distributed representations *for both slot names and slot fillers.* DUCS can encode fine semantic distinctions as subtle variations on the canonical pattern for a slot. For example, in the bird schema listed below, if we ask about the NOSE of the bird instead of the BEAK, DUCS can still answer the question even though the bird does not have NOSE because the slot representation of BEAK is similar to that of NOSE.

(bird NAME tweety)
(bird CLASS robin)
(bird BEAK red)
(bird CAN-FLY yes)

DUCS's schema is mainly focused on the Minsky style frames [Minsky, 1981] which is different from DYNASTY's schema since DYNASTY uses semantic case-role based schema. Minsky style frames can have more general slot names since it was developed to mainly describe visual objects. For example, in the above bird schema, the slot itself is a concept which can be encoded as similarity-based representations. However, DYNASTY's slot (case-role) must be orthogonally represented to provide the necessary structural difference to the learned DSRs (see figure 2.2 in section 2.5).

Chapter 7

Current status, limitations and future work

7.1 Current status

The code for DYNASTY's training/performance and the various analysis routines were developed on a HP9000/350 workstation with 16M memory, using standard C. The amount of C code for the full DYNASTY system, including various training and analysis programs, is about 3000 lines (excluding duplicate codes). Currently DYNASTY has been trained on the 4 script-based story skeletons (appendix B.1) and on the 6 goal/plan-based story skeletons (appendix B.2). The training of DSR-Learner took about 9 days with 150 epochs, 300 cycles of BP per one epoch. The Triple-Encoder took about 5 days with same epochs and cycles. The other modules took from 1 to 4 days with 9000 epochs, 1 cycle of BP per one epoch. The epoch designates the period for training the whole data set, while a cycle designates the period for training one input/teaching-input pair. The above time is actual computing time including data loading and snapshot backup, not CPU time.

The ID units for each variable are set up as random between 0 and 1. There are 2 ID units, thus allowing maximum of 4 unique instances for each variable (only using binary values for ID). However only 2 instances for each variable are actually used in performance. As a result, DYNASTY can process up to a maximum of 32 script-based stories because the average number of variables in one story skeleton is 3 (i.e. $2^3 \times 4$). For the same reason, DYNASTY can process up to a maximum of 24 goal/plan-based stories because the average number of variables in one story skeleton is 2 (i.e. $2^2 \times 6$).

7.2 Limitations and their resolution

DYNASTY has several limitations for the complete story understanding system. Part of the limitations come from the limitation of current connectionist technology, and part of them are due to DYNASTY's main research goal, that is, to explore connectionism as a new paradigm for the script/goal/plan analysis and story understanding. DYNASTY is not designed to process many distinct sources of knowledge, such as BORIS [Dyer, 1983]. Below we describe some limitations of DSRs and the current DYNASTY implementation, and discuss how these limitations can be fixed in the future. For some of these limitations, the long term future research directions will be discussed in the next section.

(1) *The sequentiality of DSR encoding/decoding*: The DSR learning process is highly sequential at the knowledge level [Sumida and Dyer, 1989]. In the concept and proposition encoding cycle, the triples are fed to the network sequentially, one at a time, and the order of the sequence is fixed with the order of propositions provided as the training data (i.e. *the order must always be same*). Hence the decoding is also sequential in the reverse order

of encoding. The fixed proposition sequence is essential for the correct decoding process, and is a characteristic of RAAM [Pollack, 1990] style networks. The sequentially of DSR encoding/decoding is due to its stack-like properties at the symbol level. For example, in a symbolic stack, each element is entered into the stack sequentially, and popped out of the stack in the reverse order (Last-In-First-Out order). So in the stack, all the elements before A should be accessed when we access the element A. So sequentiality is an inherent property of stack structures for correct data accessing. This sequentiality is also ubiquitous in symbolic AI applications, such as when traversing LISP P-lists.

(2) *Instance similarity vs binding performance*: The DSR encoding process forms the representations of words according to their usage in the restricted proposition space. Therefore, if two instances of the same script roles or variables are used exactly same in the proposition space, the two representations become identical. There is no way to distinguish the two instances from their usage. Clearly we need sensory-level information to distinguish the same categorical concepts, and attaching random IDs is a first approach to this sensory grounding [Harnad, 1989; Miikkulainen, 1990a]. Without IDs, the binding cannot be propagated through the network, since the network cannot distinguish different instances of the same categories.

The alternate way of solving the instance similarity problem without IDs is by adding/ deleting more propositions in the training data to distinguish the various instances for the same concept category. But in practice, it is not easy to find the correct number of propositions to make **John** and **Jack** similar for generalization as a **person** concept but still distinct enough for good binding propagation performance. Adjusting the proposition space for distinct instance representations was examined in our previous model [Lee et al., 1990].

(3) *BP-based global-dictionary implementation*: Currently DYNASTY's GD is implemented with two BP-based HF (Hetero-associative Feed-forward) networks. This implementation, while it is simple, has one major limitation as a connectionist lexicon network, which is that BP-based networks cannot handle many-to-many or one-to-many mappings. Since BP basically implements functions from one domain to another domain, the single element in the input domain cannot be mapped to more than one element in the output domain. Why is this a problem in a connectionist lexicon model? The basic problem is that this kind of network cannot handle the case of multiple meanings (ambiguous words). For example, the word "bat" has two meanings such as "baseball-bat" or "live-bat". When the BP-based ASCII-to-DSR network is fed the input "bat", it can only produce a blending of two DSRs for "baseball-bat" and "live-bat", but humans can always retrieve one or the other concept according to the context. While sometimes this kind of blending is necessary as a composite representation, we need to single out one concept representation in a lexicon model. In the multiple lexical realization case, the problem is equally severe when two almost identical DSRs need to be translated into different lexical forms. For example, the DSR-to-ASCII network only produces a mixed blending ASCII code of "John" and "Jack" when the two DSRs are almost the same. Kohonen's topological feature map appears to be a good candidate as a self-organizing network that can do many-to-many mappings [Kohonen, 1984], and such a distributed lexicon model, based on the Kohonen's network, has been implemented in DISCERN [Miikkulainen, 1990b].

(4) *Symbolic working memory*: DYNASTY's working memory is currently implemented as a symbolic hash-table for searching expanded goal/plan trees using linked list and pointer following, and DYNASTY currently has no connectionist episodic memory model. Although all high-level representations including events, scripts and goals/plans are automatically formed using the Triple-Encoder, there is no episodic memory mechanism to store these representations by single trial learning and to retrieve the representations later to do the high-level processing such as question-answering.

(5) *Symbolic copy operation during backward binding propagation*: DYNASTY's forward binding propagation is performed in a connectionist way using ID propagation. However, the backward binding propagation cannot use ID propagation because the GP-Associator module is trained in the forward search direction from the stored goals/plans to the current input. Connectionist backward binding propagations could be obtained if we were to employ bigger knowledge structures which contain every role that might appear in a single input story. However, this is impossible in goal/plan analysis since the several goals/plans need to be dynamically connected to process a single story. A more plausible way of resolving this is to train the GP-Associator in both forward and backward search directions using independent set of weights or to find a single set of weights which can do the bidirectional associations [Kosko, 1987].

(6) *Symbolic controls for interfacing modules*: DYNASTY's performance is achieved by connecting each module via passing one module's output to another's input. Currently, DYNASTY performs all kinds of data passing and conversions (for interfacing) using symbolic controls. In other words, DYNASTY is a simulated connectionist machine using a von Neumann computer with algorithmic control. To build the real machine using neural hardware, these symbolic controls must be replaced by neurally plausible signal passing and conversion through neural pathways. Nothing very complicated is involved in this control except the format conversion between event-triples and case-role assignment forms (for the ST-Parser and TS-Generator). Most of the controls for data passing between modules can be implemented by training other networks which can open the gateways using multiplicative connections [Rumelhart et al., 1986a; Pollack, 1987]. However, some of the high-level monitoring process such as random ID generation still needs symbolic controls, and neural implementation of this high-level monitoring process is an open issue in current connectionist theory.

(7) *Limitations on natural language parsing and generation*: DYNASTY's ST-Parser and TS-Generator is by no means a complete natural language parser and generator. ST-Parser cannot handle any complex syntactic structure such as relative clauses without preprocessing. The relative clauses (embedded propositions) in natural language input must currently be marked in DYNASTY by using parentheses, so that ST-Parser knows whatever propositions inside parentheses must be processed first. Building a complete parser that can handle complex syntax using neural networks is a separate research issue by itself [Fanty, 1986]. ST-Parser ignores function words such as conjunctives, prepositions, articles, etc. The phrases (idioms) are considered to be a single word. However, function words can be processed during ST-parsing by assigning some distributed representations to them, and incorporating their representations in the ST-Parser training data. By this method, the TS-Generator generates some of the function words to make the output inference chains readable. But since DSRs cannot be formed for the function words in the current DYNASTY, a random

distributed representation must be assigned to each function word, which is useless during parsing. The TS-Generator also has many limitations. The TS-Generator just translates from the triples to natural language by placing each word in its proper position and adding some function words. The TS-Generator lacks numerous forms of knowledge for natural language generation, such as knowledge for proper word selection, proper morphological form selection, and discourse analysis for topic focusing [McKeown, 1985].

(8) *Limitations on script/goal/plan processing*: DYNASTY has several limitations in its script/goal/plan processing capabilities. For example, DYNASTY cannot handle embedded scripts, multiple scripts and script deviations using solely a script application mechanism. Instead, DYNASTY switches to a plan application mechanism to handle these kinds of special script application problems. However, DYNASTY must be told which story is script-based or goal/plan-based by the user. DYNASTY cannot yet decide on its own whether to use a script or plan application mechanism solely based on the input story. Symbolic systems such as BORIS [Dyer, 1983] can do this. However, building a connectionist story understanding system as powerful as BORIS is a long term research goal. DYNASTY's plan understanding abilities are also restricted compared to other symbolic systems, such as PAM [Wilensky, 1978]. For example, DYNASTY can handle only stories that have a single planning agent, and cannot handle goal/plan interactions between several planners. Understanding goal/plan interactions requires meta-level rules [Wilensky, 1983] about planning, and DYNASTY does not achieve this level yet.

(9) *Random representations for function words*: The DSR scheme is good at representing word semantics in terms of propositions. In the DSR scheme, a word representation is formed based on the semantic case-role structures of the word. However, function words such as articles (e.g. "the") and prepositions (e.g. "at") do not have semantic case-role structures. As a result, the DSR scheme is not useful for automatically forming representations for these function words. Currently, DYNASTY employs random distributed representations for the function words. Another related problem for the DSR scheme is its inability to handle phrases (idioms) such as "look at" vs. "look to". DYNASTY considers these phrases to be a single word.

There is a method that could be employed to form function word representations using DSR scheme, that is, using syntactic categories instead of semantic case-roles as slot names (in the triples). For example, consider forming representations of an preposition "at" using the following propositions:

p1: John ate steak **at** the Sizzler.
p2: John threw a stone **at** Mary.
p3: John died **at** the church.

The proposition-triples are formed in order to be loaded into the proposition-encoding network, such as:

[p1 AGENT John] [p1 ACT ate] [p1 OBJECT apple] [p1 LOCATION Sizzler]
[p2 AGENT John] [p2 ACT threw] [p2 OBJECT stone] [p2 TO Mary]
[p3 AGENT John] [p3 ACT died] [p3 LOCATION church]

Now we can form the concept-triples for the preposition "at" which are loaded into the concept-encoding network by using these 3 propositions.

[at LOC-PREP p1] [at DIR-PREP p2] [at LOC-PREP p3]

where LOC-PREP (location preposition) and DIR-PREP (direction preposition) are new case-role names which designate syntactic/semantic categories of the preposition. The real problem is to devise these proper case-roles which can categorize every function word into appropriate syntactic/semantic groups. Neither the original thematic case-role theory [Fillmore, 1968] nor the several extended semantic case-role theories [Schank, 1973; Bruce, 1975] attempted to incorporate function words into the case-role structures. Once the function word representations are formed, each word in phrases can be processed separately, thus giving DYNASTY the ability to handle phrases in the input story.

7.3 Future research directions

All of the current limitations of DYNASTY supply directions for the future work and some of them were discussed in the previous section. In this section, we will discuss long term future research needed to overcome the current limitations and to give more potential to the DYNASTY approach for story understanding. DYNASTY inspires several types of future research directions, concerning (1) incorporating sensory information in the DSRs, (2) developing a connectionist episodic memory model, (3) modeling a question-answering process, and (4) developing DSR-based machine translator.

7.3.1 Incorporating sensory information in the proposition space

DSRs in DYNASTY have limited semantics from the provided proposition space. When two words behave (i.e. are used) exactly the same in the proposition space, then the DSRs developed for these word concepts become exactly identical. While this semantic interpretation is valuable on the grounds that language semantics comes from usage and similarity supports generalizations, this semantics has limitations in that it cannot support binding propagation which is essential for high-level reasoning systems. DYNASTY's solution for binding performance is to attach random ID units to the DSRs, modeled after [Miikkulainen and Dyer, 1989; Miikkulainen, 1990a]. However, it is better for DSRs to learn sensory information as well as conceptual information from the proposition space itself, rather than to attach external random bits to the learned DSRs. We believe that word semantics for objects consists of at least two components: a categorical part and a sensory part [Harnad, 1989]. The categorical part holds the general conceptual information of the words; the sensory part contains a sensory level identifying information. So the representation of **John** should have categorical information such as **person**, as well as sensory information, such as visual appearance, voices, etc. DYNASTY is provided with the proposition space that is extracted from the sentences in the script/goal/plan-based story understanding domain. So the developed DSRs cannot encode the sensory information since each representation comes from the propositional encodings of the words. While this proposition space is reasonable for DYNASTY's current task, the DSRs only reflect the usages of the words in the sentences (propositions) based on semantic case-role theory [Fillmore, 1968; Schank, 1973]. In the future, the DSRs should have sensory information in the representations so that two words can have different representations even if they are in the exactly same categories. The visual micro-world representation in [Allen, 1988] is the first rudimentary step in this direction.

7.3.2 Connectionist episodic memory model

DYNASTY currently has no connectionist episodic long-term memory model for storing, indexing and retrieving the developed script/goal/plan representations. An episodic memory would provide pattern completion and associative retrieval functions to DYNASTY, retrieving complete script/goal/plan representations from partial representations. DYNASTY finesses connectionist episodic memory issues by employing a symbolic hash-table to store the inferred goal/plan representations.

BP-based networks have inherent limitations as episodic memories since BP-based networks are plagued from unlearning properties [Hinton et al., 1984]. To learn a new pattern association, a BP-based network must be trained again on the entire training data, including the old training set, lest the network forget previously learned pattern associations [Dyer, 1990b]. But this deficiency is not an inherent limitation of connectionist networks, but a limitations of the BP learning algorithm. Recently, a slightly modified BP has been used to model connectionist episodic memory that can reduce the retroactive interferences [Kortge, 1990], and also a hierarchical Kohonen feature map has been used to model script-based episodic memory [Miikkulainen, 1990c]. But more research must be devoted to developing connectionist episodic memories that are comparable to symbolic episodic memory models, such as that described in [Kolodner, 1983].

7.3.3 Modeling question-answering in DYNASTY

Building a question-answering architecture as part of DYNASTY is not difficult for the single plan (script) processing case because there is no need to find the relevant plans for the input questions. Figure 7.1 shows a possible question-answering architecture for single plan processing in DYNASTY.

This architecture can be used in script-based question answering tasks such as:

```
Q:Who entered the Chart-House?
A: John entered the Chart-House
Q: What did John order?
A: John ordered steak
```

The question is parsed in the same manner as input stories except that the *don't know* patterns are used for the unspecified case-roles.

[ev30 ACT went], [ev30 AGENT ?], [ev30 TO chart-house]
[ev31 ACT order], [ev31 AGENT John], [ev31 OBJECT ?]

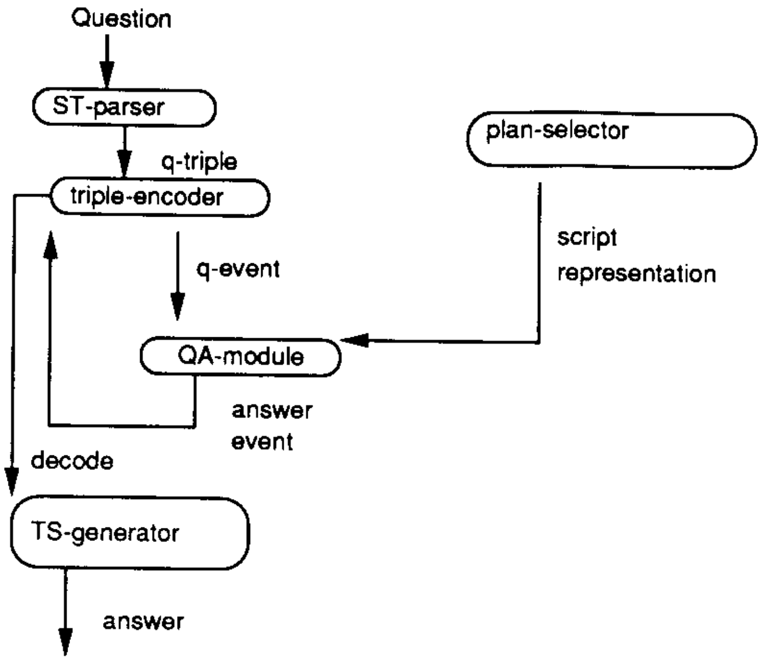


Figure 7.1: **Question-answering architecture for script processing.**

while the ? designates the don't know patterns (all 0.5). The Triple-Encoder constructs the question-event representations, and the Plan-Selector provides the necessary plan (script) representations to the QA-network. The QA-network is a plain 3 layer BP network which associates the input question-event plus script representations with the output answer-event representations. These answer-event representations are decoded using the Triple-Encoder, and fed to the TS-Generator to produce answers. Goal/plan-based question-answering is more difficult since DYNASTY does not have an associative memory model to select the relevant goals/plans for the input questions. The selected goals/plans are supposed to provide the necessary contexts to the QA-network.

7.3.4 DSR-based machine translation

We claim that DSRs can serve as building blocks for connectionist language understanding/processing systems that need to access the propositional contents of word meanings. One example of another task domain is machine translation. Although [Allen, 1987] tried to do English/Spanish translation using a BP network, his system just transforms each English/Spanish sentence as one whole pattern, and does not provide any solutions based on the distributed representations.

Below is a possible architecture for a DSR-based English/Korean translation system. The global dictionary is bilingual; it has three entries: English-word, Korean-word, and DSRs.

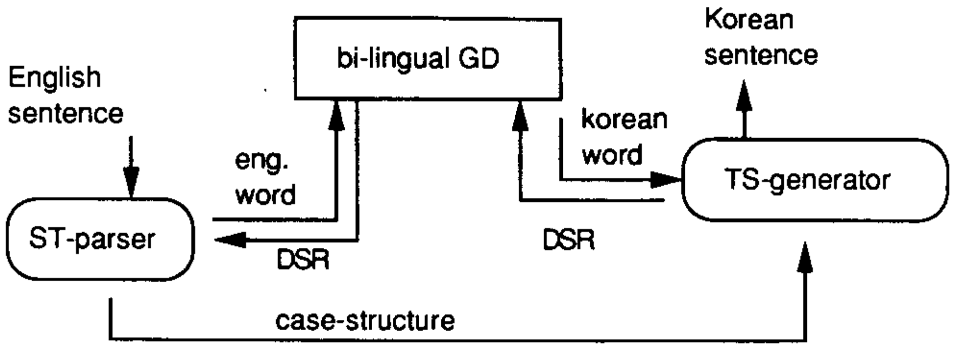


Figure 7.2: **DSR-based machine translation architecture.**

Suppose the system was to translate this sentence into Korean:

```
John went to the Chart-House.
```

Each word in an English sentence is changed into its DSR, and fed to the English-parser word by word. The output case-role representations are independent of any language, and are a meaning representation of the sentence [Schank, 1973]. These case-role representations are fed to the Korean-generator to produce the DSR for each word, thus forming a Korean language word sequence, and the DSRs are converted to Korean words using the bilingual global dictionary. The output shows the Korean word order for this input sentence:

```
John Chart-House E Gan-da
(John Chart-House to went).
```

The above scheme is modeled after *symbolic* dictionary-based translation approaches. However we do not deal with "meaningless" symbol-to-symbol translation here, but deal with meaningful representations (DSRs) of the words. The network can extract translation regularities from the word representations. For example, ACT words (e.g. went) will always be at the end of the Korean translations.

Chapter 8

Summary and conclusions

8.1 Summary

DYNASTY is a natural language understanding model built from modular distributed connectionist architectures with symbolic controls. DYNASTY's tasks are (1) formation of word representations, and (2) script/goal/plan-based story understanding. DYNASTY adopts a symbolic neuro-engineering approach; that is, the global system architecture is modeled after symbolic systems while each component is implemented using various connectionist architectures.

DYNASTY's word representations are stored in a *global dictionary*, and the modules communicate using these representations. The word representations are automatically learned from the domain knowledge which is presented to the system in the form of propositions. The semantic part of the word representations are called *distributed semantic representations* (DSRs) and have many desirable properties to support high-level symbolic reasoning in a connectionist framework: automaticity, portability, structure-encoding ability, and similarity-based representations. The DSRs are learned using two XRAAM architectures. Each word meaning is encoded using the related propositions and each proposition is encoded using the involved words. The words and propositions gradually form their own representations while influencing each other through the network. DYNASTY's vocabulary in the global-dictionary can be extended by cloning new instances of the items, i.e. generating a number of items with the same properties but with distinct identities by attaching a unique ID-representation [Miikkulainen and Dyer, 1989; Miikkulainen and Dyer, in press] to the DSRs.

DYNASTY's script/goal/plan analysis architecture is modular, and consists of different connectionist architectures, namely, auto-associative recurrent (AR), hetero-associative recurrent (HR), and hetero-associative feed-forward (HF) architectures. Each module in DYNASTY is trained separately and in parallel within the data dependencies. The training data for each module is specified using the DYNASTY data-specification language. The DYNASTY architecture supports both goal/plan analysis and script processing since a script is a special type of plan in DYNASTY. The DYNASTY architecture is divided into 3 big models according to the 3 major subtasks for the script/goal/plan-based story understanding. The 3 models are (1) the representation subsystem, (2) the goal/plan analysis subsystem, and (3) the linguistic subsystem. The representation subsystem consists of a global-dictionary and triple-encoder and provides word and high-level distributed representations for the remaining modules. The goal/plan analysis subsystem consists of a plan-selector, a goal/plan relation associator and an action generator. This subsystem constructs a goal/plan inference chain from the event sequence. The linguistic subsystem consists of a parser and a generator which converts the natural language sentence to the case-role triples, and vice versa.

DYNASTY's processing model is modeled after symbolic script and goal/plan processing systems. At the task level, DYNASTY operates like a symbolic system, but each component actually operates with backpropagation-based networks. DYNASTY's goal/plan knowledge is encoded in OR/AND trees, which specify the relations between goals and plans. DYNASTY searches the goal/plan space by expanding goal/plan trees until it finds connections from the existing knowledge to the new input knowledge. Role-bindings and binding propagations are performed automatically when the ID units in the word representations are propagated through the network. During the goal/plan tree expanding process, bindings are also propagated backward using pattern-copy operations through the goal/plan inference chain to handle the bindings that cannot be performed using forward binding propagation alone. DYNASTY uses a symbolic working memory to store the developed goal/plan inference chains.

DSRs show similarity properties according to the provided proposition space. When the proposition space gets richer, the clusters of similar words become more natural. DYNASTY's generalization performance shows an average of 84.7% correctness for new stories. DYNASTY shows fault tolerance and graceful degradation performance when some of the weights and units are removed. DYNASTY also can process statistically-biased stories, and produce different inference chains according to a specific planner and situation. These statistically-biased generalizations are very difficult for pure symbolic systems, such as SAM [Cullingford, 1978] and PAM [Wilensky, 1978].

8.2 Conclusion

There are two major goals of the DYNASTY project: (1) to develop distributed connectionist representations to support high-level symbol processing, and (2) to implement a script/goal/plan-based natural language understanding system in a connectionist framework.

For the first goal, we have presented distributed semantic representations (DSRs) which serve as an adequate foundation for constructing and manipulating conceptual knowledge in a connectionist architecture. We have presented an architecture, based on XRAAMs, to automatically learn DSRs using the domain knowledge. Our experiments indicate that DSRs show many desirable properties that can be used in high-level cognitive systems in a symbolic application domain. We have shown that DSRs can serve as building blocks in constructing connectionist cognitive architectures by developing the DYNASTY system.

For the second goal, we have developed DYNASTY, a modular connectionist system for the high-level inferencing which can (1) automatically form distributed representations of words, events and scripts/goals/plans from input sentences in the domain of script/goal-based story understanding, (2) generate complete script event sequences from fragmentary inputs, and (3) generate goal/plan inference chains from input actions. Moreover, the high-level representations (DSRs of concepts, events, scripts, and goals/plans) formed contain constituent structure that can be decoded and extracted, making the semantic content available for multiple tasks.

DYNASTY provides many desirable features of connectionist implementation, which are not possible in pure symbolic systems, such as automatic and statistically-biased generalizations, automatic knowledge encodings through training, fault tolerance and graceful performance degradation. From DYNASTY, we have learned that a single gigantic BP network has obvious limitations in modeling complex natural language understanding tasks, but modular network architectures with suitable distributed representations and symbolic AI constraints can perform the desired tasks.

Bibliography

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, (9):147–169.

Allen, R. B. (1987). Several studies on natural language and back-propagation. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*. IEEE.

Allen, R. B. (1988). Sequential connectionist networks for answering simple questions about a microworld. In *Proceedings of the Tenth Annual Cognitive Science Society Conference*, pages 489–495, Hillsdale, NJ. Lawrence Erlbaum Associates.

Alvarado, S. J. (1990). *Understanding editorial text: a computer model of argument comprehension*. Kluwer Academic Publishers.

Alvarado, S. J., Dyer, M. G., and Flowers, M. (1990). Argument representations for editorial text. *Knowledge-based systems*, 3(2):87–107.

Anderson, J. A. and Rosenfeld, E., editors (1988). *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA.

Arens, Y. (1986). *Cluster: An approach to contextual language understanding*. PhD thesis, UC Berkeley. Report No. UCB/CSD 86/293.

Bower, G. H., Black, J. B., and Turner, T. J. (1979). Scripts in memory for text. *Cognitive Psychology*, (11):177–220.

Bruce, B. (1975). Case systems for natural language. *Artificial Intelligence*, 6:327–360.

Carbonell, J. G. (1979). *Subjective understanding: computer models of belief systems*. PhD thesis, Computer Science Department, Yale University. Research Report No. 150.

Chalmers, D. J. (1990). Syntatic transformations on distributed representations. *Connection Science*, 2(1;2).

Charniak, E. (1972). Toward a model of children's story comprehension. Technical Report AI-TR-266, MIT AI lab.

Charniak, E. (1978). On the use of framed knowledge in language comprehension. *Artificial Intelligence*, 11(3).

Chun, H. W. and Mimo, A. (1987). A model of schema selection using marker passing and connectionist spreading activation. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, pages 887–896, Hillsdale, NJ. Lawrence Erlbaum Associates.

Cottrell, G. W. and Small, S. L. (1983). A connectionist scheme for modelling word sense disambiguation. *Cognition and Brain Theory*, 6(1):89–120.

Cullingford, R. E. (1978). *Script Application: Computer Understanding of Newspaper Stories.* PhD thesis, Department of Computer Science, Yale University. Technical Report 116.

DeJong, G. F. (1979). *Skimming Stories in Real Time: An Experiment in Integrated Understanding.* PhD thesis, Department of Computer Science, Yale University. Research Report 158.

Dolan, C. P. (1989). *Tensor Manipulation Networks: Connectionist and Symbolic Approaches to Comprehension, Learning and Planning.* PhD thesis, Computer Science Department, UCLA.

Dolan, C. P. and Dyer, M. G. (1987). Symbolic schemata, role binding, and the evolution of structure in connectionist memories. In *Proceedings of the IEEE First Annual International Conference on Neural Networks*, pages Vol. 2, 287–298. IEEE.

Dolan, C. P. and Smolensky, P. (1989). Implementing a connectionist production system using tensor products. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 265–272, Los Altos, CA. Morgan Kaufmann Publishers, Inc.

Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension.* MIT Press, Cambridge, MA.

Dyer, M. G. (1990a). Distributed symbol formation and processing in connectionist networks. *Journal of experimental and theoretical artificial intelligence*, 2(4):313–345.

Dyer, M. G. (1990b). Symbolic NeuroEngineering for natural language processing: A multilevel research approach. In Barnden, J. and Pollack, J., editors, *Advances in Connectionist and Neural Computation Theory.* Ablex Publ. (in press).

Dyer, M. G., Cullingford, R. E., and Alvarado, S. (1987). Scripts. In Shapiro, S. C., editor, *Encyclopedia of Artificial Intelligence*, pages 980–994. John Wiley and Sons, New York.

Dyer, M. G., Flowers, M., and Wang, A. (1988). Weight matrix = pattern of activation: encoding semantic networks as distributed representations in DUAL, a PDP architecture. Technical Report UCLA-AI-88-5, Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles.

Dyer, M. G., Flowers, M., and Wang, A. (in press). Distributed symbol discovery through symbol recirculation: Toward natural language processing in distributed connectionist networks. In Reilly, R. and Sharkey, N., editors, *Connectionist approaches to natural language understanding.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Elman, J. L. (1988). Finding structure in time. Technical Report 8801, Center for Research in Language, University of California, San Diego.

Fanty, M. (1986). Context-free parsing with connectionist networks. In Denker, J., editor, *AIP conference proceedings 151: neural Networks for computing*, pages 140–145. American Institute of Physics, New York.

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. In Feigenbaum, E. A. and Feldman, J. A., editors, *Computers and Thought*. McGraw-Hill, New York.

Feldman, J. A. (1986). Neural representation of conceptual knowledge. Technical Report TR 189, Department of Computer Science, Univ. of Rochester, N.Y.

Feldman, J. A. and Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science*, (6):205–254.

Fillmore, C. J. (1968). The case for case. In Bach, E. and Harms, R. T., editors, *Universals in Linguistic Theory*, pages 1–90. Holt, Rinehart and Winston, New York.

Fodor, J. and Pylyshyn, Z. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, (28):3–71.

Golden, R. M. (1986). Representing causal schemata in connectionist systems. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, pages 13–22, Hillsdale, NJ. Lawrence Erlbaum Associates.

Grossberg, S., editor (1988). *Neural Networks and Natural Intelligence*. MIT Press, Cambridge, MA.

Hanson, S. J. and Kegl, J. (1987). Parsnip: A connectionist network that learns natural language grammar from exposure to natural language sentences. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, pages 106–119, Hillsdale, NJ. Lawrence Erlbaum Associates.

Harnad, S. (1989). The symbol grounding problem. In *CNLS Conference on Emergent Computation*.

Hartigan, J. A. (1975). *Clustering Algorithms*. John Wiley and Sons, N.Y.

Hendler, J. (1988). *Integrating marker-passing and problem solving: A spreading activation approach to improved choice in planning*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, pages 2–12, Hillsdale, NJ. Lawrence Erlbaum Associates.

Hinton, G. E. (1987). Connectionist learning procedures. Technical Report CMU-CS-87-115, Computer Science Department, Carnegie-Mellon University.

Hinton, G. E. (1988). Representing part-whole hierarchies in connectionist networks. In *Proceedings of the Tenth Annual Cognitive Science Society Conference*, pages 48–54, Hillsdale, NJ. Lawrence Erlbaum Associates.

Hinton, G. E. and Anderson, J. A., editors (1981). *Parallel Models of Associative Memory.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*, pages 77–109. MIT Press, Cambridge, MA.

Hinton, G. E., Sejnowski, T. J., and Ackley, D. H. (1984). Boltzmann machines: Constraint satisfaction networks that learn. Technical Report CMU-CS-84-119, Computer Science Department, Carnegie-Mellon University.

Jordan, M. I. (1986). Serial order: A parallel, distributed processing approach. Technical Report 8604, Institute for Cognitive Science, University of California, San Diego.

Kohonen, T. (1984). *Self-Organization and Associative Memory*, chapter 9. Springer-Verlag, Berlin; New York.

Kolodner, J. L. (1980). Retrieval and organizational strategies in conceptual memory: a computer model. Technical Report TR187, Yale University, Department of Computer Science. Ph D Thesis.

Kolodner, J. L. (1983). Reconstructive memory: A computer model. *Cognitive Science*, (7):281–328.

Kortge, C. A. (1990). Episodic memory in connectinist networks. In *Proceedings of the Twelfth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.

Kosko, B. (1987). Competitive adaptive bidirectional associative memories. In *Proceedings of the IEEE First Annual International Conference on Neural Networks.* IEEE.

Kuhn, T. S. (1970). *The Structure of Scientific Revolutions.* International Encyclopedia of Unified Science, Vol. 2, No. 2. 2, The University of Chicago Press, Chicago, second, enlarged edition.

Lange, T. E. and Dyer, M. G. (1989). High-level inferencing in a connectionist network. *Connection Science*, 1(2).

Lebowitz, M. (1980). Generalization and memory in an integrated understanding system. Technical Report TR186, Yale University, Department of Computer Science. Ph. D Thesis.

Lee, G., Flowers, M., and Dyer, M. G. (1989). A symbolic /connectionist script applier mechanism. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, pages 714–721, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.

Lee, G., Flowers, M., and Dyer, M. G. (1990). Learning distributed representations of conceptual knowledge and their application to script-based story processing. *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research*, 2(4):313–346.

Lehnert, W. G. (1978). *The Process of Question Answering.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Lehnert, W. G. (1988). Knowledge-based natural language understanding. In Shorbe, H. E., editor, *Exploring artificial intelligence.* Morgan Kaufmann Publishing, Inc.

McClelland, J. L. and Kawamoto, A. H. (1986). Mechanisms of sentence processing: Assigning roles to constituents. In McClelland, J. L. and Rumelhart, D. E., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume II: Psychological and Biological Models*, pages 272–326. MIT Press, Cambridge, MA.

McKeown, K. (1985). *Text generation: using discourse strategies and focus constraints to generate natural language text.* Cambridge University Press.

Mead, C. (1987). Silicon models of neural computation. In *Proceedings of the IEEE First Annual International Conference on Neural Networks.* IEEE.

Miikkulainen, R. (1990a). *DISCERN: A distributed artificial neural network model of script processing and memory.* PhD thesis, UCLA, Computer Science.

Miikkulainen, R. (1990b). A distributed feature map model of the lexicon. In *Proceedings of the Twelfth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.

Miikkulainen, R. (1990c). Script recognition with hierarchical feature maps. *Connection Science.* (in press).

Miikkulainen, R. and Dyer, M. G. (1988). Forming global representations with extended backpropagation. In *Proceedings of the IEEE Second Annual International Conference on Neural Networks*, pages Vol. 1, 285–292. IEEE.

Miikkulainen, R. and Dyer, M. G. (1989). A modular neural network architecture for sequential paraphrasing of script-based stories. In *Proceedings of the International Joint Conference on Neural Networks.* IEEE.

Miikkulainen, R. and Dyer, M. G. (in press). Natural language processing with modular PDP networks and distributed lexicon. *Cognitive Science.*

Minsky, M. (1981). A framework for representing knowledge. In Haugeland, J., editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence.* MIT Press, Cambridge, MA.

Minsky, M. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, expanded edition.

Mozer, M. C. (1984). The perception of multiple objects: a parallel, distributed processing approach. unpublished thesis proposal, Institute for cognitive science, UCSD.

Newell, A. (1980). Physical symbol systems. *cognitive science*, 4:135–183.

Norvig, P. (1986). *A unified theory of inference for text understanding.* PhD thesis, Computer Science Division, University of California, Berkeley.

Pazzani, M. J. (1988). *Learning causal relationships: an integration of empirical and explanation-based learning methods.* PhD thesis, Computer Science Department, University of California, Los Angeles.

Pazzani, M. J. and Dyer, M. G. (1989). Memory organization and explanation-based learning. *International journal of expert systems: research and applications*, 2(3):331–358.

Pinker, S. and Prince, A. (1988). On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28.

Pollack, J. B. (1987). Cascaded back-propagation on dynamic connectionist networks. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.

Pollack, J. B. (1988). Recursive auto-associative memory: Devising compositional distributed representations. Technical Report MCCS-88-124, Computing Research Laboratory, New Mexico State University.

Pollack, J. B. (1989). Implications of recursive distributed representations. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems I.* Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence.* (in press).

Quilici, A. E. (1991). *The correction machine: a computer model of recognizing and processing belief justification in argumentative dialogs.* PhD thesis, Computer Science Department, University of California, Los Angeles.

Rieger, C. (1975). Conceptual memory. In Schank, R., editor, *Conceptual information processing.* North-Holland, Amsterdam.

Riesbeck, C. (1975). Conceptual analysis. In Schank, R., editor, *Conceptual information processing.* North-Holland, Amsterdam.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, Washington.

Rumelhart, D. E. (1975). Notes on a schema for stories. In Bobrow, D. and Collins, A., editors, *Representation and understanding*, pages 211–236. NewYork: Academic press.

Rumelhart, D. E., Hinton, G. E., and McClelland, J. L. (1986a). A general framework for parallel distributed processing. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations.* MIT Press, Cambridge, MA.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*, pages 318–362. MIT Press, Cambridge, MA.

Rumelhart, D. E. and McClelland, J. L. (1987). Learning the past tenses of English verbs: Implicit rules or parallel distributed processing. In MacWhinney, B., editor, *Mechanisms of Language Acquisition.* Erlbaum, Hillsdale, NJ.

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986c). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, Cambridge, MA.

Schank, R. (1975). *Concetual information processing.* North Holland, Amsterdam.

Schank, R. (1982). *Dynamic memory: A theory of reminding and learning in computers and people.* Cambridge university press.

Schank, R. and Abelson, R. (1977). *Scripts, Plans, Goals, and Understanding - An Inquiry into Human Knowledge Structures.* The Artificial Intelligence Series. Lawrence Erlbaum Associates, Hillsdale, NJ.

Schank, R. and Riesbeck, C. K., editors (1981). *Inside Computer Understanding.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Schank, R. C. (1973). Identification of conceptualization underlying natural language. In Schank, R. and Colby, R., editors, *Computer models of thought and language*, pages 187–248. W. H. Freeman and Company.

Schmidt, C. F., Sridharan, N. S., and Goodson, J. L. (1978). The plan recognition problem: An intersection of psychology and artificial intelligence. *Artificial Intelligence.* special issue on applications in the sciences and medicine.

Sejnowski, T. J. and Rosenberg, C. R. (1986). Nettalk: A parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, Electrical Engineering and Computer Science, Johns Hopkins University.

Sharkey, N. E., Sutcliffe, R., and Wobcke, W. (1986). Mixing binary and continuous schemes for knowledge access. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.

Shastri, L. (1987). A connectionist encoding of semantic networks. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Cognitive Science Society, Lawrence Erlbaum Associates.

Smolensky, P. (1987a). A method for connectionist variable binding. Technical Report CU-CS-356-87, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder.

Smolensky, P. (1987b). On variable binding and the representation of symbolic structures in connectionist systems. Technical Report CU-CS-355-87, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder.

Smolensky, P. (1988). On the proper treatment of connectionism. *The Behavioral and Brain Sciences*, 11(1):1–74.

St. John, M. F. and McClelland, J. L. (1989). Applying contextual constraints in sentence comprehension. In Touretzky, D. S., Hinton, G. E., and Sejnowski, T. J., editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 338–346, Los Altos, CA. Morgan Kaufmann Publishers, Inc.

Sumida, R. A. and Dyer, M. G. (1989). Storing and generalizing multiple instances while maintaining knowledge-level parallelism. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Los Altos, CA. Morgan Kaufmann Publishers, Inc.

Sumida, R. A., Dyer, M. G., and Flowers, M. (1988). Integrating marker passing and connectionism for handling conceptual and structural ambiguities. In *Proceedings of the Tenth Annual Cognitive Science Society Conference*, Hillsdale, NJ. Lawrence Erlbaum Associates.

Touretzky, D. S. and Geva, S. A. (1988). A distributed connectionist representation for concept structures. In *Proceedings of the Tenth Annual Cognitive Science Society Conference*, pages 155–163, Hillsdale, NJ. Lawrence Erlbaum Associates.

Touretzky, D. S. and Hinton, G. E. (1988). A distributed connectionist production system. *Cognitive Science*, 12(3):423–436.

vanGelder, T. (1989). *Distributed representation.* PhD thesis, Graduate faculty of art and science, University of Pittsburgh.

Waltz, D. L. and Pollack, J. B. (1985). Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, (9):51–74.

Wilensky, R. (1978). *Understanding goal-based stories.* PhD thesis, Computer science department, Yale university.

Wilensky, R. (1981). Micro pam. In Schank, R. and Riesbeck, C., editors, *Inside computer understanding*, pages 180–196. Hillsdale, NJ.

Wilensky, R. (1983). *Planning and understanding: a computational approach to human reasoning.* Addison-Wesley.

Winograd, T. (1972). *Understanding natural language.* Academic Press, New York.

Zernik, U. (1987). *Strategies of Language Acquisition: Learning Phrases from Examples in Context.* PhD thesis, Computer Science Department, University of California, Los Angeles. Technical Report UCLA-AI-87-1.

APPENDIX A

DYNASTY proposition space

This chapter lists DYNASTY's proposition space which is used to develop DSRs for whole vocabularies listed in section 3.3.2. The propositions are listed in two different triple forms (concept-triples and event-triples) which are machine readable. Concept-triples are the training data for the concept-encoding network and event-triples are the training data for the proposition-encoding network in the DSR-Learner module (both described in section 2.5). The propositions listed here are the proposition generators. The variables (with ?-mark) in those proposition generators are replaced with the instances (listed in section A.1) during DSR learning. The variables are randomly replaced with one of the two instances at every epoch, and are maintained the same during one epoch.

A.1 Variables and instances

```
?person: John, Mary
?cook-utensil: pan, micro-wave
?raw-food: fish, chicken
?guide-book: michelin-guide, yellow-page
?restaurant: sizzler, mamasion
?food: steak, lobster
?market: vons, lucky
?restaurant: sizzler, mamasion
?class: computer, biology
?professor: simth, alan
?doctor: dr-kim, dr-park
?hospital: ucla-hospital, usc-hospital
```

A.2 Proposition-triples

Proposition-triples for proposition-encoding network

```
ev1 state hungry;  ev1 agent ?person
ev2 act borrowed; ev2 agent ?person; ev2 object ?cook-utensil; ev2 from friend
ev3 act entered; ev3 agent ?person; ev3 location kitchen
ev4 act picked-up; ev4 agent ?person; ev4 object ?raw-food
ev5 act read; ev5 agent ?person; ev5 object ?guide-book
ev6 act borrowed; ev6 agent ?person; ev6 object money; ev6 from friend
ev7 act got-into; ev7 agent ?person; ev7 location car
ev8 act walked; ev8 agent ?person; ev8 to ?restaurant
ev9 act stole; ev9 agent ?person; ev9 object money; ev9 from waiter
ev10 act got; ev10 agent ?person; ev10 object ?food
ev11 state wanted; ev11 agent ?person; ev11 object ev11
ev12 act asked; ev12 agent ?person; ev12 object location;
ev12 obj-attr ?market; ev12 to friend
ev13 act went; ev13 agent ?person; ev13 to bank
ev14 act went; ev14 agent ?person; ev14 to ?restaurant
ev15 state wanted; ev15 agent ?person; ev15 object ev15
ev16 act stopped; ev16 agent ?person; ev16 location ?gas-station
ev17 state knew; ev17 agent ?person; ev17 object location;
ev17 obj-attr ?restaurant
```

```
ev18 state wanted; ev18 agent ?person; ev18 object ev18
ev19 act picked-up; ev19 agent ?person; ev19 object ?guide-book
ev20 act got; ev20 agent ?person; ev20 object ?guide-book
ev21 state wanted; ev21 agent ?person; ev21 object ev21
ev22 state knew; ev22 agent ?person; ev22 object location; ev22 obj-attr ?market
ev23 state wanted; ev23 agent ?person; ev23 object ev23
ev24 act called-up; ev24 agent ?person; ev24 to friend
ev25 state wanted; ev25 agent ?person; ev25 object ev25
ev26 act walked; ev26 agent ?person; ev26 to pay-phone
ev27 state had; ev27 agent ?person; ev27 object money; ev27 mode not
ev28 act went; ev28 agent ?person; ev28 to bank
ev29 state wanted; ev29 agent ?person; ev29 object ev29
ev30 act walked; ev30 agent ?person; ev30 to ?market
ev31 act got; ev31 agent ?person; ev31 object ?raw-food
ev32 state wanted; ev32 agent ?person; ev32 object ev32
ev33 act asked; ev33 agent ?person; ev33 object location;
ev33 obj-attr ?restaurant; ev33 to friend
ev34 act drove; ev34 agent ?person; ev34 to ?restaurant
ev35 act borrowed; ev35 agent ?person; ev35 object coin; ev35 from waiter
ev36 act bought; ev36 agent ?person; ev36 object ?raw-food; ev36 location ?market
ev37 act got; ev37 agent ?person; ev37 object ?cook-utensil
ev38 state wanted; ev38 agent ?person; ev38 object ev38
ev39 act drove; ev39 agent ?person; ev39 to ?market
ev40 act entered; ev40 agent ?person; ev40 location ?restaurant
ev41 act seated; ev41 agent waiter; ev41 object ?person
ev42 act brought; ev42 agent waiter; ev42 object menu
ev43 act read; ev43 agent ?person; ev43 object menu
ev44 act ordered; ev44 agent ?person; ev44 object ?food
ev45 act ate; ev45 agent ?person; ev45 object ?food
ev46 act paid; ev46 agent ?person; ev46 object bill
ev47 act left; ev47 agent ?person; ev47 object tip
ev48 act left-for; ev48 agent ?person; ev48 from ?restaurant; ev48 to home
ev49 state needed; ev49 agent ?person; ev49 object ?food
ev50 act entered; ev50 agent ?person; ev50 location ?market
ev51 act got; ev51 agent ?person; ev51 object cart
ev52 act picked-up; ev52 agent ?person; ev52 object ?food
ev53 act waited; ev53 agent ?person; ev53 location line
ev54 act paid; ev54 agent ?person; ev54 to cashier
ev55 act left-for; ev55 agent ?person; ev55 from ?market; ev55 to home
ev56 state knew; ev56 agent ?person; ev56 object ?class
ev57 state wanted; ev57 agent ?person; ev57 object ev57
ev58 act entered; ev58 agent ?person; ev58 to ?class
ev59 act sat-down; ev59 agent ?person; ev59 location seat
ev60 act took-out; ev60 agent ?person; ev60 object notebook
ev61 act listened; ev61 agent ?person; ev61 from ?professor
ev62 act took-note; ev62 agent ?person
ev63 act checked; ev63 agent ?person; ev63 object time
ev64 act left-for; ev64 agent ?person; ev64 from ?class; ev64 to home
ev65 state sick; ev65 agent ?person
ev66 act went; ev66 agent ?person; ev66 to hospital
ev67 act checked-in; ev67 agent ?person; ev67 to receptionist
ev68 act sat-down; ev68 agent ?person; ev68 location seat
ev69 act read; ev69 agent ?person; ev69 object ?magazine
ev70 act entered; ev70 agent ?person; ev70 location exam-room
ev71 act tested; ev71 agent nurse; ev71 object ?person
ev72 act examined; ev72 agent ?doctor; ev72 object ?person
ev73 act left-for; ev73 agent ?person; ev73 from hospital; ev73 to home
ev74 state hungry; ev74 agent ?person; ev74 mode not
ev75 act cooked; ev75 agent ?person; ev75 object ?raw-food; ev75 location kitchen
ev76 state had; ev76 agent ?person; ev76 object ?cook-utensil
ev77 act borrowed; ev77 agent ?person; ev77 object ?cook-utensil; ev77 from friend
ev78 state inside; ev78 agent ?person; ev78 location kitchen
ev79 state had; ev79 agent ?person; ev79 object ?raw-food
ev80 act picked-up; ev80 agent ?person; ev80 object ?raw-food
ev81 act ate; ev81 agent ?person; ev81 object ?food; ev81 location ?restaurant
ev82 state knew; ev82 agent ?person; ev82 object location;
ev82 obj-attr ?restaurant
ev83 act read; ev83 agent ?person; ev83 object ?guide-book
ev84 state had; ev84 agent ?person; ev84 object money
ev85 act borrowed; ev85 agent ?person; ev85 object money; ev85 from friend
ev86 state inside; ev86 agent ?person; ev86 location ?restaurant
ev87 act drove; ev87 agent ?person; ev87 to ?restaurant
```

```
ev88 state inside; ev88 agent ?person; ev88 location car
ev89 act walked; ev89 agent ?person; ev89 to ?restaurant
ev90 act stole; ev90 agent ?person; ev90 object money; ev90 from waiter
ev91 state had; ev91 agent ?person; ev91 object ?food
ev92 act bought; ev92 agent ?person; ev92 object ?food; ev92 location ?market
ev93 state knew; ev93 agent ?person; ev93 object location; ev93 obj-attr ?market
ev94 act asked; ev94 agent ?person; ev94 object location;
ev94 obj-attr ?market; ev94 to friend
ev95 act withdrew; ev95 agent ?person; ev95 object money; ev95 from bank
ev96 state inside; ev96 agent ?person; ev96 location bank
ev97 state had; ev97 agent ?person; ev97 object gasoline
ev98 act bought; ev98 agent ?person; ev98 object gasoline;
ev98 location ?gas-station
ev99 state inside; ev99 agent ?person; ev99 location ?gas-station
ev100 state had; ev100 agent ?person; ev100 object ?guide-book
ev101 act picked-up; ev101 agent ?person; ev101 object ?guide-book
ev102 act bought; ev102 agent ?person; ev102 object ?guide-book;
ev102 location ?market
ev103 state had; ev103 agent ?person; ev103 object comm-link; ev103 to friend
ev104 act called-up; ev104 agent ?person; ev104 to friend
ev105 state near; ev105 agent ?person; ev105 location pay-phone
ev106 act walked; ev106 agent ?person; ev106 to pay-phone
ev107 act drove; ev107 agent ?person; ev107 to bank
ev108 act ate; ev108 agent ?person; ev108 object ?food
ev109 state inside; ev109 agent ?person; ev109 location ?market
ev110 act walked; ev110 agent ?person; ev110 to ?market
ev111 act bought; ev111 agent ?person; ev111 object ?raw-food;
ev111 location ?market
ev112 act asked; ev112 agent ?person; ev112 object location;
ev112 obj-attr ?restaurant; ev112 to friend
ev113 state had; ev113 agent ?person; ev113 object coin
ev114 act borrowed; ev114 agent ?person; ev114 object coin; ev114 from waiter
ev115 act borrowed; ev115 agent ?person; ev115 object money; ev115 from friend
ev116 act drove; ev116 agent ?person; ev116 to ?market
ev117 state knew; ev117 agent ?person; ev117 object ?class
ev118 state sick; ev118 agent ?person; ev118 mode not
g119 goal s-hunger; g119 agent ?person
p120 plan pb-restaurant; p120 agent ?person; p120 object ?food;
p120 location ?restaurant
p121 plan pb-cook; p121 agent ?person;  p121 object ?raw-food
p122 plan pb-eat; p122 agent ?person; p122 object ?food
g123 goal d-know; g123 agent ?person; g123 object location; g123 obj-attr ?market
g124 goal d-know; g124 agent ?person; g124 object location;
g124 obj-attr ?restaurant
g125 goal d-know; g125 agent ?person; g125 object phone-number;
g125 obj-attr friend
g126 goal d-know; g126 agent ?person; g126 object ?class
p127 plan pb-ask; p127 agent ?person; p127 object location;
p127 obj-attr ?market; p127 to friend
p128 plan pb-ask; p128 agent ?person; p128 object location;
p128 obj-attr ?restaurant; p128 to friend
p129 plan pb-ask; p129 agent ?person; p129 object phone-number;
p129 obj-attr friend
p130 plan pb-read; p130 agent ?person; p130 object ?guide-book
p131 plan pb-lecture; p131 agent ?person; p131 object ?class; p131 from ?professor
g132 goal d-cont; g132 agent ?person; g132 object coin
g133 goal d-cont; g133 agent ?person; g133 object ?guide-book
g134 goal d-cont; g134 agent ?person; g134 object gasoline
g135 goal d-cont; g135 agent ?person; g135 object ?food
g136 goal d-cont; g136 agent ?person; g136 object ?raw-food
g137 goal d-cont; g137 agent ?person; g137 object ?cook-utensil
p138 plan pb-shopping; p138 agent ?person; p138 object ?guide-book;
p138 location ?market
p139 plan pb-shopping; p139 agent ?person; p139 object gasoline;
p139 location ?gas-station
p140 plan pb-shopping; p140 agent ?person; p140 object ?food;
p140 location ?market
p141 plan pb-shopping; p141 agent ?person; p141 object ?raw-food;
p141 location ?market
p142 plan pb-shopping; p142 agent ?person; p142 object ?cook-utensil;
p142 location ?market
p143 plan pb-borrow; p143 agent ?person; p143 object coin; p143 from waiter
```

```
p144 plan pb-borrow; p144 agent ?person; p144 object ?guide-book; p144 from friend
p145 plan pb-borrow; p145 agent ?person; p145 object ?cook-utensil;
p145 from friend
p146 plan pb-grasp; p146 agent ?person; p146 object coin
p147 plan pb-grasp; p147 agent ?person; p147 object ?guide-book
p148 plan pb-grasp; p148 agent ?person; p148 object ?food
p149 plan pb-grasp; p149 agent ?person; p149 object ?raw-food
p150 plan pb-grasp; p150 agent ?person; p150 object ?cook-utensil
g151 goal d-cont; g151 agent ?person; g151 object money
p152 plan pb-withdraw; p152 agent ?person; p152 object money; p152 from bank
p153 plan pb-steal; p153 agent ?person; p153 object money; p153 from waiter
p154 plan pb-borrow; p154 agent ?person; p154 object money; p154 from friend
g155 goal d-prox; g155 agent ?person; g155 object pay-phone
g156 goal d-prox; g156 agent ?person; g156 location ?restaurant
g157 goal d-prox; g157 agent ?person; g157 location bank
g158 goal d-prox; g158 agent ?person; g158 location ?market
g159 goal d-prox; g159 agent ?person; g159 location car
p160 plan pb-walk; p160 agent ?person; p160 to pay-phone
p161 plan pb-walk; p161 agent ?person; p161 to ?restaurant
p162 plan pb-walk; p162 agent ?person; p162 to bank
p163 plan pb-walk; p163 agent ?person; p163 to ?market
p164 plan pb-walk; p164 agent ?person; p164 to car
p165 plan pb-drive; p165 agent ?person; p165 to ?restaurant
p166 plan pb-drive; p166 agent ?person; p166 to bank
p167 plan pb-drive; p167 agent ?person; p167 to ?market
g168 goal d-link; g168 agent ?person; g168 to friend
p169 plan pb-phone; p169 agent ?person; p169 to friend
p170 plan pb-letter; p170 agent ?person; p170 to friend
g171 goal p-health; g171 agent ?person
p172 plan pb-doctor; p172 agent ?person; p172 to ?doctor
```

A.3 Concept-triples

Concept-triples for concept-encoding network

```
?class from ev64; ?class object ev117; ?class object ev56;
?class object g126; ?class object p131; ?class to ev58
?cook-utensil object ev2; ?cook-utensil object ev37; ?cook-utensil object ev76;
?cook-utensil object ev77; ?cook-utensil object g137; ?cook-utensil object p142;
?cook-utensil object p145; ?cook-utensil object p150
?doctor agent ev72; ?doctor to p172
?food object ev10; ?food object ev108; ?food object ev44; ?food object ev45;
?food object ev49; ?food object ev52; ?food object ev81; ?food object ev91;
?food object ev92; ?food object g135; ?food object p120; ?food object p122;
?food object p140; ?food object p148
?gas-station location ev16; ?gas-station location ev98;
?gas-station location ev99; ?gas-station location p139
?guide-book object ev100; ?guide-book object ev101; ?guide-book object ev102;
?guide-book object ev19; ?guide-book object ev20; ?guide-book object ev5;
?guide-book object ev83; ?guide-book object g133; ?guide-book object p130;
?guide-book object p138; ?guide-book object p144; ?guide-book object p147
?magazine object ev69
?market from ev55; ?market location ev102; ?market location ev109;
?market location ev111; ?market location ev36; ?market location ev50;
?market location ev92; ?market location g158; ?market location p138;
?market location p140; ?market location p141; ?market location p142;
?market obj-attr ev12; ?market obj-attr ev22; ?market obj-attr ev93;
?market obj-attr ev94; ?market obj-attr g123; ?market obj-attr p127;
?market to ev110; ?market to ev116; ?market to ev30;
?market to ev39; ?market to p163; ?market to p167;
?person agent ev1; ?person agent ev100; ?person agent ev101;
?person agent ev102; ?person agent ev103; ?person agent ev104;
?person agent ev105; ?person agent ev106; ?person agent ev107;
?person agent ev108; ?person agent g119; ?person agent g123;
?person agent g124; ?person agent g125; ?person agent g126;
?person agent g132; ?person agent g133; ?person agent g134;
?person agent g171; ?person agent p120; ?person agent p121;
?person agent p122; ?person agent p127; ?person agent p128;
```

```
?person agent p129; ?person agent p130; ?person agent p131;
?person agent p138; ?person object ev41; ?person object ev71;
?person object ev72
?professor from ev61; ?professor from p131
?raw-food object ev111; ?raw-food object ev31; ?raw-food object ev36;
?raw-food object ev4; ?raw-food object ev75; ?raw-food object ev79;
?raw-food object ev80; ?raw-food object g136; ?raw-food object p121;
?raw-food object p141; ?raw-food object p149
?restaurant from ev48; ?restaurant location ev40; ?restaurant location ev81;
?restaurant location ev86; ?restaurant location g156; ?restaurant location p120;
?restaurant obj-attr ev112; ?restaurant obj-attr ev17;
?restaurant obj-attr ev33; ?restaurant obj-attr ev82;
?restaurant obj-attr g124; ?restaurant obj-attr p128;
?restaurant to ev14; ?restaurant to ev34; ?restaurant to ev8;
?restaurant to ev87; ?restaurant to ev89; ?restaurant to p161;
?restaurant to p165
asked act ev112; asked act ev12; asked act ev33; asked act ev94
ate act ev108; ate act ev45; ate act ev81
bank from ev95; bank from p152; bank location ev96; bank location g157;
bank to ev107; bank to ev13; bank to ev28; bank to p162; bank to p166;
bill object ev46
borrowed act ev114; borrowed act ev115; borrowed act ev2; borrowed act ev35;
borrowed act ev6; borrowed act ev77; borrowed act ev85
bought act ev102; bought act ev111; bought act ev36; bought act ev92;
bought act ev98; brought act ev42;
called-up act ev104; called-up act ev24
car location ev7; car location ev88; car location g159; car to p164
cart object ev51
cashier to ev54
checked act ev63
checked-in act ev67
coin object ev113; coin object ev114; coin object ev35; coin object g132;
coin object p143; coin object p146;
comm-link object ev103
cooked act ev75
d-cont goal g132; d-cont goal g133; d-cont goal g134; d-cont goal g135;
d-cont goal g136; d-cont goal g137; d-cont goal g151
d-know goal g123; d-know goal g124; d-know goal g125; d-know goal g126
d-link goal g168;
d-prox goal g155; d-prox goal g156; d-prox goal g157; d-prox goal g158;
d-prox goal g159
drove act ev107; drove act ev116; drove act ev34; drove act ev39;
drove act ev87
entered act ev3; entered act ev40; entered act ev50; entered act ev58;
entered act ev70
exam-room location ev70
examined act ev72
friend from ev115; friend from; ev2 friend from ev6; friend from ev77;
friend from ev85; friend from p144; friend from p145; friend from p154;
friend obj-attr g125; friend obj-attr p129; friend to ev103; friend to ev104;
friend to ev112; friend to ev12; friend to ev24; friend to ev33;
friend to ev94; friend to g168; friend to p127; friend to p128; friend to p169;
friend to p170
gasoline object ev97; gasoline object ev98; gasoline object g134;
gasoline object p139
got act ev10; got act ev20; got act ev31; got act ev37; got act ev51
got-into act ev7
had state ev100; had state ev103; had state ev113; had state ev27;
had state ev76; had state ev79; had state ev84; had state ev91; had state ev97
home to ev48; home to ev55; home to ev64; home to ev73
hospital from ev73; hospital to ev66
hungry state ev1; hungry state ev74
inside state ev109; inside state ev78; inside state ev86; inside state ev88;
inside state ev96; inside state ev99
kitchen location ev3; kitchen location ev75; kitchen location ev78
knew state ev117; knew state ev17; knew state ev22; knew state ev56;
knew state ev82; knew state ev93
left act ev47
left-for act ev48; left-for act ev55; left-for act ev64; left-for act ev73;
line location ev53
listened act ev61
location object ev112; location object ev12; location object ev17;
location object ev22; location object ev33; location object ev82;
location object ev93; location object ev94; location object g123;
location object g124; location object p127; location object p128;
menu object ev42; menu object ev43
money object ev115; money object ev27; money object ev6; money object ev84;
money object ev85; money object ev9; money object ev90; money object ev95;
money object g151; money object p152; money object p153; money object p154
near state ev105
needed state ev49
not mode ev118; not mode ev27; not mode ev74
notebook object ev60
nurse agent ev71
ordered act ev44
p-health goal g171
paid act ev46; paid act ev54
pay-phone location ev105; pay-phone object g155; pay-phone to ev106;
pay-phone to ev26; pay-phone to p160
pb-ask plan p127; pb-ask plan p128; pb-ask plan p129
pb-borrow plan p143; pb-borrow plan p144; pb-borrow plan p145; pb-borrow plan p154
pb-cook plan p121
pb-doctor plan p172
pb-drive plan p165; pb-drive plan p166; pb-drive plan p167
pb-eat plan p122
pb-grasp plan p146; pb-grasp plan p147; pb-grasp plan p148; pb-grasp plan p149;
pb-grasp plan p150
pb-lecture plan p131
pb-letter plan p170
pb-phone plan p169
pb-read plan p130
pb-restaurant plan p120
pb-shopping plan p138; pb-shopping plan p139; pb-shopping plan p140;
pb-shopping plan p141; pb-shopping plan p142
pb-steal plan p153
pb-walk plan p160; pb-walk plan p161; pb-walk plan p162; pb-walk plan p163;
pb-walk plan p164
pb-withdraw plan p152
phone-number object g125; phone-number object p129
picked-up act ev101; picked-up act ev19; picked-up act ev4; picked-up act ev52;
picked-up act ev80
read act ev43; read act ev5; read act ev69; read act ev83
receptionist to ev67
s-hunger goal g119
sat-down act ev59; sat-down act ev68
seat location ev59; seat location ev68
seated act ev41
sick state ev118; sick state ev65
stole act ev9; stole act ev90
stopped act ev16
tested act ev71
time object ev63
tip object ev47
took-note act ev62
took-out act ev60
waited act ev53; waiter agent ev41; waiter agent ev42; waiter from ev114;
waiter from ev35; waiter from ev9; waiter from ev90; waiter from p143;
waiter from p153
walked act ev106; walked act ev110; walked act ev26; walked act ev30;
walked act ev8; walked act ev89
wanted state ev11; wanted state ev15; wanted state ev18; wanted state ev21;
wanted state ev23; wanted state ev25; wanted state ev29; wanted state ev32;
wanted state ev38; wanted state ev57
went act ev13; went act ev14; went act ev28; went act ev66
withdrew act ev95
```

APPENDIX B

DYNASTY I/O story skeletons

DYNASTY's input stories and output inference chains are listed below in skeleton forms with the variables and their instances. The actual stories and inference chains are formed from these skeletons by specifying the ID parts for each variable that appears in the skeleton. The skeletons are used to analyze DSRs and DYNASTY performance in several experiments described in chapter 5.

B.1 Goal/plan story skeletons

Goal/plan story skeletons are listed under the 3 different types: (1) type1 – stories about how several preconditions of a plan should be fulfilled (horizontal reasoning), (2) type 2 – stories about how series of preconditions are fulfilled sequentially (vertical reasoning), and (3) type 3 – stories that combine type 1 and type 2. Each sentence beginning with a → is an input sentence, and output inference chains follow each input sentence skeleton. The goal/plan knowledge to produce the inference chains are listed in the appendix C.1 and appendix C.2.

Variables and instances:

```
?person: John, Mary
?cook-utensil: pan, micro-wave
?raw-food: fish, chicken
?guide-book: michelin-guide, yellow-page
?restaurant: sizzler, mamasion
?food: steak, lobster
?market: vons, lucky
```

type1

skeleton 1

```
--> person was hungry.
?person was not hungry.

--> ?person borrowed ?cook-utensil from friend.
?person was not hungry. ?person cooked ?raw-food.
?person had ?cook-utensil.  ?person borrowed ?cook-utensil from friend.

--> ?person entered kitchen.
?person cooked ?raw-food. ?person was inside kitchen.

--> ?person picked-up ?raw-food.
?person cooked ?raw-food. ?person had ?raw-food.  ?person picked-up ?raw-food.
```

skeleton 2

```
--> ?person was hungry.
?person was not hungry.

--> ?person read ?guide-book.
?person was not hungry.  ?person ate ?food at ?restaurant.
?person knew ?restaurant location.  ?person read ?guide-book.

--> ?person borrowed money from friend.
?person ate ?food at ?restaurant.  ?person had money.
?person borrowed money from friend.

--> ?person got-into car.
?person ate ?food at ?restaurant.  ?person was inside ?restaurant.
?person drove to ?restaurant. ?person was inside car.
```

type2

<u>skeleton 1</u>

```
--> ?person was hungry.
?person was not hungry.

--> ?person wanted (?person knew ?restaurant location).
?person was not hungry.  ?person ate ?food at ?restaurant.
?person knew ?restaurant location.

--> ?person picked-up ?guide-book.
?person knew ?restaurant location.  ?person read ?guide-book.
?person had ?guide-book.  ?person picked-up ?guide-book.
```

<u>skeleton 2</u>

```
--> ?person was hungry.
?person was not hungry.

--> ?person wanted (?person got ?guide-book).
?person was not hungry.  ?person ate ?food at ?restaurant.
?person knew ?restaurant location.  ?person read ?guide-book.
?person had ?guide-book.

--> ?person wanted (?person knew book-store location).
?person had ?guide-book.  ?person bought ?guide-book at book-store.
?person knew book-store location.

--> ?person wanted (?person called-up friend).
?person knew book-store location.
?person asked friend about book-store location.
?person had communication-link to friend.  ?person called-up friend.

--> ?person walked to pay-phone.
?person called-up friend.  ?person was near pay-phone.
?person walked to pay-phone.
```

type3

<u>skeleton 1</u>

```
--> ?person was hungry.
?person was not hungry.

--> ?person asked friend about ?restaurant location.
?person was not hungry.  ?person ate ?food at ?restaurant.
?person knew ?restaurant location.
?person asked friend about ?restaurant location.
```

```
--> ?person drove to ?restaurant.
?person ate ?food at ?restaurant.  ?person was inside ?restaurant.
?person drove to ?restaurant.

--> ?person had no money
?person ate ?food at ?restaurant.  ?person had money.

--> ?person wanted (?person called-up friend).
?person had money.  ?person borrowed money from friend.
?person had comm-link to friend.  ?person called-up friend.

--> ?person borrowed coin from waiter.
?person called-up friend.  ?person had coin.
?person borrowed coin from waiter.
```

<u>skeleton 2</u>

```
--> ?person was hungry.
?person was not hungry.

--> ?person bought ?raw-food at ?market.
?person was not hungry.  ?person cooked ?raw-food in kitchen.
?person had ?raw-food.  ?person bought ?raw-food at ?market.

--> ?person wanted (?person got ?cook-utensil).
?person cooked ?raw-food in kitchen.  ?person had ?cook-utensil.

--> ?person borrowed ?cook-utensil from friend.
?person had ?cook-utensil.  ?person borrowed ?cook-utensil from friend.
```

B.2 Script story skeletons

Script story skeletons are listed below with the output paraphrases. The variables serve as script roles. The script knowledge used to produce the paraphrases is listed in the appendix C.3.

Variables (script roles) and instances:

```
?person: John, Mary
?restaurant: sizzler, mamasion
?food: steak, lobster
?raw-food: fish, chicken
?market: vons, lucky
?class: computer, biology
?professor: simth, alan
?doctor: dr-kim, dr-park
?hospital: ucla-hospital, usc-hospital
```

<u>skeleton 1</u>

```
--> ?person was hungry.
?person was not hungry.

--> ?person entered ?restaurant. ?person ate ?food. ?person left tip.
?person was not hungry.  ?person entered ?restaurant.
waiter seated ?person.  waiter brought menu.
?person read menu.  ?person ordered ?food.
?person ate ?food.  ?person paid bill.
?person left tip.  ?person left ?restaurant for home.
```

<u>skeleton 2</u>

```
--> ?person needed ?raw-food.
?person had ?raw-food.

--> ?person entered ?market. ?person picked-up ?raw-food. ?person paid to cashier.
?person had ?raw-food.  ?person entered ?market.
?person got cart.  ?person picked-up ?raw-food.
?person waited in line.  ?person paid to cashier.
?person left ?market for home.
```

<u>skeleton 3</u>

```
--> ?person wanted (?person learned ?class).
?person learned ?class.

--> ?person entered classroom. ?person listened to ?professor.
--> ?person left classroom for home.
?person learned ?class.  ?person entered classroom.
?person sat-down on seat.  ?person took-out notebook.
?person listened to ?professor.  ?person took-note.
?person checked time.  ?person left ?classroom for home.
```

<u>skeleton 4</u>

```
--> ?person was sick.
?person was not sick.

--> ?person went to ?hospital.  ?person read magazine. ?doctor examined ?person.
?person was not sick.  ?person went to ?hospital.
?person checked-in with receptionist.  ?person sat-down on seat.
?person read magazine.  ?person entered exam-room.
nurse tested ?person.  ?doctor examined ?person.
?person left ?hospital for home.
```

APPENDIX C

DYNASTY goal/plan knowledge

C.1 Goals and plans in DYNASTY

Below are listed the specific goals and plans used in the DYNASTY. The goals and plans are represented using case-role triples. Appendix E shows some of the goal/plan representation examples in the training data. Plans are diverse in their complexity, from the ones which can be mapped to single action to the ones which should be mapped to the specific sequence of actions (script). The general treatment of these goals and plans can be found in [Schank and Abelson, 1977].

Goals

- s-hunger (?person) : ?person wants not to be hungry
- d-know (?person, ?information) : ?person wants to know the ?information
- d-cont (?person, ?phy-obj) : ?person wants to have ?phy-obj
- d-prox (?person, ?place) : ?person wants to be inside/near ?place
- d-link (?person, ?person2) : ?person wants to have communication link to ?person2
- p-health (?person) : ?person wants to preserve his health

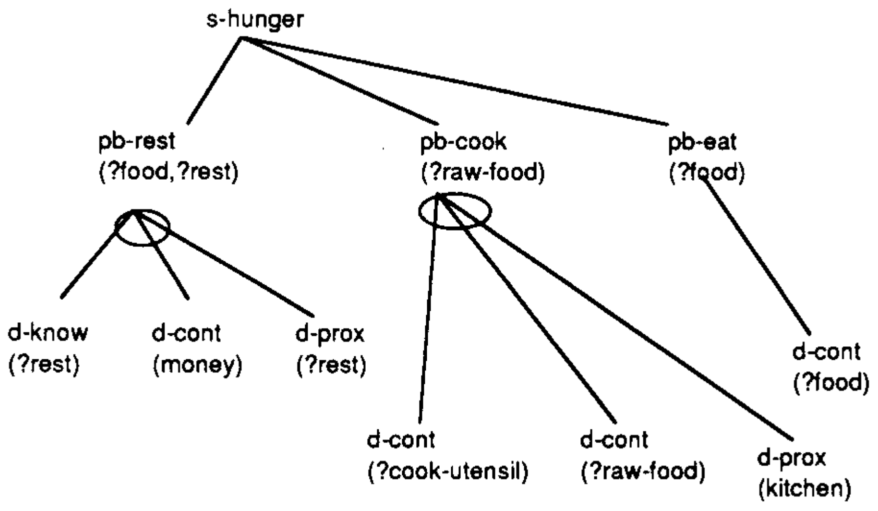
Plans

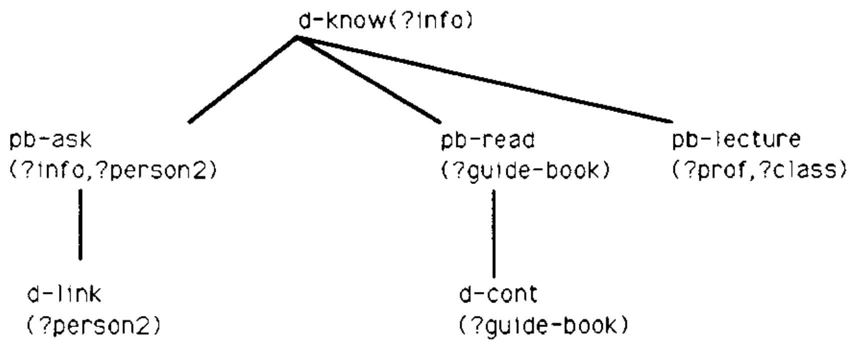
- pb-restaurant (?person, ?food, ?restaurant) : ?person executes restaurant script to eat ?food at the specific ?restaurant
- pb-cook (?person, ?raw-food) : ?person cooks ?raw-food in the kitchen
- pb-eat (?person, ?food) : ?person eats ?food
- pb-ask (?person, ?information, ?person2) : ?person asks ?information to ?person2
- pb-read (?person, ?book) : ?person reads ?book
- pb-lecture (?person, ?class, ?professor) : ?person executes take-lecture script to learn about ?class from ?professor
- pb-shopping (?person, ?phy-obj, ?market) : ?person executes shopping script to buy ?phy-obj at the ?market

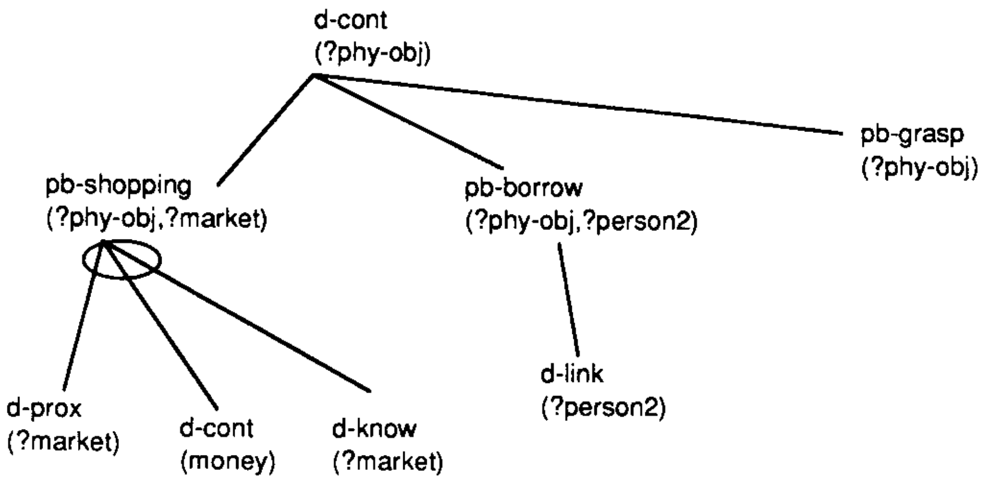
- pb-borrow (?person, ?phy-obj, ?person2) : ?person borrows ?phy-obj from ?person2
- pb-grasp (?person, ?phy-obj) : ?person picks-up ?phy-obj
- pb-walk (?person, ?place) : ?person walks to the ?place
- pb-drive (?person, ?place) : ?person drives a car to the ?place
- pb-phone (?person, ?person2) : ?person calls up to ?person2
- pb-letter (?person, ?person2) : ?person writes a letter to ?person2
- pb-visit-doc (?person, ?doctor, ?hospital) : ?person executes a visit-doctor script to be examined by ?doctor at the ?hospital

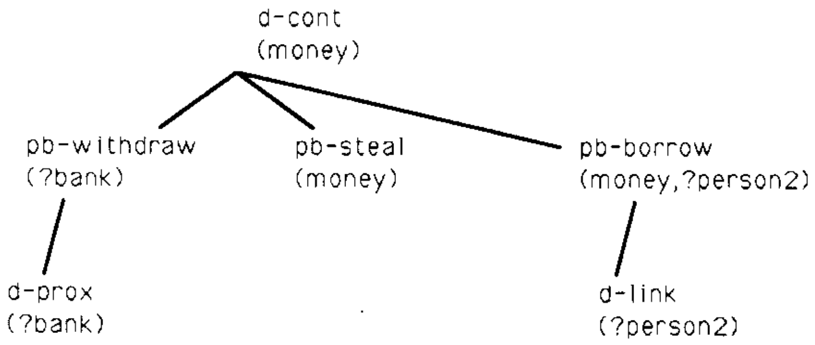
C.2 Goal/plan relations

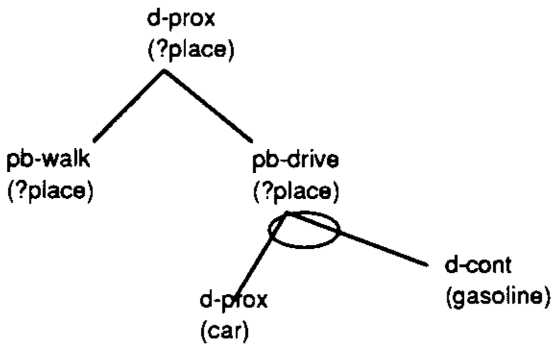
DYNASTY's goal/plan trees are shown below. These goal/plan trees designate all of the goal/plan relations used in the DYNASTY. Ovals indicate AND nodes, where several preconditions must be met in order to execute a plan. All of the AGENT role fillers (?person) are omitted for simplicity.

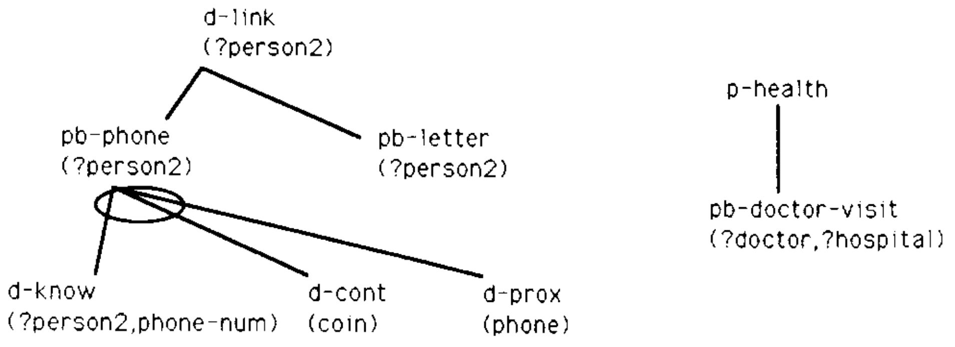





d-cont (?phy-obj)
pb-shopping (?phy-obj,?market)
pb-borrow (?phy-obj,?person2)
pb-grasp (?phy-obj)
d-prox (?market)
d-cont (money)
d-know (?market)
d-link (?person2)


d-cont (money)
pb-withdraw (?bank)
pb-steal (money)
pb-borrow (money,?person2)
d-prox (?bank)
d-link (?person2)


d-prox (?place)
pb-walk (?place)
pb-drive (?place)
d-prox (car)
d-cont (gasoline)



C.3 DYNASTY script knowledge

This appendix shows DYNASTY script knowledge with script roles and instances. DYNASTY is trained with complete input (shown below), but during performance, DYNASTY is tested to see if, given only a partially instantiated input, whether DYNASTY can recognize complete script events with all the correct bindings filled in. Therefore, during performance, DYNASTY was given instantiated versions of only the starred sections (see below), which represent fragments of complete scripts. All of the scripts roles are in capital letters.

Restaurant-Script
Roles: PERSON, RESTAURANT, FOOD
Instances: John, Mary, Sizzler, Macdonald, steak, hamburger

```
PERSON entered RESTAURANT*
waiter seated PERSON
waiter brought menu
PERSON read menu
PERSON ordered FOOD
PERSON ate FOOD*
PERSON paid bill
PERSON left a tip*
PERSON left RESTAURANT for home
```

Attending-Lecture-Script
Roles: PERSON, CLASS, PROFESSOR
Instances: John, Mary, Bio-class, CS-class, Dr-Minsky, Dr-Turing

```
PERSON entered CLASS*
PERSON sat down on the seat*
PERSON took out a notebook
PERSON listened to the PROFESSOR
PERSON took-notes*
PERSON checked the time
PERSON left the CLASS for home*
```

Shopping-Script
Roles: PERSON, STORE, ITEM
Instances: John, Mary, Lucky, Vons, meat, milk

```
PERSON went to STORE*
PERSON get cart
PERSON picked out ITEM*
PERSON waited in line
PERSON paid to cashier*
PERSON left STORE for home
```

Visiting-Doctor-Script
Roles: PERSON, DOCTOR, HOSPITAL
Instances: John, Mary, Dr-Kim, Dr-Johnson, ucla-hos, usc-hos

```
PERSON went to HOSPITAL*
PERSON checked in with receptionist*
PERSON sat down on the seat
PERSON read magazine*
PERSON entered exam-room
nurse tested PERSON
DOCTOR examined PERSON*
PERSON left HOSPITAL for home
```

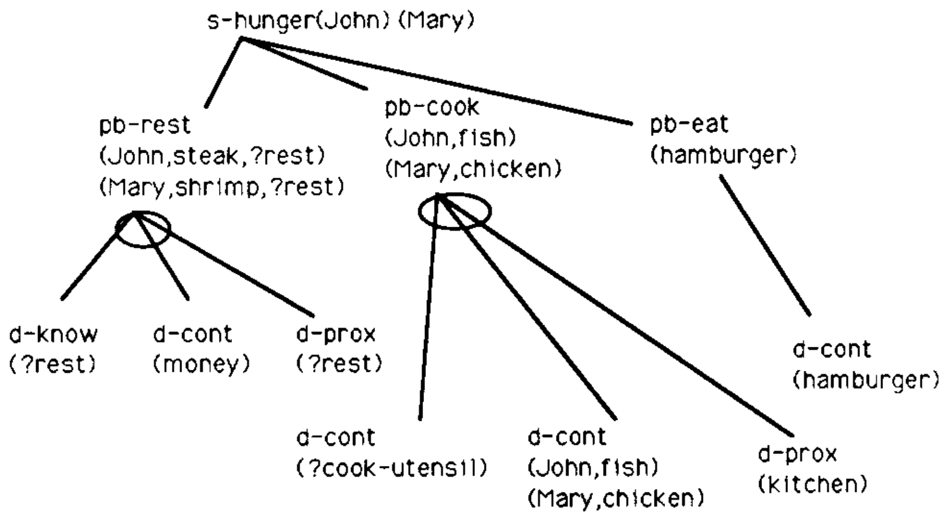
APPENDIX D

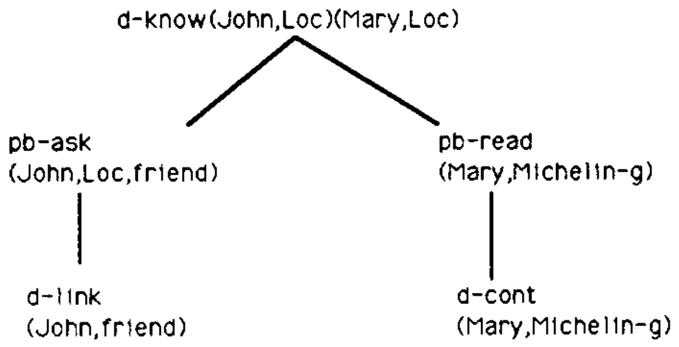
Statistically-biased generalizations

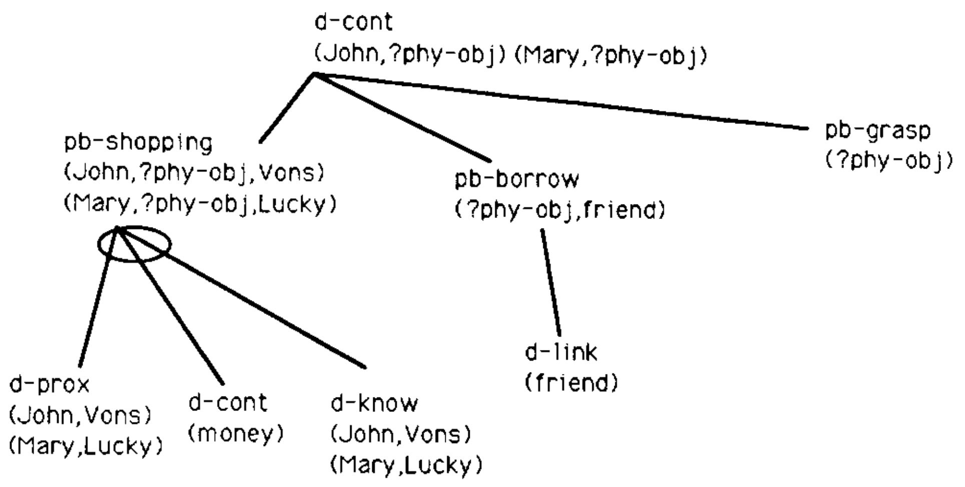
This chapter shows statistically-biased training data and example I/O's for the section 5.3.1.

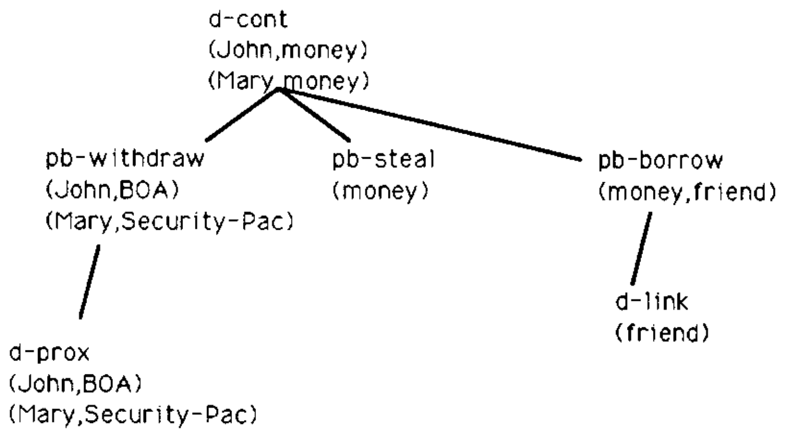
D.1 Statistically-biased goal/plan relations

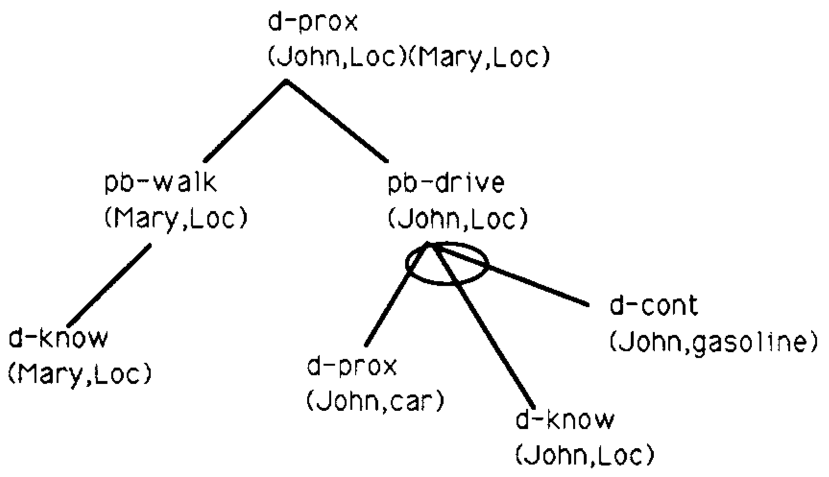
This section lists DYNASTY gp-trees which are statistically-biased. In this experiment, DYNASTY is trained with statistically-biased plan-selection preferences and default role fillers for various actors. For example, John always asks to know, but Mary always reads guide-books. The two different case-role fillers in the parentheses designate the default roles for different actors. Ovals indicate AND nodes as before.

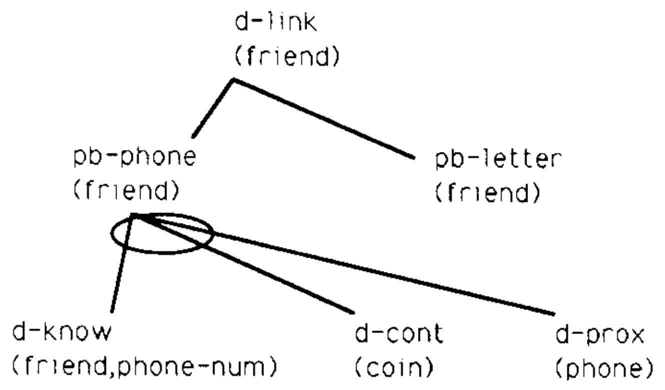





d-cont
(John,?phy-obj) (Mary,?phy-obj)
pb-shopping
(John,?phy-obj,Vons)
(Mary,?phy-obj,Lucky)
pb-borrow
(?phy-obj,friend)
pb-grasp
(?phy-obj)
d-prox
(John,Vons)
(Mary,Lucky)
d-cont
(money)
d-know
(John,Vons)
(Mary,Lucky)
d-link
(friend)


d-cont
(John,money)
(Mary,money)
pb-withdraw
(John,BOA)
(Mary,Security-Pac)
pb-steal
(money)
pb-borrow
(money,friend)
d-prox
(John,BOA)
(Mary,Security-Pac)
d-link
(friend)


d-prox
(John,Loc)(Mary,Loc)
pb-walk
(Mary,Loc)
pb-drive
(John,Loc)
d-know
(Mary,Loc)
d-prox
(John,car)
d-know
(John,Loc)
d-cont
(John,gasoline)



D.2 Statistically-biased I/O

This section lists the examples of statistically-biased input/outputs which were discussed in section 5.3.1. To produce these I/O's, DYNASTY was trained with the statistically-biased gp-trees in the previous section. The convention of listing is same as in section B.1.

example 1

```
--> John was hungry.
John was not hungry.

--> John wanted (John knew Sizzler location).
John was not hungry.  John ate steak at Sizzler.  John knew Sizzler location.

--> John called up friend.
John knew Sizzler location.  John asked friend about Sizzler location.
John had comm-link to friend.  John called up friend.
```

example 2

```
--> Mary was hungry.
Mary was not hungry.

--> Mary wanted (Mary knew Sizzler location).
Mary was not hungry.  Mary ate shrimp at Sizzler.  Mary knew Sizzler location.

--> Mary called up friend.
Mary knew Sizzler location.  Mary read Michelin-guide.
Mary had Michelin-guide.  Mary borrowed Michelin-guide from friend.
Mary had comm-link to friend.  Mary called up friend.
```

example 3

```
--> John was hungry.
John was not hungry.

--> John wanted (John went to Sizzler).
John was not hungry.  John ate steak at Sizzler.  John was inside Sizzler.

--> John asked about Sizzler location.
John was inside Sizzler.  John drove to Sizzler.
John knew Sizzler location.  John asked about Sizzler location.
```

example 4

```
--> Mary was hungry.
Mary was not hungry.

--> Mary wanted (Mary went to Sizzler).
Mary was not hungry.  Mary ate shrimp at Sizzler.  Mary was inside Sizzler.

--> Mary read Michelin-guide.
Mary was inside Sizzler.  Mary walked to Sizzler.
Mary knew Sizzler location.  Mary read Michelin-guide.
```

APPENDIX E

DYNASTY training code and data

This chapter lists DYNASTY's modules with their training data format. The purpose is to show concrete implementation of the training mechanisms, not to provide complete programs that could be downloaded and run. Most of the duplicate and low-level bookkeeping routines are omitted. Only parts of the training data which are enough to show the correct input file format are listed.

E.1 DSR-Learner

This section lists codes and data format of the DSR-Learner which was described in chapter 2 and section 5.1.

The following two programs are used to generate training data for the DSR-Learner. The first program generates unique number for each proposition. The second program produces the concept-triples (appendix A.3) from the proposition-triples (appendix A.2) in order to be loaded into the concept-encoding network. The proposition-triples are generated from the propositions using the ST-Parser program (listed in appendix E.7).

```
/* event number generation for tdata-dsr-ev file
   input file: tdata-te
   output file: tdata-dsr-ev        */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* maxi name size */
#define ifile "tdata-te"  /* input file - training data for Triple-Encoder */
#define ofile "tdata-dsr-ev" /* output tdata-dsr-ev file */
#define evnumb 1000  /* max number of triples */

/* event triple holder */
struct eventd  {
 char bank1[nsize];
 char bank2[nsize];
 char bank3[nsize];
 }  eventdl[evnumb], *evptr;

int startn = 0; /* default starting event number */
char prebank1[nsize];
FILE *fopen(),*ifp,*ofp;

main()
{
  int i;

/* print interface message */
   printf("\nevent start number?");
   scanf("%d",&startn);
   printf("\nstartn is %d\n",startn);
```

```
/* read tdata-te */
   ifp = fopen(ifile,"r");
   evptr = eventdl;

   while(fscanf(ifp,"%s",evptr->bank1) != EOF)
     if (evptr->bank1[0] != ';') {
     fscanf(ifp,"%s %s",evptr->bank2,evptr->bank3);
     evptr++;
     }

   evptr--;
   printf("%s %s %s\n",evptr->bank1,evptr->bank2,evptr->bank3);

     fclose(ifp);

/* assign number */
   ofp = fopen(ofile,"w");

   evptr = eventdl;

   while(evptr->bank1[0] != '\0')  {
     strcpy(prebank1,evptr->bank1);
     while(strcmp(evptr->bank1,prebank1) == 0 && evptr->bank1[0] != '\0') {
if (evptr->bank1[0] == 'e')
  fprintf(ofp,"ev%d %s %s\n",startn,evptr->bank2,evptr->bank3);
        else if (evptr->bank1[0] == 'p')
  fprintf(ofp,"p%d %s %s\n",startn,evptr->bank2,evptr->bank3);
        else if (evptr->bank1[0] == 'g')
  fprintf(ofp,"g%d %s %s\n",startn,evptr->bank2,evptr->bank3);

strcpy(prebank1,evptr->bank1);
evptr++;
} /* inner while */
      startn++;
      fprintf(ofp,"\n");
      }

      fclose(ofp);

}


/* datagenerator for tdata-dsr-obj from tdata-dsr-ev
inputfile: tdata-dsr-ev
outputfile: tdata-dsr-obj  */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* max name size */
#define ifile "tdata-dsr-ev"
#define ofile "tdata-dsr-obj" /* output tdata-dsr-obj file */
#define evnumb 1000

struct eventd  {
 char bank1[nsize];
 char bank2[nsize];
 char bank3[nsize];
 }  eventdl[evnumb], *evptr, *evptr2;

char curbank3[nsize];

FILE *fopen(),*ifp,*ofp;

main()
{
  int i;


/* input to memory */
```

```
   ifp = fopen(ifile,"r");
   evptr = eventdl;

   while(fscanf(ifp,"%s",evptr->bank1) != EOF) {
     fscanf(ifp,"%s %s",evptr->bank2,evptr->bank3);
     evptr++;
     }

   evptr--;
   printf("%s %s %s\n",evptr->bank1,evptr->bank2,evptr->bank3);

  fclose(ifp);

/* print triple with bank3 contiguously  */
   ofp = fopen(ofile,"w");
   evptr = eventdl;

   while(evptr->bank1[0] != '\0') {
    evptr2 = evptr;

    while(evptr2->bank1[0] != '\0') {
    if (evptr2->bank1[0] != 'z' && strcmp(evptr->bank3,evptr2->bank3) == 0) {
      fprintf(ofp,"%s %s %s\n",evptr2->bank3,evptr2->bank2,evptr2->bank1);
      evptr2->bank1[0] = 'z';
      }
      evptr2++;
      }
      evptr++;
      }
      fclose(ofp);

}
```

The following shows training data format for the DSR-Learner. The first group shows the proposition-triples to be loaded into the proposition-encoding network, and the second group shows the concept-triples to be loaded into the concept-encoding network. The complete triples to learn the DSRs for whole vocabularies are listed in appendix A.

```
;event-triples
ev1 state hungry; ev1 agent ?person
ev2 act called-up; ev2 agent ?person; ev2 to friend
ev3 state wanted; ev3 agent ?person; ev3 object ev2
ev4 state had; ev4 agent ?person; ev4 object money; ev4 mode not
ev5 act asked; ev5 agent ?person; ev5 object location;
ev5 obj-attr ?restaurant; ev5 to friend
....
....

;concept-triples
?class subject p38;
?food food-type p21; ?food object ev24; ?food object ev9
?guide-book object ev34
?person agent ev1; ?person agent ev10; ?person agent ev11;
?person agent ev12; ?person agent ev13; ?person agent ev14;
?person agent ev15; ?person agent ev16; ?person agent ev17;
?person agent ev18; ?person agent ev19;
....
....
```

The following program learns DSR representations from the given proposition sets. The propositions are loaded in two different forms: concept-triples and event-triples.

```
/* dsr-learner network module
```

```
   input file: tdata-dsr-obj, tdata-dsr-ev into memory
   input symbol dictionary: conbol-object, conbol-event, conbol-case
   output symbol dictionary: conbol-object, conbol-event
   output weight file: weight-object, weight-event
   usage: learn-dsr < sim.para > dsr-log
   input parameter: cycle, epoch, loadweight_flag
   dump weight and conbol at every snapshot interval
   weights frozen at every snapshot interval to verify encoding  */

#include <stdio.h>
#include <math.h>

#define snapshot 10 /* snapshot epoch ; dump weight and conbol */

#define nsize 20 /* number of characters in name */
#define rsize 10 /* representation size */
#define lsize 12 /* case-role size  */
#define iosize lsize+2*rsize /* input/output size in XRAAM */

/* max data size */
#define nums_ev 200 /* number of event symbol in the buffer  */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ev 1000 /* number of triples in tdata-dsr-ev  */
#define nums_td_obj 1000 /* number of triples in tdata-dsr-obj */

#define dfile "tdata-dsr-obj" /* input concept-triple file */
#define dfile2 "tdata-dsr-ev" /* input event-triple file */
#define sfile "conbol-object" /* DSR representation table for object */
#define sfile2 "conbol-event" /* DSR representation table for event, buffer */
#define lfile "conbol-case"  /* case-role representation table */
#define wfile "weight-object" /* concept-encoder weight matrix */
#define wfile2 "weight-event" /* proposition-encoder weight matrix */

/* symbol store for object */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* symbol store for event */
struct evstore {
   char name[nsize];
   float crep[rsize];
   } evstorel[nums_ev],*symp_ev;

/* symbol store for case-role */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is an array number pointing to the
object in the conbol-object file */
struct tdata_dsr_ev {
  int evnum;
  int case_role;
  int object;
  } tdata_dsr_evl[nums_td_ev],*tdp1;

struct tdata_dsr_obj {
  int object;
  int case_role;
  int evnum;
  } tdata_dsr_objl[nums_td_obj],*tdp2;

/* XRAAM network current representation holder */
char repn[nsize],linkn[nsize],noden[nsize]; /* input name holder */
float rep[rsize],link[lsize],node[rsize]; /* input rep holder  */
```

```
/* object encoder/decoder network */
float in_obj[iosize],out_obj[iosize],hidden_obj[rsize],teach_obj[iosize];
float wih_obj[iosize][rsize],who_obj[rsize][iosize];
float hbias_obj[rsize],obias_obj[iosize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* global file pointer  */
FILE *fopen(),*dfp,*dfp2,*sfp,*sfp2,*lfp,*wfp,*wfp2;

/* simulation set up */
char mcycle[4],epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.07; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/***********      main driver ***************/
main()
{
  int max,total,mtotal; /* max cycles, epochs */
  int i;

  sim_setup(); /* set-up simulation parameters */

  max = atoi(mcycle); /* max cycles in BP  */
  mtotal = atoi(epochs); /* max epochs in training  */

  read_to_store();  /* read from initial conbol to internal data structure   */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight */
  else load_weight(); /* load previous weight for continuous training */

  /* print sim environment */
  printf("\n max cycles %d",max);
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n initial BP parameter %f %f",etha,alpha);

 /* print initial weight to verify */
  print_weight();

 /* read training data into memory */
 read_td_obj();
 read_td_ev();

/* repeat cycle for each epoch; one epoch is for entire training data */
for (total=0;total<mtotal+1;total++)  { /* for max epoch */

  printf("\n epoch number %d object encoding \n",total);
  if (total % snapshot == 0) printf("\n re-encodig training verification\n");

     /* object encoding */
     tdp2 = tdata_dsr_objl;

     while(tdp2->object != 9999) {

       strcpy(repn,objstorel[tdp2->object].name);
       strcpy(linkn,casestorel[tdp2->case_role].name);
       strcpy(noden,evstorel[tdp2->evnum].name);

       for (i=0;i<rsize;i++)
rep[i] = objstorel[tdp2->object].crep[i];
       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp2->case_role].rep[i];
       for (i=0;i<rsize;i++)
node[i] = evstorel[tdp2->evnum].crep[i];

       if (total % snapshot == 0) prop_obj(total);
       else back_prop_obj(max,total);

       for (i=0;i<rsize;i++)
       objstorel[tdp2->object].crep[i] = hidden_obj[i];

       if (total == 0)
       printf("\nrestored into %s\n",objstorel[tdp2->object].name);

      tdp2++;
       } /* while */

  printf("\n epoch number %d event encoding \n",total);

 /* event encoding */
 tdp1 = tdata_dsr_evl;
  while(tdp1->evnum != 9999) {

       strcpy(repn,evstorel[tdp1->evnum].name);
       strcpy(linkn,casestorel[tdp1->case_role].name);
       if (tdp1->object >= nums_obj) /* event */
       strcpy(noden,evstorel[(tdp1->object)-nums_obj].name);
       else
       strcpy(noden,objstorel[tdp1->object].name);

       for (i=0;i<rsize;i++)
rep[i] = evstorel[tdp1->evnum].crep[i];
       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];
       for (i=0;i<rsize;i++)
if (tdp1->object >= nums_obj)  /* recursive event */
  node[i] = evstorel[(tdp1->object)-nums_obj].crep[i];
        else
  node[i] = objstorel[tdp1->object].crep[i];

       if (total % snapshot == 0) prop_ev(total);
       else back_prop_ev(max,total);

       for (i=0;i<rsize;i++)
       evstorel[tdp1->evnum].crep[i] = hidden_ev[i];

       if (total == 0)
       printf("\nrestored into %s\n",evstorel[tdp1->evnum].name);

      tdp1++;
} /* while */

 /* dump snapshot for each snapshot point */
  if (total % snapshot == 0) {
      print_sym();
      dump_weight();
      make_dict();
      }
   } /* out for */

/* post processing  */
   dump_weight();
   make_dict();

} /* end of main */

/*********** simulation bookkeeping ****************/
/* set up simulation parameters from terminal input */
sim_setup()
{
 printf("\n how many cycles in BP? (max 999)");
 scanf("%s",mcycle);
 printf("\n how many epochs in simulation?(max 99)");
 scanf("%s",epochs);
 printf("\n load previous weight? (y/n)");
 scanf("%s",load_flag);
}

/* print initial weight matrices to verify correct loading */
print_weight()
{
int i,j;

  printf("wih_obj\n");
   for (i=0;i<iosize;i++) {
    for (j=0;j<rsize;j++) printf("%f ",wih_obj[i][j]); printf("\n");
     }

  printf("who_obj\n");
   for(i=0;i<rsize;i++) {
    for(j=0;j<iosize;j++) printf("%f ",who_obj[i][j]); printf("\n");
     }

  /* bias dump */
   printf("hbias_obj\n");
   for (i=0;i<rsize;i++)
     printf("%f ",hbias_obj[i]); printf("\n");

   printf("obias_obj\n");
   for (i=0;i<iosize;i++)
     printf("%f ",obias_obj[i]); printf("\n");

   printf("wih_ev\n");
   for (i=0;i<iosize;i++) {
    for (j=0;j<rsize;j++) printf("%f ",wih_ev[i][j]); printf("\n");
     }

   printf("who_ev\n");
   for(i=0;i<rsize;i++) {
    for(j=0;j<iosize;j++) printf("%f ",who_ev[i][j]); printf("\n");
     }

   printf("hbias_ev\n");
   for (i=0;i<rsize;i++) printf("%f ",hbias_ev[i]); printf("\n");

   printf("obias_ev\n");
   for (i=0;i<iosize;i++) printf("%f ",obias_ev[i]); printf("\n");
}

/* read tables into memory */
read_to_store()
{
   int i;

   /* read to objstore */
   sfp = fopen(sfile,"r"); /* open conbol-object  */
   symp_obj = objstorel;

   while(fscanf(sfp,"%s",symp_obj->name) != EOF) {

     for(i=0;i<rsize;i++)
      fscanf(sfp,"%f",&(symp_obj->crep[i]));

     symp_obj++;
   }
   fclose(sfp);

   /* read to evstore   */
   sfp2 = fopen(sfile2,"r"); /* open conbol-event  */
   symp_ev = evstorel;
```

```
    while(fscanf(sfp2,"%s",symp_ev->name) != EOF) {

      for(i=0;i<rsize;i++)
       fscanf(sfp2,"%f",&(symp_ev->crep[i]));

      symp_ev++;
    }
    fclose(sfp2);

    /* read to case store  */
    lfp = fopen(lfile,"r"); /* open conbol-case */
    casep = casestorel;

    while(fscanf(lfp,"%s",casep->name) != EOF) {
      if (casep->name[0] != ';') {
      for(i=0;i<lsize;i++)
       fscanf(lfp,"%f",&(casep->rep[i]));
      casep++;
      }
    }
    fclose(lfp);

   /* test probe  */
   printf("\nobjstorel\n");
   printf("\n%s  ",objstorel[3].name);
   for(i=0;i<rsize;i++)
    printf("%.1f ",objstorel[3].crep[i]);

   printf("\ncasestorel\n");
   printf("%s  ",casestorel[3].name);
   for(i=0;i<lsize;i++)
    printf("%.1f ",casestorel[3].rep[i]);
   /*   test end */

}

/* read concept-triple training data file into memory for speed up */
read_td_obj()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-dsr-obj */
 tdp2 = tdata_dsr_objl;

 while(fscanf(dfp,"%s",repn) != EOF) {
   fscanf(dfp,"%s",linkn);
   fscanf(dfp,"%s",noden);

 /* object representation */
 i=0;
 while((strcmp(repn,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] == '\0') printf("\nerror no %s",repn);
 else tdp2->object = i;

 /* load event rep   */
 i=0;
 while((strcmp(noden,evstorel[i].name) != 0) && (evstorel[i].name[0] != '\0')) i++;
 if (evstorel[i].name[0] == '\0') printf("\nerror: no %s",noden);
 else tdp2->evnum = i;

 /* load case rep */
 i=0;
 while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
 if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
 else tdp2->case_role = i;

tdp2++;
} /* while */
```

```
 tdp2->object = 9999; /* end of data file */

fclose(dfp);

printf("\n tdata_dsr_objl");
for (i=0;i<40;i++)
 printf("\n %d %d %d",tdata_dsr_objl[i].object,tdata_dsr_objl[i].case_role,
 tdata_dsr_objl[i].evnum);
}

/* read event-triple training data file into memory for speed up  */
read_td_ev()
{
 int i;

 dfp2 = fopen(dfile2,"r"); /* open tdata-dsr-ev */
 tdp1 = tdata_dsr_evl;

 while(fscanf(dfp2,"%s",repn) != EOF) {
   fscanf(dfp2,"%s",linkn);
   fscanf(dfp2,"%s",noden);

 /* load event representation */
 i=0;
 while((strcmp(repn,evstorel[i].name) != 0) && (evstorel[i].name[0] != '\0')) i++;
 if (evstorel[i].name[0] == '\0') printf("\nerror: no %s\n",repn);
 else tdp1->evnum = i;

 /* load object rep   */
 i=0;
 while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] != '\0') tdp1->object = i;
 else {
   i=0;
   while((strcmp(noden,evstorel[i].name) != 0) && (evstorel[i].name[0] != '\0')) i++;
   if (evstorel[i].name[0] == '\0') printf("\nerror: no %s\n",noden);
   else tdp1->object = i + nums_obj;  /* it is event */
 }

 /* load case rep */
 i=0;
 while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
 if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
 else tdp1->case_role = i;

 tdp1++; /* next store */
 } /* while */
 tdp1->evnum = 9999; /* end of data */
 fclose(dfp2);

 printf("\n tdata_dsr_ev");
 for(i=0;i<40;i++)
  printf("\n %d %d %d",tdata_dsr_evl[i].evnum,tdata_dsr_evl[i].case_role,
  tdata_dsr_evl[i].object);

}

/**************** backpropagation and forward propagation *******/
/* perform backprop for concept-encoder network */
back_prop_obj(max,total)
int max,total;
{
/* define BP parameters; internal matrix */
  float eout[iosize],ehid[rsize],dwho[rsize][iosize],
dwih[iosize][rsize],
mwho[rsize][iosize],mwih[iosize][rsize],net,pesig,
mhbias[rsize],mobias[iosize],dhbias[rsize],dobias[iosize];
  float sigmoid();
  int i,j,pos,cycle,flag;
```

```
/* clear momentum weight matrices */
  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) { mwih[i][j] = 0; mwho[j][i] = 0; }
  for (i=0;i<rsize;i++) mhbias[i] = 0;
  for (i=0;i<iosize;i++) mobias[i] = 0;

/* load data for cycles */
  /* input layer */

  pos = 0;
  for(i=0;i<rsize;i++) in_obj[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_obj[pos++] = link[i];
  for(i=0;i<rsize;i++) in_obj[pos++] = node[i];

 /* teach layer : auto-associative network  */
   for(i=0;i<iosize;i++) teach_obj[i] = in_obj[i];

 /* back_prop procedure - do until we get desirable
 accuracy (two threshold used) or until max cycles */

   for(cycle = 0;cycle<max;cycle++)  {

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
 net = net + wih_obj[j][i]*in_obj[j];

 net = net + hbias_obj[i];
 hidden_obj[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
       net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_obj[j][i]*hidden_obj[j];

 net = net + obias_obj[i];
 out_obj[i] = sigmoid(net);
 }

  /* accuracy test  -- threshold is v= 0.1 t= 0.2 according to J. Pollack */
  flag = 0;
  for(i=0;i<rsize;i++)
    if (abs(teach_obj[i] - out_obj[i]) > 0.1) flag = 1;
  for(i=rsize;i<iosize;i++)
    if (abs(teach_obj[i] - out_obj[i]) > 0.2) flag = 1;

  if (flag == 0) {
 printf("\n epoch %d up to %d cycles\n",total,cycle);
 break; /* go to printnet */  }

  /* backward prop */
  /* calculate error signal for output */
    for(i=0;i<iosize;i++)
     eout[i] = (teach_obj[i] - out_obj[i])*out_obj[i]*(1-out_obj[i]);

  /* modify weight for eout */
  for(i=0;i<rsize;i++)
   for(j=0;j<iosize;j++) {
    dwho[i][j] = etha*eout[j]*hidden_obj[i]+alpha*mwho[i][j];
    who_obj[i][j] = who_obj[i][j] + dwho[i][j];
    mwho[i][j] = dwho[i][j];
   }

  /* adjust out bias */
  for (i=0;i<iosize;i++) {
    dobias[i] = etha*eout[i]*1 + alpha*mobias[i];
    obias_obj[i] = obias_obj[i] + dobias[i];
    mobias[i] = dobias[i];
    }

  /* calculate error signal for hidden */
  for(i=0;i<rsize;i++) {
   pesig = 0.0;
   for(j=0;j<iosize;j++)
    pesig = pesig + eout[j]*who_obj[i][j];

   ehid[i] = hidden_obj[i]*(1-hidden_obj[i])*pesig;
   }

  /* modify weight for ehid */
  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) {
    dwih[i][j] = etha*ehid[j]*in_obj[i] + alpha*mwih[i][j];
    wih_obj[i][j] = wih_obj[i][j] + dwih[i][j];
    mwih[i][j] = dwih[i][j];
    }

  /* adjust hid bias */
  for (i=0;i<rsize;i++) {
    dhbias[i] = etha*ehid[i]*1 + alpha*mhbias[i];
    hbias_obj[i] = hbias_obj[i] + dhbias[i];
    mhbias[i] = dhbias[i];
    }

   } /* for cycle */
} /* end of back_prop_obj */

/* perform backprop for proposition-encoding network */
back_prop_ev(max,total)
int max,total;
{
/* network parameters */
  float eout[iosize],ehid[rsize],dwho[rsize][iosize],
dwih[iosize][rsize],
mwho[rsize][iosize],mwih[iosize][rsize],net,pesig,
mhbias[rsize],mobias[iosize],dhbias[rsize],dobias[iosize];
  float sigmoid();
  int i,j,pos,cycle,flag;

/* clear momentum */
  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) { mwih[i][j] = 0; mwho[j][i] = 0; }
  for (i=0;i<rsize;i++) mhbias[i] = 0;
  for (i=0;i<iosize;i++) mobias[i] = 0;

/* load data for cycles */
  /* input layer */

  pos = 0;
  for(i=0;i<rsize;i++) in_ev[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_ev[pos++] = link[i];
  for(i=0;i<rsize;i++) in_ev[pos++] = node[i];

 /* teach layer : auto-associative network  */
   for(i=0;i<iosize;i++) teach_ev[i] = in_ev[i];

 /* back_prop procedure - do until we get desirable
 accuracy (two threshold used) or until max cycles */
```

```
   for(cycle = 0;cycle<max;cycle++)  {

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
net = net + wih_ev[j][i]*in_ev[j];

net = net + hbias_ev[i];
hidden_ev[i] = sigmoid(net);
}

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
       net = 0.0;
       for(j=0;j<rsize;j++)
net = net + who_ev[j][i]*hidden_ev[j];

net = net + obias_ev[i];
out_ev[i] = sigmoid(net);
}

  /* accuracy test  -- threshold is v= 0.1 t= 0.2 */
  flag = 0;
  for(i=0;i<rsize;i++)
    if (abs(teach_ev[i] - out_ev[i]) > 0.1) flag = 1;
  for(i=rsize;i<iosize;i++)
    if (abs(teach_ev[i] - out_ev[i]) > 0.2) flag = 1;

  if (flag == 0) {
printf("\n epoch %d up to %d cycles\n",total,cycle);
break;  }

  /* backward prop */
  /* calculate error signal for output */
    for(i=0;i<iosize;i++)
     eout[i] = (teach_ev[i] - out_ev[i])*out_ev[i]*(1-out_ev[i]);

  /* modify weight for eout */
  for(i=0;i<rsize;i++)
   for(j=0;j<iosize;j++) {
    dwho[i][j] = etha*eout[j]*hidden_ev[i]+alpha*mwho[i][j];
    who_ev[i][j] = who_ev[i][j] + dwho[i][j];
    mwho[i][j] = dwho[i][j];
   }

  /* adjust out bias */
  for (i=0;i<iosize;i++) {
    dobias[i] = etha*eout[i]*1 + alpha*mobias[i];
    obias_ev[i] = obias_ev[i] + dobias[i];
    mobias[i] = dobias[i];
    }

  /* calculate error signal for hidden */
  for(i=0;i<rsize;i++) {
   pesig = 0.0;
   for(j=0;j<iosize;j++)
    pesig = pesig + eout[j]*who_ev[i][j];

   ehid[i] = hidden_ev[i]*(1-hidden_ev[i])*pesig;
   }

  /* modify weight for ehid */
  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) {
    dwih[i][j] = etha*ehid[j]*in_ev[i] + alpha*mwih[i][j];
    wih_ev[i][j] = wih_ev[i][j] + dwih[i][j];
    mwih[i][j] = dwih[i][j];
    }

  /* adjust hid bias */
  for (i=0;i<rsize;i++) {
    dhbias[i] = etha*ehid[i]*1 + alpha*mhbias[i];
    hbias_ev[i] = hbias_ev[i] + dhbias[i];
    mhbias[i] = dhbias[i];
    }

  } /* for cycle */
} /* end of back_prop_ev */

/* calculate sigmoid function */
float sigmoid(x)
float x;
{
 double exp();
 return( 1.0 / (1.0 + exp(-x)));
}


/* do forward prop for concept-encoding network for encode testing */
prop_obj()
{
 float sigmoid(),net;
 int i,j,pos,flag;

  /* load data */
  pos = 0;
  for(i=0;i<rsize;i++) in_obj[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_obj[pos++] = link[i];
  for(i=0;i<rsize;i++) in_obj[pos++] = node[i];

 /* teach layer */
 for(i=0;i<iosize;i++) teach_obj[i] = in_obj[i];

   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
 net = net + wih_obj[j][i]*in_obj[j];

 net = net + hbias_obj[i];
 hidden_obj[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
        net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_obj[j][i]*hidden_obj[j];

 net = net + obias_obj[i];
 out_obj[i] = sigmoid(net);
 }


  /* print net for verify */
   printf("%s %s %s\n",repn,linkn,noden);
   printf("hidden\n");
   for(i=0;i<rsize;i++) printf("%.1f ",hidden_obj[i]); printf("\n");
   printf("output-teach pair\n");
   for(i=0;i<iosize;i++) printf("%.1f ",out_obj[i]); printf("\n");
   for(i=0;i<iosize;i++) printf("%.1f ",teach_obj[i]);   printf("\n");
```

```
} /* prop_obj */

/* perform forward prop for proposition-encoding network */
prop_ev()
{
 float sigmoid(),net;
 int i,j,pos,flag;

  /* load data */
  pos = 0;
  for(i=0;i<rsize;i++) in_ev[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_ev[pos++] = link[i];
  for(i=0;i<rsize;i++) in_ev[pos++] = node[i];

/* teach layer */
for (i=0;i<iosize;i++) teach_ev[i] = in_ev[i];

   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
 net = net + wih_ev[j][i]*in_ev[j];

 net = net + hbias_ev[i];
 hidden_ev[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
        net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_ev[j][i]*hidden_ev[j];

 net = net + obias_ev[i];
 out_ev[i] = sigmoid(net);
 }

  /* print net  */
   printf("%s %s %s\n",repn,linkn,noden);
   printf("hidden\n");
   for(i=0;i<rsize;i++) printf("%.1f ",hidden_ev[i]); printf("\n");
   printf("output-teach pair\n");
   for(i=0;i<iosize;i++) printf("%.1f ",out_ev[i]); printf("\n");
   for(i=0;i<iosize;i++) printf("%.1f ",teach_ev[i]);   printf("\n");
} /* prop_ev */

/* print symbolic dictionary */
print_sym()
{
 int i,j,k;

 float dist[10][10];

/* print out conbol-object  */
 printf("\n symbolic store snapshot \n");
 symp_obj = objstorel;
 while(symp_obj->name[0] != '\0') {
  printf("\n%s",symp_obj->name);
  printf("   ");
  for(i=0;i<rsize;i++)
   printf(" %.1f",symp_obj->crep[i]);

  symp_obj++;
  }

/* print out conbol-event buffer  */
```

```
 printf("\n symbolic store snapshot\n");
 symp_ev = evstorel;
 while(symp_ev->name[0] != '\0') {
  printf("\n%s",symp_ev->name);
  printf("   ");
  for(i=0;i<rsize;i++)
   printf(" %.1f",symp_ev->crep[i]);

  symp_ev++;
  }

/* print first 10 object E-distance in conbol-object */

for (i=0;i<10;i++)
 for(j=0;j<10;j++) {
    dist[i][j] = 0;
    for(k=0;k<rsize;k++)
     dist[i][j] = dist[i][j] + (objstorel[i].crep[k] - objstorel[j].crep[k])
                              * (objstorel[i].crep[k] - objstorel[j].crep[k]);
      }

printf("\n E-dist square between first 10 conbol-object\n");
for (i=0;i<10;i++)
printf("%s ",objstorel[i].name);
printf("\n");
for (i=0;i<10;i++) {
  for(j=0;j<10;j++)
    printf("%.1f ",dist[i][j]);
    printf("\n");
    }

}

/* make global-dictionary training data; see section E.3 */
make_dict()
{
 int i;

/* print out conbol-object  */

 sfp = fopen(sfile,"w");

 symp_obj = objstorel;
 while(symp_obj->name[0] != '\0') {
  fprintf(sfp,"\n%s",symp_obj->name);
  fprintf(sfp,"   ");
  for(i=0;i<rsize;i++)
   fprintf(sfp," %.1f",symp_obj->crep[i]);

  symp_obj++;
  }

  fclose(sfp);

/* print out conbol-event */
 sfp2 = fopen(sfile2,"w");

 symp_ev = evstorel;
 while(symp_ev->name[0] != '\0') {
  fprintf(sfp2,"\n%s",symp_ev->name);
  fprintf(sfp2,"   ");
  for(i=0;i<rsize;i++)
   fprintf(sfp2," %.1f",symp_ev->crep[i]);

  symp_ev++;
  }

 fclose(sfp2);
```

```
}

/* random number assign for initial weights; between -1 and 1 */
rassign()
{
  int i,j;

  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) {
    wih_obj[i][j] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    who_obj[j][i] = -1.0 + 2.0*(rand()/32767.0);
    }

  for(i=0;i<iosize;i++)
   for(j=0;j<rsize;j++) {
    wih_ev[i][j] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    who_ev[j][i] = -1.0 + 2.0*(rand()/32767.0);
    }

  /* initial random bias */
  for (i=0;i<rsize;i++) {
    hbias_obj[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    hbias_ev[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    }

  for (i=0;i<iosize;i++) {
    obias_obj[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    obias_ev[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
    }
}

/********** output bookkeeping ***********/
/* dump weight matrices for snapshot */
dump_weight()
{
   int i,j;

  /* dump weight for object encoder/decoder net */
   wfp = fopen(wfile,"w");

   for (i=0;i<iosize;i++) {
    for (j=0;j<rsize;j++)
     fprintf(wfp,"%f ",wih_obj[i][j]);

     fprintf(wfp,"\n");
     }

   for(i=0;i<rsize;i++) {
    for(j=0;j<iosize;j++)
     fprintf(wfp,"%f ",who_obj[i][j]);

     fprintf(wfp,"\n");
     }

  /* bias dump */
   for (i=0;i<rsize;i++)
     fprintf(wfp,"%f ",hbias_obj[i]);
   fprintf(wfp,"\n");

   for (i=0;i<iosize;i++)
     fprintf(wfp,"%f ",obias_obj[i]);
   fprintf(wfp,"\n");

     fclose(wfp);

  /* dump weight for event encoder/decoder net  */

   wfp2 = fopen(wfile2,"w");
   for (i=0;i<iosize;i++) {
    for (j=0;j<rsize;j++)
     fprintf(wfp2,"%f ",wih_ev[i][j]);

     fprintf(wfp2,"\n");
     }

   for(i=0;i<rsize;i++) {
    for(j=0;j<iosize;j++)
     fprintf(wfp2,"%f ",who_ev[i][j]);

     fprintf(wfp2,"\n");
     }
  /* bias dump */
   for (i=0;i<rsize;i++)
     fprintf(wfp2,"%f ",hbias_ev[i]);
   fprintf(wfp2,"\n");

   for (i=0;i<iosize;i++)
     fprintf(wfp2,"%f ",obias_ev[i]);
   fprintf(wfp2,"\n");

     fclose(wfp2);
}

/* load previous weight matrices for continuous simulation */
load_weight()
{

   int i,j;

  /* load weight for object encoder/decoder net */
   wfp = fopen(wfile,"r");

   printf("\n\n load weight \n\n");

   for (i=0;i<iosize;i++)
    for (j=0;j<rsize;j++)
     fscanf(wfp,"%f",&wih_obj[i][j]);


   for(i=0;i<rsize;i++)
    for(j=0;j<iosize;j++)
     fscanf(wfp,"%f",&who_obj[i][j]);

  /* bias load */
   for (i=0;i<rsize;i++)
     fscanf(wfp,"%f ",&hbias_obj[i]);

   for (i=0;i<iosize;i++)
     fscanf(wfp,"%f ",&obias_obj[i]);

     fclose(wfp);


  /* load weight for event encoder/decoder net  */

   wfp2 = fopen(wfile2,"r");

   for (i=0;i<iosize;i++)
    for (j=0;j<rsize;j++)
     fscanf(wfp2,"%f",&wih_ev[i][j]);


   for(i=0;i<rsize;i++)
    for(j=0;j<iosize;j++)
     fscanf(wfp2,"%f",&who_ev[i][j]);

  /* bias load */
```

```
    for (i=0;i<rsize;i++)
      fscanf(wfp2,"%f ",&hbias_ev[i]);

    for (i=0;i<iosize;i++)
      fscanf(wfp2,"%f ",&obias_ev[i]);

      fclose(wfp2);
}
```

E.2 Triple-Encoder

This section lists codes and data format of the Triple-Encoder which was described in section 3.3.3.

Training data format for the Triple-Encoder is listed here. This training data is the same format with the proposition-triples for the DSR-Learner, except that we can use arbitrary event numbers here.

```
ev1 state hungry; ev1 agent ?person
ev61 act called-up; ev61 agent ?person; ev61 to friend
ev60 state wanted; ev60 agent ?person; ev60 object ev61
ev67 state had; ev67 agent ?person; ev67 object money;
ev67 mode not
ev81 act asked; ev81 agent ?person; ev81 object location;
ev81 obj-attr ?restaurant; ev81 to friend
ev84 act drove; ev84 agent ?person; ev84 to ?restaurant
ev87 act borrowed; ev87 agent ?person; ev87 object coin;
ev87 from waiter
....
....
```

The Triple-Encoder training program is listed here. Note that some of the routines are omitted in the listing. These omitted codes are duplicates (with slightly different variable names) of the routines with the same function names in the DSR-Learner program.

```
/* triple-encoder network module
   input data file: tdata-te into memory
   input symbol dictionary: global-dict (rsize-idsize), conbol-case
   output weight file: weight-te
   usage: triple-encode < sim.para > te-log    */

#include <stdio.h>
#include <math.h>

#define snapshot 50 /* snapshot at every 50 epochs */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define iosize lsize+2*rsize /* XRAAM input/output size */
#define idsize 2 /* id unit size */

/* max data size */
#define nums_obj 100 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_te 400 /* number of triples in tdata-te  */

#define dfile "tdata-te" /* input training data file */
#define sfile "global-dict" /* global-dictionary file; result of DSR-learner */
#define lfile "conbol-case" /* case-role file */
#define wfile "weight-te" /* weight matrix file */
```

```
/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is an array number pointing to the object
representations in the global-dictionary */
struct tdata_te {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_te_l[nums_td_te],*tdp1;

/* previous event rep holder  */
char preev[nsize];
float preevr[rsize];

/* XRAAM network accessory */
char repn[nsize],linkn[nsize],noden[nsize]; /* input name holder */
float rep[rsize],link[lsize],node[rsize]; /* input rep holder  */

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp;

/* simulation set up */
char mcycle[4],epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.1; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global convergence flag */
int gdone;

/******* main driver *************************/
main()
{
/* max cycles, epochs */
  int max,total,mtotal;
  int i;
  char tempev[nsize];
  float ran; /* random number output */

 /* simulation parameter setup from terminal */
  sim_setup(); /* not listed */

  max = atoi(mcycle); /* max cycles in BP  */
  mtotal = atoi(epochs); /* max epochs in training  */

  read_to_store();  /* read from initial gd and case to internal data
       structure; not listed  */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max cycles %d",max);
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log file */
  print_weight(); /* not listed */

 /* read traning data into memory; object hold -1 if embedded */
 read_td_te();

/* perform training for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* for max epoch */

  printf("\n epoch number %d\n",total);

  /* adjust bp parameter; decrease halfway at some interval  */
  for (i=1;i<5;i++)
  if (total == i*2*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

  if (total % snapshot == 0)
     printf("\n re-encoding weight freezed training verify ");

/* random id assign for object; gd is sorted so that every variable
is at front side */
symp_obj = objstorel;
while(symp_obj->name[0] == '?') {
 for (i=0;i<idsize;i++) {
    ran = rand()/32767.0;  /* generate random number between 0 and 1;
machine dependent code  */
    symp_obj->crep[i] = ran;
    }
symp_obj++;
}

/* triple encoding through all data; one epoch  */
gdone = 1; /* initial convergence flag  */

strcpy(tempev,"xxx");
tdp1 = tdata_te_l;

/* for all training data */
while(tdp1->evnum[0] != '\0') {

strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];

if (tdp1->object >= 0)  /* no recursive event */
```

```
          for (i=0;i<rsize;i++)
      node[i] = objstorel[tdp1->object].crep[i];
         else if (strcmp(tdp1->object2,preev) == 0)
  for (i=0;i<rsize;i++)
      node[i] = preevr[i];
         else printf("\error no %s\n",noden);

        if (total % snapshot == 0) prop_ev(total); /* not listed */
        else back_prop_ev(max,total); /* not listed */

        for (i=0;i<rsize;i++)
           rep[i] = hidden_ev[i];

       strcpy(tempev,repn); /* for event group checking */
       tdp1++;
} /* while */

       if (total % snapshot == 0) dump_weight();
       if (gdone == 1 && total % snapshot != 0) {
        printf("\n all data converged at epoch %d\n",total);
        break;
        }

   } /* out for */

/* post processing  */
   dump_weight(); /* not listed */

} /* main */

/******************* training data and environment *************/
/* read training data file into memory */
read_td_te()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-te */
 tdp1 = tdata_te_1;

 while(fscanf(dfp,"%s",repn) != EOF) {
   fscanf(dfp,"%s",linkn);
   fscanf(dfp,"%s",noden);

 /* load event name */
 strcpy(tdp1->evnum,repn);

 /* load object pointer  */
 i=0;
 while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] != '\0') tdp1->object = i;
 else {
   tdp1->object = -1; /* event not object */
   strcpy(tdp1->object2,noden);
 }

 /* load case rep */
 i=0;
 while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
 if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
 else tdp1->case_role = i;

 tdp1++; /* next store */
 } /* while */
 fclose(dfp);

}
```

E.3 Global-Dictionary

This section lists training data format and codes for the Global-Dictionary which was described in section 3.3.2.

Training data for the Global-Dictionary. Left side are listed symbolic names which will be converted into 10-unit ASCII representations. Right side are listed 12-unit ID + DSR representations for each word.

?class	0	0	0.2	0.1	0.2	0.2	0.3	0.4	0.8	0.8	0.1	0.4
computer	0	1	0.2	0.1	0.2	0.2	0.3	0.4	0.8	0.8	0.1	0.4
biology	1	0	0.2	0.1	0.2	0.2	0.3	0.4	0.8	0.8	0.1	0.4
?cook-utensil	0	0	0.3	0.7	0.3	0.1	0.1	0.7	0.5	0.8	0.6	0.0
pan	0	1	0.3	0.7	0.3	0.1	0.1	0.7	0.5	0.8	0.6	0.0
micro-wave	1	0	0.3	0.7	0.3	0.1	0.1	0.7	0.5	0.8	0.6	0.0
?doctor	0	0	0.3	0.4	0.5	0.8	0.6	0.2	0.7	0.9	0.1	0.9
dr-kim	0	1	0.3	0.4	0.5	0.8	0.6	0.2	0.7	0.9	0.1	0.9
dr-park	1	0	0.3	0.4	0.5	0.8	0.6	0.2	0.7	0.9	0.1	0.9
?food	0	0	0.4	0.8	0.4	0.2	0.2	0.6	0.5	0.8	0.7	0.0
steak	1	0	0.4	0.8	0.4	0.2	0.2	0.6	0.5	0.8	0.7	0.0
lobster	0	1	0.4	0.8	0.4	0.2	0.2	0.6	0.5	0.8	0.7	0.0
?guide-book	0	0	0.4	0.7	0.4	0.1	0.1	0.7	0.5	0.8	0.6	0.0
michelin-guide	0	1	0.4	0.7	0.4	0.1	0.1	0.7	0.5	0.8	0.6	0.0
yellow-page	1	0	0.4	0.7	0.4	0.1	0.1	0.7	0.5	0.8	0.6	0.0
?market	0	0	0.2	0.2	0.4	0.7	0.8	0.2	0.6	0.6	0.5	0.6
vons	0	1	0.2	0.2	0.4	0.7	0.8	0.2	0.6	0.6	0.5	0.6
lucky	1	0	0.2	0.2	0.4	0.7	0.8	0.2	0.6	0.6	0.5	0.6
?person	0	0	0.6	0.8	0.6	0.7	0.1	0.4	0.7	0.9	0.3	0.1
john	1	0	0.6	0.8	0.6	0.7	0.1	0.4	0.7	0.9	0.3	0.1
mary	0	1	0.6	0.8	0.6	0.7	0.1	0.4	0.7	0.9	0.3	0.1
?professor	0	0	0.8	1.0	0.2	0.4	0.1	0.9	0.7	0.1	0.7	0.7
simth	1	0	0.8	1.0	0.2	0.4	0.1	0.9	0.7	0.1	0.7	0.7
alan	0	1	0.8	1.0	0.2	0.4	0.1	0.9	0.7	0.1	0.7	0.7
?raw-food	0	0	0.4	0.8	0.4	0.2	0.2	0.6	0.6	0.8	0.6	0.0
chicken	1	0	0.4	0.8	0.4	0.2	0.2	0.6	0.6	0.8	0.6	0.0
fish	0	1	0.4	0.8	0.4	0.2	0.2	0.6	0.6	0.8	0.6	0.0
?restaurant	0	0	0.2	0.2	0.4	0.6	0.9	0.2	0.6	0.6	0.5	0.6
sizzler	0	1	0.2	0.2	0.4	0.6	0.9	0.2	0.6	0.6	0.5	0.6
mamasion	1	0	0.2	0.2	0.4	0.6	0.9	0.2	0.6	0.6	0.5	0.6
asked	0	0	0.3	0.8	0.4	0.9	0.1	0.3	0.8	0.5	0.3	0.8
ate	0	0	0.7	0.2	0.5	0.9	0.2	0.5	0.8	0.5	0.3	0.3
bank	0	0	0.2	0.2	0.4	0.6	0.9	0.2	0.6	0.6	0.5	0.6
bill	0	0	0.7	0.8	0.5	0.3	0.2	0.6	0.6	0.8	0.5	0.0
borrowed	0	0	0.5	0.4	0.3	0.7	0.1	0.8	0.7	0.3	0.5	0.2
bought	0	0	0.5	0.2	0.3	0.7	0.3	0.7	0.6	0.3	0.6	0.1
brought	0	0	0.7	0.4	0.5	0.9	0.3	0.1	0.6	0.6	0.1	0.3
called-up	0	0	0.4	0.5	0.3	0.9	0.2	0.6	0.7	0.7	0.6	0.1
car	0	0	0.1	0.1	0.2	0.5	0.9	0.4	0.5	0.7	0.3	0.3
cart	0	0	0.7	0.8	0.6	0.3	0.2	0.5	0.6	0.9	0.5	0.0
cashier	0	0	0.2	0.2	0.5	0.5	0.8	0.2	0.7	0.6	0.4	0.3
checked	0	0	0.7	0.5	0.5	0.9	0.2	0.3	0.6	0.6	0.3	0.1
checked-in	0	0	0.4	0.5	0.4	0.9	0.3	0.6	0.7	0.7	0.6	0.1
coin	0	0	0.5	0.5	0.4	0.1	0.1	0.6	0.5	0.8	0.5	0.0
comm-link	0	0	0.3	0.6	0.3	0.2	0.2	0.7	0.8	0.7	0.7	0.2
cooked	0	0	0.7	0.1	0.3	0.7	0.5	0.4	0.6	0.3	0.4	0.1
d-cont	0	0	0.8	0.8	0.5	0.0	0.5	0.6	0.2	1.0	0.7	1.0
d-know	0	0	0.6	0.7	0.6	0.0	0.5	0.7	0.1	0.9	0.9	1.0
d-link	0	0	0.5	0.8	0.4	0.0	0.7	0.6	0.5	0.9	0.8	1.0
d-prox	0	0	0.5	0.6	0.5	0.0	0.8	0.7	0.2	0.9	0.8	1.0
drove	0	0	0.4	0.5	0.4	0.9	0.2	0.6	0.7	0.6	0.6	0.1
entered	0	0	0.5	0.2	0.4	0.8	0.3	0.6	0.5	0.6	0.6	0.1
examined	0	0	0.7	0.8	0.5	1.0	0.2	0.2	0.7	0.8	0.2	0.3
exam-room	0	0	0.4	0.1	0.2	0.7	0.8	0.6	0.3	0.5	0.8	0.1
friend	0	0	0.2	0.2	0.5	0.8	0.8	0.2	0.5	0.6	0.4	0.8
gasoline	0	0	0.6	0.3	0.2	0.1	0.5	0.8	0.8	0.8	0.6	0.0
got	0	0	0.8	0.6	0.6	0.9	0.1	0.4	0.5	0.6	0.2	0.1
got-into	0	0	0.5	0.2	0.3	0.8	0.5	0.5	0.5	0.6	0.6	0.1
had	0	0	0.8	0.8	0.5	0.0	0.5	0.6	0.2	0.9	0.7	1.0

home	0	0	0.2	0.3	0.2	0.8	0.7	0.1	0.8	0.1	0.2	1.0
hospital	0	0	0.1	0.1	0.3	0.5	0.4	0.5	0.4	0.7	0.3	0.8
hungry	0	0	0.9	0.6	0.2	0.0	0.4	0.5	0.5	0.8	0.9	1.0
inside	0	0	0.6	0.5	0.4	0.0	0.8	0.6	0.2	0.9	0.8	1.0
kitchen	0	0	0.6	0.0	0.1	0.5	0.8	0.5	0.2	0.4	0.8	0.2
knew	0	0	0.4	0.8	0.7	0.0	0.7	0.9	0.6	0.7	0.8	1.0
left	0	0	0.8	0.6	0.6	0.9	0.1	0.4	0.5	0.6	0.3	0.1
left-for	0	0	0.4	0.7	0.2	0.9	0.1	0.5	0.9	0.2	0.3	0.9
line	0	0	0.4	0.1	0.2	0.5	0.9	0.5	0.3	0.4	0.8	0.1
listened	0	0	0.8	0.8	0.2	0.9	0.0	0.6	0.7	0.5	0.4	0.4
location	0	0	0.3	0.8	0.2	0.3	0.1	0.6	0.9	0.8	0.6	0.2
menu	0	0	0.8	0.7	0.6	0.5	0.2	0.3	0.7	0.8	0.5	0.1
money	0	0	0.5	0.4	0.2	0.1	0.1	0.9	0.8	0.7	0.6	0.1
near	0	0	0.6	0.6	0.5	0.0	0.9	0.6	0.3	1.0	0.8	1.0
needed	0	0	0.8	0.9	0.6	0.0	0.6	0.4	0.3	0.9	0.5	1.0
not	0	0	1.0	0.3	0.0	1.0	0.5	0.2	0.3	0.3	0.5	1.0
notebook	0	0	0.8	0.8	0.7	0.4	0.1	0.4	0.7	0.9	0.4	0.0
nurse	0	0	0.9	0.9	0.9	0.9	0.4	0.1	0.4	0.8	0.1	1.0
ordered	0	0	0.8	0.7	0.5	0.9	0.1	0.3	0.7	0.6	0.2	0.2
p-health	0	0	0.8	1.0	0.7	0.0	0.3	0.3	0.7	0.8	0.2	1.0
paid	0	0	0.5	0.4	0.4	0.9	0.2	0.5	0.7	0.7	0.3	0.1
pay-phone	0	0	0.2	0.2	0.5	0.6	0.8	0.2	0.6	0.7	0.5	0.4
pb-ask	0	0	0.5	0.5	0.6	0.9	0.2	0.6	0.8	0.2	0.5	0.7
pb-borrow	0	0	0.5	0.4	0.3	0.8	0.2	0.8	0.7	0.2	0.6	0.2
pb-cook	0	0	0.7	0.6	0.5	0.9	0.1	0.3	0.6	0.6	0.3	0.1
pb-doctor	0	0	0.6	0.6	0.3	0.9	0.1	0.6	0.7	0.7	0.6	0.4
pb-drive	0	0	0.4	0.5	0.3	0.9	0.2	0.6	0.7	0.6	0.6	0.1
pb-eat	0	0	0.7	0.7	0.5	0.9	0.1	0.3	0.6	0.6	0.2	0.1
pb-grasp	0	0	0.7	0.7	0.6	0.9	0.1	0.3	0.6	0.6	0.2	0.1
pb-lecture	0	0	0.9	0.8	0.2	0.8	0.0	0.6	0.7	0.1	0.2	0.5
pb-letter	0	0	0.5	0.5	0.4	0.9	0.2	0.5	0.7	0.7	0.6	0.3
pb-phone	0	0	0.5	0.5	0.4	0.9	0.2	0.5	0.7	0.7	0.6	0.3
pb-read	0	0	0.7	0.7	0.5	0.9	0.1	0.3	0.6	0.6	0.2	0.2
pb-restaurant	0	0	0.6	0.2	0.3	0.8	0.5	0.5	0.6	0.3	0.6	0.2
pb-shopping	0	0	0.5	0.2	0.3	0.8	0.4	0.7	0.7	0.3	0.6	0.1
pb-steal	0	0	0.9	0.7	0.2	0.8	0.1	0.6	0.7	0.1	0.2	0.3
pb-walk	0	0	0.4	0.4	0.4	0.9	0.2	0.5	0.6	0.6	0.5	0.1
pb-withdraw	0	0	0.6	0.3	0.3	0.7	0.2	0.8	0.7	0.2	0.7	0.1
phone-number	0	0	0.4	0.6	0.4	0.1	0.2	0.9	0.9	0.5	0.8	0.1
picked-up	0	0	0.7	0.6	0.6	0.9	0.1	0.3	0.6	0.6	0.2	0.1
read	0	0	0.7	0.6	0.6	0.9	0.1	0.3	0.6	0.6	0.2	0.1
receptionist	0	0	0.2	0.2	0.5	0.8	0.8	0.1	0.7	0.6	0.4	0.6
s-hunger	0	0	0.8	1.0	0.8	0.0	0.3	0.3	0.7	0.8	0.2	1.0
sat-down	0	0	0.5	0.1	0.3	0.8	0.5	0.4	0.5	0.5	0.5	0.1
seat	0	0	0.4	0.1	0.2	0.6	0.9	0.6	0.3	0.4	0.8	0.1
seated	0	0	0.7	0.5	0.6	0.9	0.3	0.1	0.5	0.6	0.1	0.5
sick	0	0	0.6	1.0	0.3	0.0	0.8	0.6	0.7	0.8	0.3	1.0
stole	0	0	0.8	0.6	0.2	0.7	0.1	0.6	0.7	0.1	0.2	0.3
stopped	0	0	0.6	0.1	0.3	0.8	0.5	0.4	0.5	0.5	0.6	0.1
tested	0	0	0.8	0.7	0.5	1.0	0.2	0.2	0.6	0.8	0.2	0.4
time	0	0	0.7	0.7	0.5	0.4	0.2	0.5	0.6	0.9	0.5	0.0
tip	0	0	0.7	0.7	0.6	0.3	0.2	0.5	0.6	0.9	0.5	0.0
took-note	0	0	0.6	0.7	0.7	0.8	0.3	0.4	0.8	0.4	0.3	0.4
took-out	0	0	0.7	0.7	0.6	0.9	0.2	0.3	0.6	0.6	0.2	0.1
waited	0	0	0.6	0.1	0.4	0.8	0.5	0.4	0.5	0.5	0.6	0.0
waiter	0	0	0.8	0.9	0.2	0.4	0.1	0.9	0.6	0.1	0.7	0.6
walked	0	0	0.4	0.5	0.4	0.9	0.2	0.6	0.7	0.7	0.6	0.1
wanted	0	0	0.9	0.6	0.3	0.0	0.7	0.8	0.1	1.0	0.9	1.0
went	0	0	0.5	0.5	0.3	0.9	0.1	0.7	0.7	0.7	0.7	0.2
withdrew	0	0	0.5	0.4	0.3	0.7	0.2	0.9	0.7	0.3	0.6	0.1

The Global-Dictionary training code consists of two programs: ASCII-to-DSR and DSR-to-ASCII mapping programs. Some of the duplicate routines are not listed.

ASCII-to-DSR mapping code is listed here.

```
/* global-dict network module; ascii-to-dsr
   input training data file: global-dict into memory
   output weight file: weight-a2d
   usage: gd-a2d < sim.para > a2d-log */

#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define isize_a2d nsize /* input layer size */
#define hsize_a2d 15 /* hidden layer size */
#define osize_a2d rsize /* output layer size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */

#define sfile "global-dict" /* input training data file */
#define wfile "weight-a2d" /* output weight file */

/* global dictionary -- training data  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;


/* main network */
/* current network holder */
char bankn1_a2d[nsize];
float bank1_a2d[nsize],bank2_a2d[rsize]; /* input rep holder  */

/* gd main network */
float in_a2d[isize_a2d],out_a2d[osize_a2d],hidden_a2d[hsize_a2d],
teach_a2d[osize_a2d];
float wih_a2d[isize_a2d][hsize_a2d],who_a2d[hsize_a2d][osize_a2d];
float hbias_a2d[hsize_a2d],obias_a2d[osize_a2d];

/* global file pointer  */
FILE *fopen(),*sfp,*wfp;

/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.07; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* main driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  sim_setup(); /* set up simulation parameters */
  mtotal = atoi(epochs); /* max epochs in training  */
  read_td();  /* read from initial gd as training data  */

  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log */
```

```
  print_weight(); /* not listed */

/* for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter; decrease halfway  */
  for (i=1;i<5;i++)
  if (total == i*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* for every training pair in the global-dict */
gdone = 1;
symp_obj = objstorel;
while (symp_obj->name[0] != '\0') {  /* for all the words */

  strcpy(bankn1_a2d,symp_obj->name);

/* normalize ascii for a(97) -- z(122) into 0 -- 1
   if less than 0; then ? or -  or space       */

  for (i=0;i<nsize;i++)
     bank1_a2d[i] = (float) (symp_obj->name[i] - 96) / 26.0;

  for(i=0;i<rsize;i++)
     bank2_a2d[i] = symp_obj->crep[i];


     if (total % snapshot == 0) {
prop(total); /* verify -- weight freezed */
}
     else back_prop(total);

   if (total % snapshot == 0) dump_weight();

   symp_obj++;
   } /* while */

   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight(); /* not listed */

} /* main */

/* simulation setup */
sim_setup()
{
  printf("\n how many epochs (max 99)");
  scanf("%s",epochs);
  printf("\n load previous weight (y/n)");
  scanf("%s",load_flag);
}

/* read training data file; symbolic global-dictionary is actual
training data file which is listed in this section */
read_td()
{
   int i;
   /* read to objstore */
   sfp = fopen(sfile,"r"); /* open global-dict  */
   symp_obj = objstore1;

   while(fscanf(sfp,"%s",symp_obj->name) != EOF) {

      /* read rep */
     for(i=0;i<rsize;i++)
      fscanf(sfp,"%f",&(symp_obj->crep[i]));

     symp_obj++;
   }
   fclose(sfp);


  /* test probe  */
  printf("\nobjstore1\n");
  printf("\n%s  ",objstore1[3].name);
  for(i=0;i<rsize;i++)
   printf("%.1f ",objstore1[3].crep[i]);

}

/* perform backprop */
back_prop(total)
int total;
{
  float eout[osize_a2d],ehid[hsize_a2d],dwho[hsize_a2d][osize_a2d],
dwih[isize_a2d][hsize_a2d],
mwho[hsize_a2d][osize_a2d],mwih[isize_a2d][hsize_a2d],net,pesig,
mhbias[hsize_a2d],mobias[osize_a2d],dhbias[hsize_a2d],dobias[osize_a2d];
  float sigmoid();
  int i,j,pos,cycle,flag;

  for(i=0;i<isize_a2d;i++)
   for(j=0;j<hsize_a2d;j++)
    mwih[i][j] = 0;

  for(i=0;i<hsize_a2d;i++)
   for(j=0;j<osize_a2d;j++)
    mwho[i][j] = 0;

  for (i=0;i<hsize_a2d;i++) mhbias[i] = 0;
  for (i=0;i<osize_a2d;i++) mobias[i] = 0;

/* load data */
  /* input layer */

  for(i=0;i<nsize;i++)
   in_a2d[i] = bank1_a2d[i];

 /* teach layer : hetero-associative network  */
  for(i=0;i<rsize;i++)
   teach_a2d[i] = bank2_a2d[i];

 /* back_prop procedure */

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<hsize_a2d;i++)  {
       net = 0.0;
       for(j=0;j<isize_a2d;j++)
 net = net + wih_a2d[j][i]*in_a2d[j];

 net = net + hbias_a2d[i];
 hidden_a2d[i] = sigmoid(net);
 }

   /* from hidden to output */
```

```
    for(i=0;i<osize_a2d;i++) {
      net = 0.0;
      for(j=0;j<hsize_a2d;j++)
net = net + who_a2d[j][i]*hidden_a2d[j];

net = net + obias_a2d[i];
out_a2d[i] = sigmoid(net);
}

 /* convergence check */
 for (i=0;i<osize_a2d;i++)
 if (abs(teach_a2d[i] - out_a2d[i]) > 0.22) gdone = 0;
 if (gdone == 1) printf("\n data  %s converge at epoch %d\n",
 bankn1_a2d,total);

 /* backward prop */
 /* calculate error signal for output */
   for(i=0;i<osize_a2d;i++)
    eout[i] = (teach_a2d[i] - out_a2d[i])*out_a2d[i]*(1-out_a2d[i]);

 /* modify weight for eout */
 for(i=0;i<hsize_a2d;i++)
  for(j=0;j<osize_a2d;j++) {
   dwho[i][j] = etha*eout[j]*hidden_a2d[i]+alpha*mwho[i][j];
   who_a2d[i][j] = who_a2d[i][j] + dwho[i][j];
   mwho[i][j] = dwho[i][j];
  }

 /* adjust out bias */
 for (i=0;i<osize_a2d;i++) {
   dobias[i] = etha*eout[i]*1 + alpha*mobias[i];
   obias_a2d[i] = obias_a2d[i] + dobias[i];
   mobias[i] = dobias[i];
   }

 /* calculate error signal for hidden */
 for(i=0;i<hsize_a2d;i++) {
  pesig = 0.0;
  for(j=0;j<osize_a2d;j++)
   pesig = pesig + eout[j]*who_a2d[i][j];

  ehid[i] = hidden_a2d[i]*(1-hidden_a2d[i])*pesig;
  }

 /* modify weight for ehid */
 for(i=0;i<isize_a2d;i++)
  for(j=0;j<hsize_a2d;j++) {
   dwih[i][j] = etha*ehid[j]*in_a2d[i] + alpha*mwih[i][j];
   wih_a2d[i][j] = wih_a2d[i][j] + dwih[i][j];
   mwih[i][j] = dwih[i][j];
   }

 /* adjust hid bias */
 for (i=0;i<hsize_a2d;i++) {
   dhbias[i] = etha*ehid[i]*1 + alpha*mhbias[i];
   hbias_a2d[i] = hbias_a2d[i] + dhbias[i];
   mhbias[i] = dhbias[i];
   }

} /* back_prop */

/* perform forward prop for verify */
prop(total)
int total;
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  for(i=0;i<nsize;i++)
   in_a2d[i] = bank1_a2d[i];

 /* teach layer : hetero-associative network  */
  for(i=0;i<rsize;i++)
   teach_a2d[i] = bank2_a2d[i];

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<hsize_a2d;i++)  {
       net = 0.0;
       for(j=0;j<isize_a2d;j++)
 net = net + wih_a2d[j][i]*in_a2d[j];

 net = net + hbias_a2d[i];
 hidden_a2d[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<osize_a2d;i++) {
       net = 0.0;
       for(j=0;j<hsize_a2d;j++)
 net = net + who_a2d[j][i]*hidden_a2d[j];

 net = net + obias_a2d[i];
 out_a2d[i] = sigmoid(net);
 }

  /* print network status to the log file    */
   printf("\n %d %s \n",total,bankn1_a2d);
   printf("input layer\n");
   for (i=0;i<isize_a2d;i++)
    printf("%.1f ",in_a2d[i]);
   printf("\nhidden layer\n");
   for (i=0;i<hsize_a2d;i++)
    printf("%.1f ",hidden_a2d[i]);
   printf("\noutput-teach pair\n");
   for(i=0;i<osize_a2d;i++)
    printf("%.1f ",out_a2d[i]);
   printf("\n");
   for(i=0;i<osize_a2d;i++)
    printf("%.1f ",teach_a2d[i]);
   printf("\n");

}

/* sigmoid function */
float sigmoid(x)
float x;
{
 double exp();
 return( 1.0 / (1.0 + exp(-x)));
}
```

DSR-to-ASCII mapping code is listed here.

```
/* global-dict network module; dsr-to-ascii
   input training data: global-dict into memory
   output weight file: weight-d2a
   usage: gd-d2a < sim.para > d2a-log */
```

```
#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define isize_d2a rsize /* input layer size */
#define hsize_d2a 15 /* hidden layer size */
#define osize_d2a nsize /* output layer size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */

#define sfile "global-dict" /* input training data into memory */
#define wfile "weight-d2a" /* output weight file */

/* global dictionary -- training data  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;


/* main network */
/* current network holder */
char bankn2_d2a[nsize];
float bank1_d2a[rsize],bank2_d2a[nsize]; /* input rep holder  */

/* gd main network */
float in_d2a[isize_d2a],out_d2a[osize_d2a],hidden_d2a[hsize_d2a],
teach_d2a[osize_d2a];
float wih_d2a[isize_d2a][hsize_d2a],who_d2a[hsize_d2a][osize_d2a];
float hbias_d2a[hsize_d2a],obias_d2a[osize_d2a];

/* global file pointer  */
FILE *fopen(),*sfp,*wfp;

/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.07; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* main driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  sim_setup(); /* not listed */
  mtotal = atoi(epochs); /* max epochs in training  */
  read_td();  /* read from initial gd as training data; not listedi; see
  ASCII-to-DSR module listing */

  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log */
```

```
  print_weight(); /* not listed */

/* for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter */
  for (i=1;i<5;i++)
  if (total == i*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* for every training pair in the global-dict */
gdone = 1;
symp_obj = objstorel;
while (symp_obj->name[0] != '\0') {  /* for all the words */

  strcpy(bankn2_d2a,symp_obj->name);

/* normalize ascii for a(97) -- z(122) into 0 -- 1
   if less than 0; then ? or -  or space       */
  for (i=0;i<nsize;i++)
     bank2_d2a[i] = (float) (symp_obj->name[i] - 96) / 26.0;

  for(i=0;i<rsize;i++)
     bank1_d2a[i] = symp_obj->crep[i];


     if (total % snapshot == 0) {
prop(total); /* verify -- weight frozen; not listed  */
}
     else back_prop(total); /* not listed */

   if (total % snapshot == 0) dump_weight();

   symp_obj++;
   } /* while */

   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight(); /* not listed */

} /* main */
```

E.4 Plan-Selector

This section lists the Plan-Selector code and its training data which were described in section 3.4.2.

Training data format for the Plan-Selector. This training data is for the story of type 3, skeleton 1 in appendix B.1.

```
;storytype3
;story1
IF ev1 state hungry; ev1 agent ?person
```

```
THEN g2 goal s-hunger; g2 agent ?person
AND no

IF ev61 act called-up; ev61 agent ?person; ev61 to friend
ev60 state wanted; ev60 agent ?person; ev60 object ev61
THEN p62 plan pb-phone; p62 agent ?person; p62 to friend
AND no

IF ev67 state had; ev67 agent ?person; ev67 object money;
ev67 mode not
THEN g68 goal d-cont; g68 agent ?person; g68 object money
AND no

IF ev81 act asked; ev81 agent ?person; ev81 object location;
ev81 obj-attr ?restaurant; ev81 to friend
THEN p82 plan pb-ask; p82 agent ?person; p82 object location;
p82 obj-attr ?restaurant; p82 to friend
AND yes

IF ev84 act drove; ev84 agent ?person; ev84 to ?restaurant
THEN p85 plan pb-drive; p85 agent ?person; p85 to ?restaurant
AND yes

IF ev87 act borrowed; ev87 agent ?person; ev87 object coin;
ev87 from waiter
THEN p88 plan pb-borrow; p88 agent ?person; p88 object coin;
p88 from waiter
AND yes
```

The Plan-Selector training program is listed here.

```
/* plan-selector network module
   input data file: tdata-ps into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te; load Triple-Encoder weight file
   output weight file: weight-ps
   usage: plan-select < sim.para > ps-log
   sim.para: load only epochs and loadweight_flag     */

#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 30 /* context size */
#define isize csize+rsize  /* input layer size */
#define osize rsize+1 /* output layer size */
#define iosize rsize+lsize+rsize /* Triple-Encoder IO layer size */
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ps 1000 /* number of triples in tdata-te  */
#define nums_td_in 400 /* number of internal data */

#define dfile "tdata-ps" /* input training data file */
#define sfile "global-dict" /* global dictionary file */
#define lfile "conbol-case" /* case-role representation file */
#define wfile "weight-te" /* Triple-Encoder weight file */
#define wfile2 "weight-ps" /* output weight file */

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

 /* case-role store */
 struct casestore {
    char name[nsize];
    float rep[lsize];
    } casestorel[nums_c],*casep;

 /* training data store ; each entry is an array number in the dictionary */
 struct tdata_ps {
   char evnum[nsize];
   int case_role;
   int object;
   char object2[nsize]; /* for embedded event */
   } tdata_ps_l[nums_td_ps],*tdp1;

 /* internal training data; converted from tdata_ps structure for internal use  */
 struct tdata_internal {
   char key_word[nsize];
   char evnum[nsize];
   float trep[rsize];
   } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

 /* triple encoder related network */
 /* for triple encoder and intput name holder */
 char repn[nsize],linkn[nsize],noden[nsize];
 float rep[rsize],link[lsize],node[rsize];

 /* event encoder/decoder network */
 float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
 float wih_ev[iosize][rsize],who_ev[rsize][iosize];
 float hbias_ev[rsize],obias_ev[iosize];

 /* main network */
 /* current network holder */
 char bankn1[nsize],bankn2[nsize],bankn3[nsize],bankn4[nsize];
 float bank1[csize],bank2[rsize],bank3[rsize],bank4; /* input rep holder  */
 /* plan-selector main network */
 float in[isize],out[osize],hidden[csize],teach[osize];
 float wih[isize][csize],who[csize][osize];
 float hbias[csize],obias[osize];

 /* global file pointer  */
 FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

 /* simulation set up */
 char epochs[3],load_flag[2];

 /* default BP parameter */
 float etha = 0.07; /* learining rate  */
 float alpha = 0.5; /* momentum factor */

 /* global done */
 int gdone;

 /************** main driver ********************/
 main()
 {
  /* network parameters for cycles, epochs */
   int total,mtotal;
   int i,j;
   float ran;

   sim_setup(); /* simulation para setup */
   load_weight_te(); /* triple encoder weight file */

   mtotal = atoi(epochs); /* max epochs in training  */

   read_to_store();  /* read from initial gd and case to internal data
```

```
      structure   */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight */
  else load_weight(); /* load previous weight for continuous training */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log */
  print_weight();

 /* read training data into memory; object hold -1 if embedded */
 read_td_ps();

/* for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter */
  for (i=1;i<5;i++)
  if (total == i*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* random id assign for object; gd is sorted so every variable
is up front  */
symp_obj = objstorel;
while(symp_obj->name[0] == '?') {
 for (i=0;i<idsize;i++) {
    ran = rand()/32767.0;
    symp_obj->crep[i] = ran;
    }
symp_obj++;
}

/* The training data consists of triples which should be converted into
12 unit vector representations using Triple-Encoder module */
triple_encode(total); /* internal data build up using Triple-Encoder module */

if (total % snapshot == 0) printf("\n re-encoding weight freeze ");

/* for every training pair in the tdata_internal; do main training here  */
gdone = 1;
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0')  {  /* while  3 */

tdpi2 = tdpi;
/* then part assign for bank3 and bank4  */
while(strcmp(tdpi2->key_word,"then") != 0) tdpi2++;
    if (strcmp(tdpi2->key_word,"then") == 0) {
 strcpy(bankn3,tdpi2->evnum);
 for (i=0;i<rsize;i++)  bank3[i] = tdpi2->trep[i];
 }
    else printf("\n no then part");
   tdpi2++;
   if (strcmp(tdpi2->key_word,"and") == 0) {
   strcpy(bankn4,tdpi2->evnum);
   bank4 = tdpi2->trep[0];
   }
   else printf("\n no then part");

 /* if part for bank1 and bank2 */
 while(strcmp(tdpi->key_word,"then")!= 0) { /* while 4 */
 if (strcmp(tdpi->key_word,"if") == 0 ||
     strcmp(tdpi->key_word,"follows") == 0) {
strcpy(bankn2,tdpi->evnum);
        for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
}
 else printf("\n no if part");

 if (strcmp(tdpi->key_word,"if") == 0)
     for (i=0;i<csize;i++) bank1[i] = 0;


     if (total % snapshot == 0) {
prop(total); /* verify -- weight freezed */
}
     else back_prop(total);

       for (i=0;i<csize;i++) bank1[i] = hidden[i];

      tdpi++;
} /* while  4 */

  while(strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') tdpi++;
  } /* while  3 */

   if (total % snapshot == 0) dump_weight();
   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight();

} /* main */

/*********** Triple-Encoder interface *******************************/
/* call Triple-Encoder module to covert triples into 12 unit vectors */
triple_encode(total)
int total;
{
int i,j;
/* previous event rep holder  */
char preev[nsize];
float preevr[rsize];

char tempev[nsize];

if (total == 0)
printf("\n event-encodig verify for epoch 0\n");

/* clear tdata_internal */
tdpi = tdata_internal_l;
while (tdpi->key_word[0] != '\0') {
  tdpi->key_word[0] = '\0';
  tdpi++;
  }

/* get triple representation for this id; store into internal
data file; tdata_internal  */
   tdp1 = tdata_ps_l;
   tdpi = tdata_internal_l;
   while(tdp1->evnum[0] != '\0') {  /* while 1 */

     if (strcmp(tdp1->evnum,"and") == 0) {
 strcpy(tdpi->key_word,tdp1->evnum);
 tdp1++;
 strcpy(tdpi->evnum,tdp1->evnum);
 if (strcmp(tdp1->evnum,"yes") == 0) tdpi->trep[0] = 1;
 else tdpi->trep[0] = 0; /* no */
 tdp1++;
 tdpi++;
 }
     else { /* else 1 */
         strcpy(tdpi->key_word,tdp1->evnum);
 tdp1++;
 strcpy(tempev,"xxx");
 while (strcmp(tdp1->evnum,"if") != 0 &&
strcmp(tdp1->evnum,"follows") != 0 &&
strcmp(tdp1->evnum,"then") != 0 &&
strcmp(tdp1->evnum,"and") != 0) { /* while 2 */

/* event encoding until key word */
strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];

if (tdp1->object >= 0)  /* no recursive event */
          for (i=0;i<rsize;i++)
     node[i] = objstorel[tdp1->object].crep[i];
        else if (strcmp(tdp1->object2,preev) == 0)
  for (i=0;i<rsize;i++)
     node[i] = preevr[i];
        else printf("\error no %s\n",noden);

       prop_ev(total);

       for (i=0;i<rsize;i++)
          rep[i] = hidden_ev[i];

      strcpy(tempev,repn); /* for event group checking */
      tdp1++;
} /* while 2  */

      strcpy(tdpi->evnum,repn);
      for (i=0;i<rsize;i++)
tdpi->trep[i] = rep[i];
tdpi++;
} /* else 1 */
} /* while 1 */

/* verify encoding internal data */
if (total == 0) {
printf("\n tdata_internal");
tdpi = tdata_internal_1;
for (i=0;i<50;i++) {
  printf("\n%s  %s  ",tdpi->key_word,tdpi->evnum);
  for (j=0;j<rsize;j++)
    printf("%.1f ",tdpi->trep[j]);
    tdpi++;
    }
} /* if */
} /* end of triple_encode */

/* sim parameter setup */
sim_setup()
{
 printf("\n how many epochs in simulation?(max 99)");
 scanf("%s",epochs);
 printf("\n load prvious weight? (y/n)");
 scanf("%s",load_flag);

}
/************ simulation setup bookkeeping **********/
/* print initial weight  for verification */
print_weight()
{
int i,j;

  /* for triple encoder */
   printf("wih_ev\n");
   for (i=0;i<iosize;i++) {
    for (j=0;j<rsize;j++) printf("%f ",wih_ev[i][j]); printf("\n");
     }
   printf("who_ev\n");
   for(i=0;i<rsize;i++) {
    for(j=0;j<iosize;j++) printf("%f ",who_ev[i][j]); printf("\n");
     }

   printf("hbias_ev\n");
   for (i=0;i<rsize;i++) printf("%f ",hbias_ev[i]); printf("\n");

   printf("obias_ev\n");
   for (i=0;i<iosize;i++) printf("%f ",obias_ev[i]); printf("\n");

  /* for plan-selector  */
   printf("wih\n");
   for (i=0;i<isize;i++) {
    for (j=0;j<csize;j++) printf("%f ",wih[i][j]); printf("\n");
     }
   printf("who\n");
   for(i=0;i<csize;i++) {
    for(j=0;j<osize;j++) printf("%f ",who[i][j]); printf("\n");
     }

   printf("hbias\n");
   for (i=0;i<csize;i++) printf("%f ",hbias[i]); printf("\n");

   printf("obias\n");
   for (i=0;i<osize;i++) printf("%f ",obias[i]); printf("\n");
}

/* read symbols into memory */
read_to_store()
{
   int i;

   /* read to objstore */
   sfp = fopen(sfile,"r"); /* open conbol-object  */
   symp_obj = objstore1;

   while(fscanf(sfp,"%s",symp_obj->name) != EOF) {

      /* read rep */
     for(i=0;i<rsize;i++)
      fscanf(sfp,"%f",&(symp_obj->crep[i]));

     symp_obj++;
   }
   fclose(sfp);


   /* read to case store  */
   lfp = fopen(lfile,"r"); /* open conbol-case */
   casep = casestorel;

   while(fscanf(lfp,"%s",casep->name) != EOF) {
     if (casep->name[0] != ';') {
     for(i=0;i<lsize;i++)
      fscanf(lfp,"%f",&(casep->rep[i]));
     casep++;
     }
   }
   fclose(lfp);

  /* test probe  */
  printf("\nread_to_store probe\n");
  printf("\nobjstorel\n");
  printf("\n%s  ",objstorel[3].name);
  for(i=0;i<rsize;i++)
   printf("%.1f ",objstorel[3].crep[i]);

  printf("\ncasestorel\n");
  printf("%s  ",casestorel[3].name);
  for(i=0;i<lsize;i++)
   printf("%.1f ",casestorel[3].rep[i]);
  /*   test end */

}

/* read training data into memory; transfered to internal data
using Triple-Encoder */
read_td_ps()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-ps */
 tdp1 = tdata_ps_1;

 while(fscanf(dfp,"%s",repn) != EOF) {
 if (repn[0] == ';') continue; /* skip one word comment */
 else if (strcmp(repn,"if") == 0 || /* key word */
  strcmp(repn,"follows") == 0 ||
  strcmp(repn,"then") == 0 ||
  strcmp(repn,"and") == 0 ||
  strcmp(repn,"yes") == 0 ||
  strcmp(repn,"no") == 0) {
    strcpy(tdp1->evnum,repn);
    tdp1++;
            continue;
    }
 else { /* triple */
   fscanf(dfp,"%s",linkn);
   fscanf(dfp,"%s",noden);

 /* load event name */
 strcpy(tdp1->evnum,repn);

 /* load object pointer  */
 i=0;
 while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] != '\0') tdp1->object = i;
 else {
   tdp1->object = -1; /* event not object */
   strcpy(tdp1->object2,noden);
 }
 /* load case rep */
 i=0;
 while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
 if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
 else tdp1->case_role = i;

 tdp1++; /* next store */
 } /* outer else */
 } /* while */
 fclose(dfp);

/* verify read_td */
tdp1 = tdata_ps_1;
for (i=0;i<100;i++) {
  printf("\n %s %d %d %s",tdp1->evnum,tdp1->case_role,tdp1->object,
  tdp1->object2);
  tdp1++;
  }
}

/*********** backprop and forward prop ****************/
/* perform backprop for plan-selector */
back_prop(total)
int total;
{
  float eout[osize],ehid[csize],dwho[csize][osize],
dwih[isize][csize],
mwho[csize][osize],mwih[isize][csize],net,pesig,
mhbias[csize],mobias[osize],dhbias[csize],dobias[osize];
  float sigmoid();
  int i,j,pos,cycle,flag;

  for(i=0;i<isize;i++)
   for(j=0;j<csize;j++)
    mwih[i][j] = 0;

  for(i=0;i<csize;i++)
   for(j=0;j<osize;j++)
    mwho[i][j] = 0;

  for (i=0;i<csize;i++) mhbias[i] = 0;
  for (i=0;i<osize;i++) mobias[i] = 0;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<csize;i++)
   in[pos++] = bank1[i];

  for(i=0;i<rsize;i++)
   in[pos++] = bank2[i];

 /* teach layer : hetero-associative network  */
  pos = 0;
  for(i=0;i<rsize;i++)
   teach[pos++] = bank3[i];

  teach[pos++] = bank4;

 /* back_prop procedure */

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<csize;i++)  {
       net = 0.0;
       for(j=0;j<isize;j++)
 net = net + wih[j][i]*in[j];
 net = net + hbias[i];
 hidden[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<osize;i++) {
       net = 0.0;
       for(j=0;j<csize;j++)
 net = net + who[j][i]*hidden[j];

 net = net + obias[i];
 out[i] = sigmoid(net);
 }

  /* convergence check */
  for (i=0;i<osize;i++)
  if (abs(teach[i] - out[i]) > 0.22) gdone = 0;
  if (gdone == 1) printf("\n data  %s %s %s converge at epoch %d\n",
  bankn2,bankn3,bankn4,total);

  /* backward prop */
  /* calculate error signal for output */
    for(i=0;i<osize;i++)
     eout[i] = (teach[i] - out[i])*out[i]*(1-out[i]);

  /* modify weight for eout */
  for(i=0;i<csize;i++)
   for(j=0;j<osize;j++) {
    dwho[i][j] = etha*eout[j]*hidden[i]+alpha*mwho[i][j];
    who[i][j] = who[i][j] + dwho[i][j];
    mwho[i][j] = dwho[i][j];
   }

  /* adjust out bias */
  for (i=0;i<osize;i++) {
    dobias[i] = etha*eout[i]*1 + alpha*mobias[i];
    obias[i] = obias[i] + dobias[i];
    mobias[i] = dobias[i];
    }

  /* calculate error signal for hidden */
  for(i=0;i<csize;i++) {
   pesig = 0.0;
   for(j=0;j<osize;j++)
    pesig = pesig + eout[j]*who[i][j];

   ehid[i] = hidden[i]*(1-hidden[i])*pesig;
   }

  /* modify weight for ehid */
  for(i=0;i<isize;i++)
   for(j=0;j<csize;j++) {
    dwih[i][j] = etha*ehid[j]*in[i] + alpha*mwih[i][j];
    wih[i][j] = wih[i][j] + dwih[i][j];
    mwih[i][j] = dwih[i][j];
    }

  /* adjust hid bias */
  for (i=0;i<csize;i++) {
    dhbias[i] = etha*ehid[i]*1 + alpha*mhbias[i];
    hbias[i] = hbias[i] + dhbias[i];
    mhbias[i] = dhbias[i];
    }

} /* back_prop */

/* perform forward prop for verification */
```

```
prop(total)
int total;
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<csize;i++) in[pos++] = bank1[i];
  for(i=0;i<rsize;i++) in[pos++] = bank2[i];

 /* teach layer : hetero-associative network  */
  pos = 0;
  for(i=0;i<rsize;i++) teach[pos++] = bank3[i];
  teach[pos++] = bank4;

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<csize;i++)  {
       net = 0.0;
       for(j=0;j<isize;j++)
 net = net + wih[j][i]*in[j];

 net = net + hbias[i];
 hidden[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<osize;i++) {
       net = 0.0;
       for(j=0;j<csize;j++)
 net = net + who[j][i]*hidden[j];

 net = net + obias[i];
 out[i] = sigmoid(net);
 }

  /* print net  */
   printf("\n %d %s %s %s\n",total,bankn2,bankn3,bankn4);
   printf("input layer\n");
   for (i=0;i<isize;i++) printf("%.1f ",in[i]);
   printf("\nhidden layer\n");
   for (i=0;i<csize;i++) printf("%.1f ",hidden[i]);
   printf("\noutput-teach pair\n");
   for(i=0;i<osize;i++) printf("%.1f ",out[i]);
   printf("\n");
   for(i=0;i<osize;i++) printf("%.1f ",teach[i]);
   printf("\n");

}

/* triple-encoder forward prop used for tdata_internal */
prop_ev(total)
int total;
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<rsize;i++) in_ev[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_ev[pos++] = link[i];
  for(i=0;i<rsize;i++) in_ev[pos++] = node[i];

 /* teach layer : auto-associative network  */
   for(i=0;i<iosize;i++) teach_ev[i] = in_ev[i];

 /* prop procedure */
  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
 net = net + wih_ev[j][i]*in_ev[j];

 net = net + hbias_ev[i];
 hidden_ev[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
       net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_ev[j][i]*hidden_ev[j];

 net = net + obias_ev[i];
 out_ev[i] = sigmoid(net);
 }

  /* print net for initial verify  */
   if (total  == 0) {
   printf("\n%s %s %s\n",repn,linkn,noden);
   printf("hidden\n");
   for (i=0;i<rsize;i++) printf("%.1f ",hidden_ev[i]);
     printf("\n");
   printf("output-teach pair\n");
   for(i=0;i<iosize;i++) printf("%.1f ",out_ev[i]);
   printf("\n");
   for(i=0;i<iosize;i++) printf("%.1f ",teach_ev[i]);
   printf("\n");
   }

} /* prop_ev */

float sigmoid(x)
float x;
{
 double exp();
 return( 1.0 / (1.0 + exp(-x)));
}

/* random assign for initial weights */
rassign()
{
  int i,j;

  for(i=0;i<isize;i++)
   for(j=0;j<csize;j++)
    wih[i][j] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */

  for(i=0;i<csize;i++)
   for(j=0;j<osize;j++)
    who[i][j] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */

  /* initial random bias */
  for (i=0;i<csize;i++)
    hbias[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */
   for (i=0;i<osize;i++)
     obias[i] = -1.0 + 2.0*(rand()/32767.0); /* -1 to 1 */


}
/********* output bookkeeping ****************/
/* dump weight for snapshot */
dump_weight()
{

    int i,j;

   /* dump weight for plan-selector net  */

    wfp2 = fopen(wfile2,"w");

    for (i=0;i<isize;i++) {
     for (j=0;j<csize;j++) fprintf(wfp2,"%f ",wih[i][j]);
      fprintf(wfp2,"\n");
      }

    for(i=0;i<csize;i++) {
     for(j=0;j<osize;j++) fprintf(wfp2,"%f ",who[i][j]);
      fprintf(wfp2,"\n");
      }
   /* bias dump */
    for (i=0;i<csize;i++) fprintf(wfp2,"%f ",hbias[i]);
    fprintf(wfp2,"\n");

    for (i=0;i<osize;i++) fprintf(wfp2,"%f ",obias[i]);
    fprintf(wfp2,"\n");

      fclose(wfp2);
}

/* load weight files for continous simulation */
load_weight()
{

    int i,j;

   /* load weight for plan-selector net  */

    wfp2 = fopen(wfile2,"r");

    for (i=0;i<isize;i++)
     for (j=0;j<csize;j++) fscanf(wfp2,"%f ",&wih[i][j]);

    for(i=0;i<csize;i++)
     for(j=0;j<osize;j++) fscanf(wfp2,"%f ",&who[i][j]);

   /* bias load */
    for (i=0;i<csize;i++) fscanf(wfp2,"%f ",&hbias[i]);
    for (i=0;i<osize;i++) fscanf(wfp2,"%f ",&obias[i]);
      fclose(wfp2);
}

/* load triple-encoder weight files for data conversion */
load_weight_te()
{

    int i,j;

   /* load weight for event encoder/decoder net  */
    wfp = fopen(wfile,"r");

    for (i=0;i<iosize;i++)
     for (j=0;j<rsize;j++) fscanf(wfp,"%f",&wih_ev[i][j]);
   for(i=0;i<rsize;i++)
    for(j=0;j<iosize;j++) fscanf(wfp,"%f",&who_ev[i][j]);

  /* bias load */
   for (i=0;i<rsize;i++) fscanf(wfp,"%f ",&hbias_ev[i]);
   for (i=0;i<iosize;i++) fscanf(wfp,"%f ",&obias_ev[i]);

     fclose(wfp);
}
```

E.5 GP-Associator

This section lists the GP-Associator code and its training data which were described in section 3.4.1.

Training data format for the GP-Associator.

```
;s-hunger
IF 0
AND g1 goal s-hunger; g1 agent ?person
THEN p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant

IF 1
AND g1 goal s-hunger; g1 agent ?person
THEN p4 plan pb-cook; p4 agent ?person; p4 object ?raw-food

IF 2
AND g1 goal s-hunger; g1 agent ?person
THEN p6 plan pb-eat; p6 agent ?person; p6 object ?food

IF 3
AND g1 goal s-hunger; g1 agent ?person
THEN nil

;s-restaurant
IF 0
AND p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant
THEN g8 goal d-know; g8 agent ?person; g8 object location;
g8 obj-attr ?restaurant

IF 1
AND p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant
THEN g10 goal d-cont; g10 agent ?person; g10 object money

IF 2
AND p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant
THEN g11 goal d-prox; g11 agent ?person; g11 location ?restaurant

IF 3
AND p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant
THEN nil

;d-knowrestaurant
IF 0
AND g8 goal d-know; g8 agent ?person; g8 object location;
g8 obj-attr ?restaurant
THEN p25 plan pb-ask; p25 agent ?person; p25 object location;
p25 obj-attr ?restaurant; p25 to friend

IF 1
AND g8 goal d-know; g8 agent ?person; g8 object location;
```

```
g8 obj-attr ?restaurant
THEN p29 plan pb-read; p29 agent ?person; p29 object ?guide-book

IF 2
AND g8 goal d-know; g8 agent ?person; g8 object location;
g8 obj-attr ?restaurant
THEN nil

;d-knowphonenumber
IF 0
AND g23 goal d-know; g23 agent ?person; g23 object phone-number;
g23 obj-attr friend
THEN nil

;pb-askrestaurant
IF 0
AND p25 plan pb-ask; p25 agent ?person; p25 object location;
p25 obj-attr ?restaurant; p25 to friend
THEN g25 goal d-link; g25 agent ?person; g25 to friend

IF 1
AND p25 plan pb-ask; p25 agent ?person; p25 object location;
p25 obj-attr ?restaurant; p25 to friend
THEN nil

;d-contcoin
IF 0
AND g35 goal d-cont; g35 agent ?person; g35 object coin
THEN p37 plan pb-borrow; p37 agent ?person; p37 object coin;
p37 from waiter

IF 1
AND g35 goal d-cont; g35 agent ?person; g35 object coin
THEN p37 plan pb-grasp; p37 agent ?person; p37 object coin

IF 2
AND g35 goal d-cont; g35 agent ?person; g35 object coin
THEN nil

;d-contmoney
IF 0
AND g10 goal d-cont; g10 agent ?person; g10 object money
THEN p50 plan pb-withdraw; p50 agent ?person; p50 object money;
p50 from bank

IF 1
AND g10 goal d-cont; g10 agent ?person; g10 object money
THEN p52 plan pb-steal; p52 agent ?person; p52 object money;
p52 from waiter

IF 2
AND g10 goal d-cont; g10 agent ?person; g10 object money
THEN p54 plan pb-borrow; p54 agent ?person; p54 object money;
p54 from friend

IF 3
AND g10 goal d-cont; g10 agent ?person; g10 object money
THEN nil

....
....
```

The GP-Associator program is listed here. For some of the omitted routines, see the Plan-Selector code listing.

```
/* gp-associator network module
   input training data file: tdata-gp into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te
   output weight file: weight-gp
   usage: gp-assoc < sim.para > gp-log
   sim.para: only load maxepoch, loadweight_flag */

#include <stdio.h>
#include <math.h>

#define snapshot 400 /* snapshot at every 400 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define hsize 10 /* hidden size */
#define scsize 2  /* counter size */
#define isize scsize+rsize  /* input layer size */
#define osize rsize /* output layer size */
#define iosize rsize+lsize+rsize /* triple-encoder I/O size */
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_gp 1000 /* number of triples in tdata-gp  */
#define nums_td_in 30 /* number of internal data for ONE gp */

#define dfile "tdata-gp" /* input training data file */
#define sfile "global-dict" /* symbol file for object */
#define lfile "conbol-case" /* case-role file */
#define wfile "weight-te" /* triple-encoder weight file */
#define wfile2 "weight-gp" /* output weight file */

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is an array number in the dictionary */
struct tdata_gp {
  char evnum[nsize];
  int case_role; /* hold counter value  0 to 3 too */
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_gp_l[nums_td_gp],*tdp1,*tdp2;

/* internal training data; converted using triple-encoder from tdata_gp  */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize]; /* first two array elts holds counter value */
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];
```

```
/* main network */
/* current network holder */
char bankn2[nsize],bankn3[nsize];
float bank1[scsize],bank2[rsize],bank3[rsize]; /* input rep holder  */
/* gp-assoc main network */
float in[isize],out[osize],hidden[hsize],teach[osize];
float wih[isize][hsize],who[hsize][osize];
float hbias[hsize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.1; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

/* previous event rep holder for triple-encoder  */
char preev[nsize];
float preevr[rsize];
char tempev[nsize];

  sim_setup(); /* not listed */
  load_weight_te(); /* triple encoder weight file; not listed */

  mtotal = atoi(epochs); /* max epochs in training  */

  read_to_store();  /* read from initial gd and case to internal data
       structure; not listed  */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log */
  print_weight(); /* not listed */

 /* read training data into memory; object hold -1 if embedded */
 read_td_gp();

/* for entire epochs */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter */
  for (i=1;i<5;i++)
  if (total == i*2*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* for all data in tdata-gp  */
```

```
tdp1 = tdata_gp_1;
gdone = 1; /* assume done */

while (tdp1->evnum[0] != '\0')  {  /* while 100 at the end of data */
tdp1++; /* skip coment */

/* random id assign for this gp association; assign to only var if id part
 clear in objstore; every var is up front in objstore */

/* clear previous assign */
symp_obj = objstorel;
while(symp_obj->name[0] == '?') {
  for (i=0;i<idsize;i++) symp_obj->crep[i] = 0;
  symp_obj++;
  }

tdp2 = tdp1;
tdp2++;
tdp2++;
while(strcmp(tdp2->evnum,"then") != 0) {  /* while 22 */
if (tdp2->object != -1 && objstorel[tdp2->object].name[0] == '?')
 for (i=0;i<idsize;i++) {
    ran = rand()/32767.0;
    objstorel[tdp2->object].crep[i] = ran;
    }

tdp2++;
} /*  while 22 */

/* start triple encode here */
if (total == 0)
  printf("\n triple-encode verify for epoch 0\n");

/* clear tdata_internal */
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0') {
  tdpi->key_word[0] = '\0';
  tdpi++;
  }

/* get triple representation for this id; store into internal
data file; tdata_internal  */

tdpi = tdata_internal_1;
while (tdp1->evnum[0] != ';' && tdp1->evnum[0] != '\0')  { /* while 101 */
     if (strcmp(tdp1->evnum,"if") == 0) {
       strcpy(tdpi->key_word,tdp1->evnum);
       if (tdp1->case_role == 0) { tdpi->trep[0] = 0; tdpi->trep[1] = 0; }
       else if (tdp1->case_role == 1) { tdpi->trep[0] = 0; tdpi->trep[1] = 1; }
       else if (tdp1->case_role == 2) { tdpi->trep[0] = 1; tdpi->trep[1] = 0; }
       else if (tdp1->case_role == 3) { tdpi->trep[0] = 1; tdpi->trep[1] = 1; }
       else printf("\n invalid counter ");
       tdpi++; tdp1++;
       }
     else if (strcmp(tdp1->evnum,"and") == 0 ||
      strcmp(tdp1->evnum,"then") == 0) { /* else if 1 */
         strcpy(tdpi->key_word,tdp1->evnum);
 tdp1++;

        if (strcmp(tdp1->evnum,"nil") == 0) {
   strcpy(tdpi->evnum,tdp1->evnum);
   tdp1++; tdpi++;
   }

        else {   /* else 10 */  /* triple encoding */

 strcpy(tempev,"xxx");
 while (strcmp(tdp1->evnum,"if") != 0 &&
strcmp(tdp1->evnum,"then") != 0 &&
tdp1->evnum[0] != ';' &&
tdp1->evnum[0] != '\0') { /* while 2 */

/* event encoding until key word */

strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];

if (tdp1->object >= 0)  /* no recursive event */
          for (i=0;i<rsize;i++)
     node[i] = objstorel[tdp1->object].crep[i];
        else if (strcmp(tdp1->object2,preev) == 0)
  for (i=0;i<rsize;i++)
     node[i] = preevr[i];
        else printf("\error no %s\n",noden);

       prop_ev(total); /* not listed */

       for (i=0;i<rsize;i++)
          rep[i] = hidden_ev[i];

      strcpy(tempev,repn); /* for event group checking */
      tdp1++;
} /* while 2  */

      strcpy(tdpi->evnum,repn);
      for (i=0;i<rsize;i++)
tdpi->trep[i] = rep[i];
tdpi++;
        }  /* else 10  end triple */

} /* else if 1 */

  else printf("\n input data error: no if and then format");

} /* while 101 */

/* verify encoding internal data */
if (total == 0) {
printf("\n tdata_internal");
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0') {
  printf("\n%s  %s  ",tdpi->key_word,tdpi->evnum);
  for (j=0;j<rsize;j++)
    printf("%.1f ",tdpi->trep[j]);
    tdpi++;
    }
}

/* do gp-association training */

if (total % snapshot == 0) printf("\n re-encoding weight freeze ");
```

```
/* for every training pair in the tdata_internal */
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0')  {  /* while  3 */

 /* if part for bank1 and bank2 */
 if (strcmp(tdpi->key_word,"if") == 0)  {
      for(i=0;i<scsize;i++)  bank1[i] = tdpi->trep[i]; /* first two bits */
      tdpi++;
      }
 else printf ("\n no if part ");

 if (strcmp(tdpi->key_word,"and") == 0) {
      strcpy(bankn2,tdpi->evnum);
      for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
      tdpi++;
      }
 else printf("\n no if part");

 /* then part for bank3 */

 if (strcmp(tdpi->key_word,"then") == 0) {
     strcpy(bankn3,tdpi->evnum);
     if (strcmp(tdpi->evnum,"nil") == 0)
 for (i=0;i<rsize;i++) bank3[i] = 0;
     else for (i=0;i<rsize;i++) bank3[i] = tdpi->trep[i];
     tdpi++;
     }
  else printf("\n no then part");

     if (total % snapshot == 0) {
prop(total); /* verify -- weight frozen; not listed  */
}
     else back_prop(total); /* not listed */

  } /* while  3 */


} /* while 100 for all data */

 if (total % snapshot == 0) dump_weight();

   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;  /* break for loop */
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight(); /* dump weight */

} /* main */

read_td_gp()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-gp */
 tdp1 = tdata_gp_1;

 while(fscanf(dfp,"%s",repn) != EOF) {
 if (repn[0] == ';') {  /* read in comment */
   strcpy(tdp1->evnum,repn);
   tdp1++;
   continue;
   }
 else if (strcmp(repn,"if") == 0) {
  strcpy(tdp1->evnum,repn);
```

```
          fscanf(dfp,"%d",&(tdp1->case_role)); /* read counter */
   tdp1++;
   continue;
   }
  else if (strcmp(repn,"and") == 0 ||
   strcmp(repn,"then") == 0 ||
   strcmp(repn,"nil") == 0) {
     strcpy(tdp1->evnum,repn);
     tdp1++;
            continue;
     }
  else { /* triple */
    fscanf(dfp,"%s",linkn);
    fscanf(dfp,"%s",noden);

  /* load event name */
  strcpy(tdp1->evnum,repn);

  /* load object pointer  */
  i=0;
  while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
  if (objstorel[i].name[0] != '\0') tdp1->object = i;
  else {
    tdp1->object = -1; /* event not object */
    strcpy(tdp1->object2,noden);
  }

  /* load case rep */
  i=0;
  while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
  if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
  else tdp1->case_role = i;

  tdp1++; /* next store */
  } /* outer else */
  } /* while */
  fclose(dfp);

/* verify read_td */
tdp1 = tdata_gp_1;
for (i=0;i<100;i++) {
  printf("\n %s %d %d %s",tdp1->evnum,tdp1->case_role,tdp1->object,
  tdp1->object2);
  tdp1++;
  }
}
```

E.6 Action-Generator

This section lists the Action-Generator code and its training data which were described in section 3.4.3.

Action-Generator training data format for the story type3, skeleton 1 in the appendix B.1.

```
;storytype3
;story1
IF g1 goal s-hunger; g1 agent ?person
THEN ev2 state hungry; ev2 agent ?person; ev2 mode not

IF p15 plan pb-restaurant; p15 agent ?person; p15 object ?food;
p15 location ?restaurant
THEN ev16 act ate; ev16 agent ?person; ev16 object ?food;
ev16 location ?restaurant
```

```
IF g17 goal d-know; g17 agent ?person; g17 object location;
g17 obj-attr ?restaurant
THEN ev18 state knew; ev18 agent ?person; ev18 object location;
ev18 obj-attr ?restaurant

IF g21 goal d-cont; g21 agent ?person; g21 object money
THEN ev22 state had; ev22 agent ?person; ev22 object money

IF p23 plan pb-borrow; p23 agent ?person; p23 object money;
p23 from friend
THEN ev24 act borrowed; ev24 agent ?person; ev24 object money;
ev24 from friend

IF g9 goal d-prox; g9 agent ?person; g9 location ?restaurant
THEN ev10 state inside; ev10 agent ?person; ev10 location ?restaurant

IF p11 plan pb-drive; p11 agent ?person; p11 to ?restaurant
THEN ev12 act drove; ev12 agent ?person; ev12 to ?restaurant

IF g25 goal d-link; g25 agent ?person; g25 to friend
THEN ev26 state had; ev26 agent ?person; ev26 object comm-link;
ev26 to friend

IF p27 plan pb-phone; p27 agent ?person; p27 to friend
THEN ev28 act called-up; ev28 agent ?person; ev28 to friend

IF p19 plan pb-ask; p19 agent ?person; p19 object location;
p19 obj-attr ?restaurant; p19 to friend
THEN ev20 act asked; ev20 agent ?person; ev20 object location;
ev20 obj-attr ?restaurant; ev20 to friend

IF g21 goal d-cont; g21 agent ?person; g21 object coin
THEN ev22 state had; ev22 agent ?person; ev22 object coin

IF p23 plan pb-borrow; p23 agent ?person; p23 object coin;
p23 from waiter
THEN ev24 act borrowed; ev24 agent ?person; ev24 object coin;
ev24 from waiter
```

The Action-Generator program is listed here.

```
/* action-generator network module
   input training data: tdata-ag into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te
   output weight file: weight-ag
   usage: act-gen < sim.para > ag-log
   input para: epoch, loadweight  */

#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 30 /* context size */
#define isize csize+rsize  /* input layer size */
#define osize rsize /* output layer size */
#define iosize rsize+lsize+rsize /* triple-encoder size */
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ag 1000 /* number of triples in tdata-te  */
#define nums_td_in 400 /* number of internal data */

#define dfile "tdata-ag" /* input training data file */
```

```
#define sfile "global-dict" /* symbol file for object */
#define lfile "conbol-case" /* case-role file */
#define wfile "weight-te" /* triple-encoder weight file */
#define wfile2 "weight-ag" /* output weight file */

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_ag {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_ag_l[nums_td_ag],*tdp1;

/* internal training data */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize];
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn1[nsize],bankn2[nsize],bankn3[nsize];
float bank1[csize],bank2[rsize],bank3[rsize]; /* input rep holder  */

/* action-gen main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 1.0; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* driver */
main()
{
  int total,mtotal;
```

```
   int i,j;
   float ran;

   sim_setup(); /* not listed */
   load_weight_te(); /* triple encoder weight file; not listed  */

   mtotal = atoi(epochs); /* max epochs in training  */

   read_to_store();  /* read from initial gd and case to internal data
        structure; not listed    */
   if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
   else load_weight(); /* load previous weight for continuous training; not listed  */

   /* print sim environment */
   printf("\n max epochs %d",mtotal);
   printf("\n load_flag %s",load_flag);
   printf("\n BP parameter %f %f",etha,alpha);

  /* print initial weight to the log */
   print_weight(); /* not listed */

  /* read training data into memory; object hold -1 if embedded */
  read_td_ag();

 /* for entire epochs */
 for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

   if (total % snapshot == 0) printf("\n epoch number %d\n",total);

   /* adjust bp parameter */
   for (i=1;i<5;i++)
   if (total == i*snapshot) {
      etha = etha / 2;
      printf("\n BP parameter set up %f %f",etha,alpha);
      break;
      }

 /* random id assign for object; gd is sorted so every variable
 is up front  */
 symp_obj = objstore1;
 while(symp_obj->name[0] == '?') {
  for (i=0;i<idsize;i++) {
     ran = rand()/32767.0;
     symp_obj->crep[i] = ran;
     }
 symp_obj++;
 }

 triple_encode(total); /* internal data build up; not listed; see
 Plan-Selector listing  */

 if (total % snapshot == 0) printf("\n re-encoding weight freeze ");

 /* for every training pair in the tdata_internal */
 gdone = 1;
 tdpi = tdata_internal_1;
 while (tdpi->key_word[0] != '\0')  {  /* while  3 */

 /* if part assign for bank2  */
    if (strcmp(tdpi->key_word,"if") == 0) {
  strcpy(bankn2,tdpi->evnum);
  for (i=0;i<rsize;i++)  bank2[i] = tdpi->trep[i];
  }
    else printf("\n no if  part");

   tdpi++;

  /* then  part for bank1 and bank2 */
  while(strcmp(tdpi->key_word,"if")!= 0 && tdpi->key_word[0] != '\0') { /* while 4 */
 if (strcmp(tdpi->key_word,"then") == 0 ||
     strcmp(tdpi->key_word,"follows") == 0) {
strcpy(bankn3,tdpi->evnum);
        for (i=0;i<rsize;i++) bank3[i] = tdpi->trep[i];
}
 else printf("\n no then part");

 if (strcmp(tdpi->key_word,"then") == 0)
     for (i=0;i<csize;i++) bank1[i] = 0;

     if (total % snapshot == 0) {
prop(total); /* verify -- weight freezed; not listed  */
}
     else back_prop(total); /* not listed */

       for (i=0;i<csize;i++) bank1[i] = hidden[i];

      tdpi++;
} /* while  4 */

  } /* while  3 */

  if (total % snapshot == 0) dump_weight(); /* not listed */

   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight(); /* not listed */

} /* main */

read_td_ag()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-ag */
 tdp1 = tdata_ag_1;

 while(fscanf(dfp,"%s",repn) != EOF) {
 if (repn[0] == ';') continue; /* skip one word comment */
 else if (strcmp(repn,"if") == 0 || /* key word */
  strcmp(repn,"follows") == 0 ||
  strcmp(repn,"then") == 0) {
    strcpy(tdp1->evnum,repn);
    tdp1++;
            continue;
    }
 else { /* triple */
   fscanf(dfp,"%s",linkn);
   fscanf(dfp,"%s",noden);

 /* load event name */
 strcpy(tdp1->evnum,repn);

 /* load object pointer  */
 i=0;
 while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] != '\0') tdp1->object = i;
 else {
   tdp1->object = -1; /* event not object */
   strcpy(tdp1->object2,noden);
 }

 /* load case rep */
```

```
  i=0;
  while((strcmp(linkn,casestorel[i].name) != 0) && (casestorel[i].name[0] != '\0')) i++;
  if (casestorel[i].name[0] == '\0') printf("\n error: no %s",linkn);
  else tdp1->case_role = i;

  tdp1++; /* next store */
  } /* outer else */
  } /* while */
  fclose(dfp);

/* verify read_td */
tdp1 = tdata_ag_1;
for (i=0;i<100;i++) {
   printf("\n %s %d %d %s",tdp1->evnum,tdp1->case_role,tdp1->object,
   tdp1->object2);
   tdp1++;
   }
}
```

E.7 ST-Parser

This section lists the ST-Parser code and its training data which were described in section 3.5.1.

Training data format for the ST-Parser.

```
;story31
IF ?person FOLLOWS hungry
THEN - AND hungry AND ?person AND - AND - AND - AND - AND - AND -
AND - AND - AND -

IF ?person FOLLOWS asked FOLLOWS friend FOLLOWS ?restaurant
FOLLOWS location
THEN asked AND - AND ?person AND location AND - AND ?restaurant AND -
AND - AND friend AND - AND - AND -

IF ?person FOLLOWS drove FOLLOWS ?restaurant
THEN drove AND - AND ?person AND - AND - AND - AND - AND
?restaurant AND - AND - AND -

IF ?person FOLLOWS had FOLLOWS not FOLLOWS money
THEN - AND had AND ?person AND moeny AND - AND - AND - AND - AND -
AND - AND - AND not

IF ?person FOLLOWS called-up FOLLOWS friend
THEN called-up AND - AND ?person AND - AND - AND - AND - AND -
AND friend AND - AND - AND -

IF ?person FOLLOWS wanted FOLLOWS ev?
THEN - AND wanted AND ?person AND ev? AND - AND - AND - AND -
AND - AND - AND - AND -

IF ?person FOLLOWS borrowed FOLLOWS coin FOLLOWS waiter
THEN borrowed AND - AND ?person AND coin AND - AND - AND - AND
waiter AND - AND - AND - AND -
```

The ST-Parser program is listed here.

```
/* st-parser network module
   input training data: tdata-stp into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te
   output weight file: weight-stp
   usage: stpar < sim.para > stp-log */

#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 12*rsize /* context size */
#define isize csize+rsize /* input layer size */
#define osize 12*rsize /* output layer size */
#define iosize rsize+lsize+rsize /* triple-encoder I/O size */
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_stp 400 /* number of triples in tdata-stp  */

#define dfile "tdata-stp" /* input training data file */
#define sfile "global-dict" /* symbol table for object */
#define lfile "conbol-case" /* case-role table */
#define wfile "weight-te" /* triple-encoder weight; for embedded sentences */
#define wfile2 "weight-stp" /* output weight file */

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is an array number in the dictionary */
struct tdata_stp {
  char key_word[nsize];
  int object;
  char object2[nsize]; /* hold symbol not in gd */
  } tdata_stp_l[nums_td_stp],*tdp1,*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* triple encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn[14][nsize];
float bank0[csize],bank[14][rsize]; /* input rep holder  */
/* stparser main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;
```

```
/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.07; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  sim_setup(); /* not listed */
  load_weight_te(); /* triple encoder weight file; not listed  */

  mtotal = atoi(epochs); /* max epochs in training  */

  read_to_store();  /* read from initial gd and case to internal data
       structure; not listed    */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log  */
  print_weight(); /* not listed */

 /* read training data into memory; object hold -1 if embedded */
 read_td_ps();

/* for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter */
  for (i=1;i<5;i++)
  if (total == i*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* random id assign for object; gd is sorted so every variable
is up front  */
symp_obj = objstore1;
while(symp_obj->name[0] == '?') {
 for (i=0;i<idsize;i++) {
    ran = rand()/32767.0;
    if (ran < 0.51) symp_obj->crep[i] = ran;
    }
symp_obj++;
}

/* for every training pair in the tdata_stp  */
gdone = 1;
tdpi = tdata_stp_1;
while (tdpi->key_word[0] != '\0')  {  /* while  3 */

/* then part assign for bank2 to bank13  */
tdpi2 = tdpi;
while(strcmp(tdpi2->key_word,"then") != 0) tdpi2++;
    if (strcmp(tdpi2->key_word,"then") == 0)
     for (j=2;j<14;j++) {

        if (tdpi2->object >= 0) {
 strcpy(bankn[j],objstorel[tdpi2->object].name);
 for (i=0;i<rsize;i++)  bank[j][i] = objstorel[tdpi2->object].crep[i];
 }
        else { /* else13 */
  strcpy(bankn[j],tdpi2->object2);
  for (i=0;i<rsize;i++) bank[j][i] = 0;
  } /* else13 */

         tdpi2++;
 } /* for */
    else printf("\n no then part");

 /* if part for bank0 and bank1 */
 while(strcmp(tdpi->key_word,"then") != 0) { /* while 4; recirculate loop */
 if (strcmp(tdpi->key_word,"if") == 0 ||
     strcmp(tdpi->key_word,"follows") == 0) {

       if (tdpi->object >=  0) {
strcpy(bankn[1],objstorel[tdpi->object].name);
        for (i=0;i<rsize;i++) bank[1][i] = objstorel[tdpi->object].crep[i];
}
       else { /* else14 */
       strcpy(bankn[1],tdpi->object2);
       for (i=0;i<rsize;i++) bank[1][i] = 0;
       } /* else 14 */
       }
 else printf("\n no if part");

 if (strcmp(tdpi->key_word,"if") == 0)
     for (i=0;i<csize;i++) bank0[i] = 0;


     if (total % snapshot == 0) {
prop(total); /* verify -- weight freezed; not listed  */
}
     else back_prop(total); /* not listed */

       for (i=0;i<csize;i++) bank0[i] = hidden[i];

      tdpi++;
} /* while  4 */

  while(strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') tdpi++;
  } /* while  3 */

   if (total % snapshot == 0) dump_weight(); /* not listed */
   if (gdone == 1 && total % snapshot != 0) {
     printf("\n all data converge at epoch %d\n",total);
     break;
     }

 } /* FOR MAX EPOCH  */

/* post processing  */
   dump_weight();

} /* main */


read_td_ps()
{
 int i;

 dfp = fopen(dfile,"r"); /* open tdata-stp */
 tdp1 = tdata_stp_1;

/* use repn as temp holder */
 while(fscanf(dfp,"%s",repn) != EOF) {
 if (repn[0] == ';') continue; /* skip one word comment */
 else if (strcmp(repn,"if") == 0 || /* key word */
  strcmp(repn,"follows") == 0 ||
  strcmp(repn,"then") == 0 ||
  strcmp(repn,"and") == 0 ||
  strcmp(repn,"yes") == 0 ||
  strcmp(repn,"no") == 0) { /* else 2 */
   strcpy(tdp1->key_word,repn);

   fscanf(dfp,"%s",noden);

 /* load object pointer  */
 i=0;
 while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
 if (objstorel[i].name[0] != '\0') tdp1->object = i;
 else {
   tdp1->object = -1; /* event or bar not object */
   strcpy(tdp1->object2,noden);
 }

} /* else 2 */

 tdp1++; /* next store */

 } /* while */

 fclose(dfp);

/* verify read_td */
tdp1 = tdata_stp_1;
while(tdp1->key_word[0] != '\0') {
  printf("\n %s %d %s",tdp1->key_word,tdp1->object,tdp1->object2);
  tdp1++;
  }
}
```

E.8 TS-Generator

This section lists the TS-Generator code and its training data which were described in section 3.5.2.

Training data format for the TS-Generator

```
;story31
IF - AND hungry AND ?person AND - AND - AND - AND - AND - AND - AND -
AND - AND not
THEN ?person FOLLOWS was FOLLOWS not FOLLOWS hungry

IF ate AND - AND ?person AND ?food AND - AND - AND - AND - AND -
AND ?restaurant AND - AND -
THEN ?person FOLLOWS ate FOLLOWS ?food FOLLOWS at FOLLOWS ?restaurant

IF - AND knew AND ?person AND location AND - AND ?restaurant AND - AND -
AND - AND - AND - AND -
THEN ?person FOLLOWS knew FOLLOWS ?restaurant FOLLOWS location

IF asked AND - AND ?person AND location AND - AND ?restaurant
AND - AND - AND friend AND - AND - AND -
THEN ?person FOLLOWS asked FOLLOWS friend FOLLOWS about
FOLLOWS ?restaurant FOLLOWS location
```

```
IF - AND inside AND ?person AND - AND - AND - AND - AND - AND -
AND ?restaurant AND - AND -
THEN ?person FOLLOWS was FOLLOWS inside FOLLOWS ?restaurant

IF drove AND - AND ?person AND - AND - AND - AND - AND - AND
?restaurant AND - AND - AND -
THEN ?person FOLLOWS drove FOLLOWS to FOLLOWS ?restaurant

IF - AND had AND ?person AND money AND - AND - AND - AND - AND - AND -
AND - AND -
THEN ?person FOLLOWS had FOLLOWS money

IF borrowed AND - AND ?person AND money AND - AND - AND - AND friend
AND - AND - AND - AND -
THEN ?person FOLLOWS borrowed FOLLOWS money FOLLOWS from FOLLOWS friend

IF - AND had AND ?person AND comm-link AND - AND - AND - AND - AND
friend AND - AND - AND -
THEN ?person FOLLOWS had FOLLOWS comm-link FOLLOWS to
FOLLOWS friend

IF called-up AND - AND ?person AND - AND - AND - AND - AND -
AND friend AND - AND - AND -
THEN ?person FOLLOWS called-up FOLLOWS friend

IF - AND had AND ?person AND coin AND - AND - AND - AND - AND - AND -
AND - AND -
THEN ?person FOLLOWS had FOLLOWS coin

IF borrowed AND - AND ?person AND coin AND - AND - AND - AND waiter
AND - AND - AND - AND -
THEN ?person FOLLOWS borrowed FOLLOWS coin FOLLOWS from FOLLOWS waiter
```

The TS-Generator program is listed here.

```
/* ts-generator network module
   input training data: tdata-tsg into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te
   output weight file: weight-tsg
   usage: tsgen < sim.para > tsg-log */

#include <stdio.h>
#include <math.h>

#define snapshot 200 /* snapshot at every 200 epoch */
#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 12*rsize /* context size */
#define isize csize+12*rsize /* input layer size */
#define osize rsize /* output layer size */
#define iosize rsize+lsize+rsize /* triple-encoder I/O size */
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_tsg 400 /* number of triples in tdata-stp  */

#define dfile "tdata-tsg" /* input training data file */
#define sfile "global-dict" /* symbol table for object */
#define lfile "conbol-case" /* case-role table */
#define wfile "weight-te" /* triple-encoder weight */
#define wfile2 "weight-tsg" /* output weight */
```

```
/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_tsg {
  char key_word[nsize];
  int object;
  char object2[nsize]; /* hold symbol not in gd */
  } tdata_tsg_l[nums_td_tsg],*tdp1,*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* triple encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn[14][nsize];
float bank0[csize],bank[14][rsize]; /* input rep holder  */
/* stparser  main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* simulation set up */
char epochs[3],load_flag[2];

/* default BP parameter */
float etha = 0.07; /* learining rate  */
float alpha = 0.5; /* momentum factor */

/* global done */
int gdone;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  sim_setup(); /* not listed */
 load_weight_te();  /* triple encoder weight file; not listed  */

  mtotal = atoi(epochs); /* max epochs in training  */

  read_to_store();  /* read from initial gd and case to internal data
       structure; not listed    */
  if (load_flag[0] == 'n') rassign();  /* random assign of weight; not listed  */
  else load_weight(); /* load previous weight for continuous training; not listed  */

  /* print sim environment */
  printf("\n max epochs %d",mtotal);
  printf("\n load_flag %s",load_flag);
  printf("\n BP parameter %f %f",etha,alpha);

 /* print initial weight to the log  */
  print_weight();  /* not listed */

 /* read training data into memory; object hold -1 if embedded */
 read_td_ps();

/* for entire epoch */
for (total=0;total<mtotal+1;total++)  { /* FOR MAX EPOCH */

  if (total % snapshot == 0) printf("\n epoch number %d\n",total);

  /* adjust bp parameter */
  for (i=1;i<5;i++)
  if (total == i*snapshot) {
     etha = etha / 2;
     printf("\n BP parameter set up %f %f",etha,alpha);
     break;
     }

/* random id assign for object; gd is sorted so every variable
is up front  */
symp_obj = objstorel;
while(symp_obj->name[0] == '?') {
 for (i=0;i<idsize;i++) {
    ran = rand()/32767.0;
    symp_obj->crep[i] = ran;
    }
symp_obj++;
}

/* for every training pair in the tdata_tsg  */
gdone = 1;
tdpi = tdata_tsg_l;
while (tdpi->key_word[0] != '\0')  {  /* while  3 */

/* if part for bank1 to bank12 */
if (strcmp(tdpi->key_word,"if") == 0)
  for (j=1;j<13;j++) {

        if (tdpi->object >= 0) {
 strcpy(bankn[j],objstorel[tdpi->object].name);
 for (i=0;i<rsize;i++)  bank[j][i] = objstorel[tdpi->object].crep[i];
 }
        else { /* else13 */
  strcpy(bankn[j],tdpi->object2);
  for (i=0;i<rsize;i++) bank[j][i] = 0;
  } /* else13 */

    tdpi++;
    }
else printf("\n no if part \n");

/* then part for bank13 */
while (strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') { /* while 4 */

if (strcmp(tdpi->key_word,"then") == 0 ||
    strcmp(tdpi->key_word,"follows") == 0)

       if (tdpi->object >=  0) {
strcpy(bankn[13],objstorel[tdpi->object].name);
        for (i=0;i<rsize;i++) bank[13][i] = objstorel[tdpi->object].crep[i];
}
       else { /* else14 */
       strcpy(bankn[13],tdpi->object2);
```

```
        for (i=0;i<rsize;i++) bank[13][i] = 0;
        } /* else 14 */

 else printf("\n no then part\n");

  if (strcmp(tdpi->key_word,"then") == 0)
      for (i=0;i<csize;i++) bank0[i] = 0;


      if (total % snapshot == 0)
 prop(total); /* verify -- weight frozen; not listed  */
      else back_prop(total); /* not listed */

        for (i=0;i<csize;i++) bank0[i] = hidden[i];

     tdpi++;

 } /* while  4 */

   } /* while  3 */

    if (total % snapshot == 0) dump_weight(); /* not listed */
    if (gdone == 1 && total % snapshot != 0) {
      printf("\n all data converge at epoch %d\n",total);
      break;
      }

  } /* FOR MAX EPOCH  */

 /* post processing  */
    dump_weight();

 } /* main */


 read_td_ps()
 {
  int i;

  dfp = fopen(dfile,"r"); /* open tdata-tsg */
  tdp1 = tdata_tsg_1;

 /* use repn as temp holder */
  while(fscanf(dfp,"%s",repn) != EOF) {
  if (repn[0] == ';') continue; /* skip one word comment */
  else if (strcmp(repn,"if") == 0 || /* key word */
   strcmp(repn,"follows") == 0 ||
   strcmp(repn,"then") == 0 ||
   strcmp(repn,"and") == 0 ||
   strcmp(repn,"yes") == 0 ||
   strcmp(repn,"no") == 0) { /* else 2 */
    strcpy(tdp1->key_word,repn);

    fscanf(dfp,"%s",noden);

  /* load object pointer  */
  i=0;
  while((strcmp(noden,objstorel[i].name) != 0) && (objstorel[i].name[0] != '\0')) i++;
  if (objstorel[i].name[0] != '\0') tdp1->object = i;
  else {
    tdp1->object = -1; /* event or bar not object */
    strcpy(tdp1->object2,noden);
  }

 } /* else 2 */

  tdp1++; /* next store */

  } /* while */
 fclose(dfp);

/* verify read_td */
tdp1 = tdata_tsg_l;
while(tdp1->key_word[0] != '\0') {
  printf("\n %s %d %s",tdp1->key_word,tdp1->object,tdp1->object2);
  tdp1++;
  }
}
```

APPENDIX F

DYNASTY performance code and data

This chapter lists DYNASTY performance code and data files which were described in chapter 4 and in section 5.2.

F.1 DYNASTY datafile

DYNASTY should be loaded with several files during performance phase including the weight-files which were outputs of DYNSATY training modules. The necessary weight-file names are listed in the source code.

The case-role representation file.

```
;event
state          1 0 0 0 0 0 0 0 0 0 0 0
act            0 1 0 0 0 0 0 0 0 0 0 0
agent          0 0 1 0 0 0 0 0 0 0 0 0
object         0 0 0 1 0 0 0 0 0 0 0 0
obj-attr       0 0 0 0 1 0 0 0 0 0 0 0
instrument     0 0 0 0 0 1 0 0 0 0 0 0
co-obj         0 0 0 0 0 0 1 0 0 0 0 0
from           0 0 0 0 0 0 0 1 0 0 0 0
to             0 0 0 0 0 0 0 0 1 0 0 0
location       0 0 0 0 0 0 0 0 0 1 0 0
time           0 0 0 0 0 0 0 0 0 0 1 0
mode           0 0 0 0 0 0 0 0 0 0 0 1
;goal/plan
goal           1 0 0 0 0 0 0 0 0 0 0 0
plan           0 1 0 0 0 0 0 0 0 0 0 0
```

The input story file format is listed below. Every story consists of several processing blocks.

```
gp ; say it is a goal/plan-based story
;for story 31

block:
ev1 state hungry; ev1 agent john

block:
ev81 act asked; ev81 agent john; ev81 object location;
ev81 obj-attr sizzler; ev81 to friend

block:
ev84 act drove; ev84 agent john; ev84 to sizzler

block:
ev67 state had; ev67 agent john; ev67 object money;
ev67 mode not

block:
ev61 act called-up; ev61 agent john; ev61 to friend;
ev60 state wanted; ev60 agent john; ev60 object ev61
```

```
block:
ev87 act borrowed; ev87 agent john; ev87 object coin;
ev87 from waiter

script ; script-based story
;story41

block:
ev1 state hungry; ev1 agent mary

block:
ev99 act entered; ev99 agent mary; ev99 location sizzler
FOLLOWS ev104 act ate; ev104 agent mary; ev104 object steak
FOLLOWS ev106 act left; ev106 agent mary; ev106 object tip

....
....
```

F.2 The DYNASTY program

The DYNASTY performance program is listed here.

```
/* dynastyv2.c performance phase

   input: block stream of event sequence (output of st-parser)
(from inputfile)
   output: goal/plan inference chain for each event (input to the ts-generator)
(produce dynastyv2-log)

   loaded weights: weight-te, weight-ps, weight-ag, weight-gp, weight-a2d,
   weight-d2a, weight-stp, weight-tsg

   input symbol file: global-dict (rsize: 12 units), conbol-case
      (global-dict contains variables and new (untrained) instances)

   searchspace: wmemory array

   algorithm:
   0. symbol to vector rep
   1. triple encoding
   2. goal/plan selecting
   3. gp search (BFS) <-- gp triple decoding, start and target
   4. matching and binding propagation while generating chains
   5. action generation for each goal/plan in the chain
   6. event triple decoding
   7. vector to symbol

   usage:  dynastv2 < inputfile > d-log

   inputfile: script flag and triple group start with "block:" keyword
   script blocks have "follows" keyword in them
   */

#include <stdio.h>
#include <math.h>

/* general network size */
#define nsize 20 /* number of max characters in symbol */
#define rsize 12 /* vector rep size */
#define lsize 12 /* case-role size */
#define idsize 2 /* id bit size */
#define csize 30 /* context bank size */
#define scsize 2 /* self-increasing counter bank size */
```

```
/* triple-encoder */
#define iosize rsize+lsize+rsize

/* plan-selector */
#define pssizei csize+rsize
#define pssizeh csize
#define pssizeo rsize+1

/* gp-associator */
#define gpsizei scsize+rsize
#define gpsizeh 10
#define gpsizeo rsize

/* action-generator */
#define agsizei csize+rsize
#define agsizeh csize
#define agsizeo rsize

/* max data size */
#define nums_obj 200 /* number of object symbol; var + instance */
#define nums_c 30 /* number of case role */
#define nums_td 100 /* number of triples in the input story */

/* global dictionary */
struct objstore {
 char name[nsize];
 float crep[rsize];
 } objstorel[nums_obj],*symp_obj;

/* case-role store */
struct casestore {
 char name[nsize];
 float rep[lsize];
 } casestorel[nums_c],*casep;

/* network structure */
/* triple encoder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* plan-selector */
char bankn2_ps[nsize],bankn3_ps[nsize];
float bank1_ps[csize],bank2_ps[rsize],bank3_ps[rsize],bank4_ps;
float in_ps[pssizei],out_ps[pssizeo],hidden_ps[pssizeh];
float wih_ps[pssizei][pssizeh],who_ps[pssizeh][pssizeo];
float hbias_ps[pssizeh],obias_ps[pssizeo];

/* gp-associator */
char bankn2_gp[nsize],bankn3_gp[nsize];
float bank1_gp[scsize],bank2_gp[rsize],bank3_gp[rsize];
float in_gp[gpsizei],out_gp[gpsizeo],hidden_gp[gpsizeh];
float wih_gp[gpsizei][gpsizeh],who_gp[gpsizeh][gpsizeo];
float hbias_gp[gpsizeh],obias_gp[gpsizeo];

/* action-generator */
char bankn2_ag[nsize],bankn3_ag[nsize];
float bank1_ag[csize],bank2_ag[rsize],bank3_ag[rsize];
float in_ag[agsizei],out_ag[agsizeo],hidden_ag[agsizeh];
float wih_ag[agsizei][agsizeh],who_ag[agsizeh][agsizeo];
float hbias_ag[agsizeh],obias_ag[agsizeo];

/* global file pointer */
FILE *fopen(),*sfp,*lfp,*wfp;

/* input story structure */
```

```
struct inputstory {
   char evnum[nsize];
   int case_role;
   int object;
   char object2[nsize]; /* for embedded event */
   } inputstory1[100],*inp;

/* working memory structure; search node in the gp-tree   */
struct gpnode {
 char noden[nsize];
 float vecrep[rsize];
 int parent; /* parent node array number */
 char evnum[10][nsize]; /* max 10 triples for each vecrep */
 char casen[10][nsize];
 char objn[10][nsize];
};

struct gpnode wmemory[128]; /* max search depth 7; 2**7 = 128 */
struct gpnode gpchain[7]; /* output gp chain for single event */
struct gpnode startn, targetn; /* start and target node for gp search */
struct gpnode decodeout; /* global temp holder for decode_ev */

int script; /* script flag */
int wmin,wmout,gpout,count; /* global index */

/**************** dynasty main driver **********************/
main()
{
int i;

/* previous event rep holder for event-encoding  */
char preev[nsize];
float preevr[rsize];
char tempev[nsize];

/* running setup */
  load_weight(); /* load knowledge for triple-encoder, plan-selector
    action-generator, gp-associator */
  read_to_store(); /* read global-dict and conbol-case */
  print_weight(); /* verify weight correctly loaded */
  read_story(); /* read single story into inputstory structure */

 /* process story */
 /* clear startn */
    clear_gpnode(startn);

  inp = inputstory1;

/* for all the blocks in the story; processed block by block */
while (inp->evnum[0] != '\0')  {  /* WHILE 1; to the end of the story */

   /* clear wmemory */
   for (i=0;i<128;i++)
     clear_gpnode(wmemory[i]);

   /* clear gpchain */
   for (i=0;i<7;i++)
     clear_gpnode(gpchain[i]);

   inp++; /* skip "block:" key word */

  /*  event-encoding: get event representation  */
  printf("\n event encoding start...\n");

  strcpy(tempev,"xxx");
  while(strcmp(inp->evnum,"block:") != 0 && inp->evnum[0] != '\0') { /* while 2 */
strcpy(repn,inp->evnum);
strcpy(linkn,casestore1[inp->case_role].name);
```

```
if (inp->object >= 0) /* object not embedded */
strcpy(noden,objstorel[inp->object].name);
else
strcpy(noden,inp->object2); /* event; embedded */


/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[inp->case_role].rep[i];

if (inp->object >= 0)  /* no recursive event */
          for (i=0;i<rsize;i++)
     node[i] = objstorel[inp->object].crep[i];
         else if (strcmp(inp->object2,preev) == 0)
  for (i=0;i<rsize;i++)
     node[i] = preevr[i];
         else printf("\error no %s\n",noden);

       prop_ev();

       for (i=0;i<rsize;i++)
          rep[i] = hidden_ev[i];

      strcpy(tempev,repn); /* for event group checking */

inp++;
} /* while 2  */

   /* event-encoding output in repn and rep */
     printf("\n %s representation ",repn);
     for (i=0;i<rsize;i++)
printf("%.1f ",rep[i]);

  /* do plan-selection */
  printf("\n\n plan selecting start..\n");
   for(i=0;i<csize;i++) bank1_ps[i] = 0;
   strcpy(bankn2_ps,repn);
   for(i=0;i<rsize;i++) bank2_ps[i] = rep[i];

   prop_ps();

   /* output is in bank3_ps, bank4_ps */
   for(i=0;i<rsize;i++) bank3_ps[i] = out_ps[i];
   bank4_ps = out_ps[rsize]; /* 1=YES, 0=No  */

   printf("\n decoding selected plan \n");
   decode_ev(bank3_ps); /* decode ev,g,p vector rep into decodeout
   structure  */

   /* do gp-association; gp tree expanding and search */
   printf("\n gp-tree expanding start....\n");

    /* if first event */
   if (startn.noden[0] == '\0') { /* top-level goal */
     /* just prepare startn and proceed to act-gen */
     printf("\n top-level goal\n");

     /* cpy structure decodeout to startn */
  strcpy(startn.noden,decodeout.noden);
  for(i=0;i<rsize;i++) startn.vecrep[i] = decodeout.vecrep[i];
  startn.parent = decodeout.parent;
  for (i=0;i<10;i++) {
    strcpy(startn.evnum[i],decodeout.evnum[i]);
    strcpy(startn.casen[i],decodeout.casen[i]);
    strcpy(startn.objn[i],decodeout.objn[i]);
    }


/* copy startn to gpchain[0] */
  strcpy(gpchain[0].noden,startn.noden);
  for(i=0;i<rsize;i++) gpchain[0].vecrep[i] = startn.vecrep[i];
  gpchain[0].parent = startn.parent;
  for (i=0;i<10;i++) {
    strcpy(gpchain[0].evnum[i],startn.evnum[i]);
    strcpy(gpchain[0].casen[i],startn.casen[i]);
    strcpy(gpchain[0].objn[i],startn.objn[i]);
    }
     printgpnode(gpchain[0]);
    }

   else {  /* else 1 ; gp-tree expanding */

      /* prepare targetn */
  strcpy(targetn.noden,decodeout.noden);
  for(i=0;i<rsize;i++) targetn.vecrep[i] = decodeout.vecrep[i];
  targetn.parent = decodeout.parent;
  for (i=0;i<10;i++) {
    strcpy(targetn.evnum[i],decodeout.evnum[i]);
    strcpy(targetn.casen[i],decodeout.casen[i]);
    strcpy(targetn.objn[i],decodeout.objn[i]);
    }


     printf("\n startn:\n");
     printgpnode(startn);
     printf("\n targetn: \n");
     printgpnode(targetn);

   /* expand gp tree until match to targetn */
     wmout = 0;
     wmin = 0;

     /* cpy startn to wmemory[0] */
  strcpy(wmemory[0].noden,startn.noden);
  for(i=0;i<rsize;i++) wmemory[0].vecrep[i] = startn.vecrep[i];
  wmemory[0].parent = startn.parent;
  for (i=0;i<10;i++) {
    strcpy(wmemory[0].evnum[i],startn.evnum[i]);
    strcpy(wmemory[0].casen[i],startn.casen[i]);
    strcpy(wmemory[0].objn[i],startn.objn[i]);
    }


     count = 0; /* self increasing counter */
     for (i=0;i<127;i++) { /* infinit loop until match  */

 wmin++;
         gennode(count); /* call gp_prop, decode_ev  and gpnodecopy */
 count++;

 if (unimatch()) break; /* if targetn matches wmemory[wmin]  */
 if (allzero(wmemory[wmin].vecrep))  {
     wmout++; /* expand next layer */
     count = 0;
     }
      }

     if (wmin == 126) {
```

```
printf("\n can not match targetn\n");
        continue; /* continue to next block  */
        }

      makegpchain(); /* gpnodecopy from wmemory[wmin] to wmemory[0], result in
      gpchain, binding backpropagation from targetn     */

      /* print gpchain */
      gpout = 0;
      while(gpchain[gpout].noden[0] != '\0') {
printgpnode(gpchain[gpout]);
gpout++;
}

      /* prepare next startn */
      if (bank4_ps < 0.4) {
  strcpy(startn.noden,targetn.noden);
  for(i=0;i<rsize;i++) startn.vecrep[i] = targetn.vecrep[i];
  startn.parent = targetn.parent;
  for (i=0;i<10;i++) {
    strcpy(startn.evnum[i],targetn.evnum[i]);
    strcpy(startn.casen[i],targetn.casen[i]);
    strcpy(startn.objn[i],targetn.objn[i]);
    }

    } /* if */
     else getnextstartn(); /* search gpchain, find first *unsuccesful* plan */

     }  /* else 1 */

     /* do act-generation */
     printf("\n action-generating start\n");

     /* go to last gp */
     gpout = 0;
     while(gpchain[gpout].noden[0] != '\0') gpout++;

     /* act gen in reverse order */
     gpout--;
     while(gpout >=  0) {

 for(i=0;i<csize;i++) bank1_ag[i] = 0;
        strcpy(bankn2_ag,gpchain[gpout].noden);
 for(i=0;i<rsize;i++) bank2_ag[i] = gpchain[gpout].vecrep[i];
 prop_ag();
 for(i=0;i<rsize;i++) bank3_ag[i] = out_ag[i];
 decode_ev(bank3_ag);
 gpout--;

 }

    }  /* WHILE 1 */

} /* end of main */

/**************** simulation setup and data clear etc ************/
/* clear search space */
clear_gpnode(temp)
struct gpnode temp;
{
  int i;

  temp.noden[0] != '\0';

  for (i=0;i<10;i++) {
      temp.evnum[i][0] = '\0';
      temp.casen[i][0] = '\0';
      temp.objn[i][0] = '\0';
      }
}

/* read symbols into memory */
read_to_store()
{
   int i;

   /* load dictionary; bypass gd-network; use symbolic table */

   printf("\n loading global-dict, conbol-case\n");

   /* read to objstore */
   sfp = fopen("p-global-dict","r"); /* open conbol-object  */
   symp_obj = objstorel;

   while(fscanf(sfp,"%s",symp_obj->name) != EOF) {

      /* read rep */
     for(i=0;i<rsize;i++)
      fscanf(sfp,"%f",&(symp_obj->crep[i]));

     symp_obj++;
   }
   fclose(sfp);


   /* read to case store  */
   lfp = fopen("conbol-case","r"); /* open conbol-case */
   casep = casestorel;

   while(fscanf(lfp,"%s",casep->name) != EOF) {
     if (casep->name[0] != ';') {
     for(i=0;i<lsize;i++)
      fscanf(lfp,"%f",&(casep->rep[i]));
     casep++;
     }
   }
   fclose(lfp);

  /* test probe  */
  printf("\nread_to_store probe");
  printf("\nobjstorel");
  printf("\n%s  ",objstorel[3].name);
  for(i=0;i<rsize;i++)
   printf("%.1f ",objstorel[3].crep[i]);

  printf("\ncasestorel\n");
  printf("%s  ",casestorel[3].name);
  for(i=0;i<lsize;i++)
   printf("%.1f ",casestorel[3].rep[i]);
  /*   test end */

}

/* load several weight files */
load_weight()
{
   int i,j;

  /* load weight for event encoder/decoder net  */

   printf("\n loading weight-te\n");
   wfp = fopen("weight-te","r");

   for (i=0;i<iosize;i++)
    for (j=0;j<rsize;j++) fscanf(wfp,"%f",&wih_ev[i][j]);
    for(i=0;i<rsize;i++)
     for(j=0;j<iosize;j++) fscanf(wfp,"%f",&who_ev[i][j]);

   /* bias load */
    for (i=0;i<rsize;i++) fscanf(wfp,"%f ",&hbias_ev[i]);
    for (i=0;i<iosize;i++) fscanf(wfp,"%f ",&obias_ev[i]);

      fclose(wfp);


   /* load weight for plan-selector net  */

    printf("\n loading weight-ps\n");
    wfp = fopen("weight-ps","r");

    for (i=0;i<pssizei;i++)
     for (j=0;j<pssizeh;j++) fscanf(wfp,"%f",&wih_ps[i][j]);

    for(i=0;i<pssizeh;i++)
     for(j=0;j<pssizeo;j++) fscanf(wfp,"%f",&who_ps[i][j]);

   /* bias load */
    for (i=0;i<pssizeh;i++) fscanf(wfp,"%f ",&hbias_ps[i]);
    for (i=0;i<pssizeo;i++) fscanf(wfp,"%f ",&obias_ps[i]);

      fclose(wfp);


   /* load weight for gp-associator  */

    printf("\n loading weight-gp\n");
    wfp = fopen("weight-gp","r");

    for (i=0;i<gpsizei;i++)
     for (j=0;j<gpsizeh;j++) fscanf(wfp,"%f",&wih_gp[i][j]);

    for(i=0;i<gpsizeh;i++)
     for(j=0;j<gpsizeo;j++) fscanf(wfp,"%f",&who_gp[i][j]);

   /* bias load */
    for (i=0;i<gpsizeh;i++) fscanf(wfp,"%f ",&hbias_gp[i]);
    for (i=0;i<gpsizeo;i++) fscanf(wfp,"%f ",&obias_gp[i]);

      fclose(wfp);

   /* load weight for action-generator net  */

    printf("\n loading weight-ag\n");
    wfp = fopen("weight-ag","r");

    for (i=0;i<agsizei;i++)
     for (j=0;j<agsizeh;j++) fscanf(wfp,"%f",&wih_ag[i][j]);
    for(i=0;i<agsizeh;i++)
     for(j=0;j<agsizeo;j++) fscanf(wfp,"%f",&who_ag[i][j]);

   /* bias load */
    for (i=0;i<agsizeh;i++) fscanf(wfp,"%f ",&hbias_ag[i]);
    for (i=0;i<agsizeo;i++) fscanf(wfp,"%f ",&obias_ag[i]);

      fclose(wfp);

} /* load_weight */

/* print loaded weight for verification */
print_weight()
{

   int i,j;

  /* print  weight for event encoder/decoder net  */
```

```
    printf("\n wih_ev\n");
     for (j=0;j<rsize;j++) printf("%f ",wih_ev[0][j]);
     printf("\n");

    printf("\n who_ev\n");
     for(j=0;j<iosize;j++) printf("%f ",who_ev[0][j]);
     printf("\n");

    printf("\n hbias_ev\n");
    for (i=0;i<rsize;i++) printf("%f ",hbias_ev[i]);

    printf("\n obias_ev\n");
    for (i=0;i<iosize;i++) printf("%f ",obias_ev[i]);

   /* print weight for plan-selector net  */

    printf("\n wih_ps\n");
     for (j=0;j<pssizeh;j++) printf("%f",wih_ps[0][j]);
     printf("\n");

    printf("\n who_ps\n");
     for(j=0;j<pssizeo;j++) printf("%f",who_ps[0][j]);
     printf("\n");

    printf("\n hbias_ps\n");
    for (i=0;i<pssizeh;i++) printf("%f ",hbias_ps[i]);

    printf("\n obias_ps\n");
    for (i=0;i<pssizeo;i++) printf("%f ",obias_ps[i]);

   /* print weight for gp-associator  */

    printf("\n wih_gp\n");
     for (j=0;j<gpsizeh;j++) printf("%f",wih_gp[0][j]);
     printf("\n");

    printf("\n who_gp\n");
     for(j=0;j<gpsizeo;j++) printf("%f",who_gp[0][j]);
     printf("\n");

    printf("\n hbias_gp\n");
    for (i=0;i<gpsizeh;i++) printf("%f ",hbias_gp[i]);
    printf("\n obias_gp\n");
    for (i=0;i<gpsizeo;i++) printf("%f ",obias_gp[i]);

   /* print weight for action-generator net  */

    printf("\n wih_ag\n");
     for (j=0;j<agsizeh;j++) printf("%f",wih_ag[0][j]);
    printf("\n");

    printf("\n who_ag\n");
     for(j=0;j<agsizeo;j++) printf("%f",who_ag[0][j]);
     printf("\n");

    printf("\n hbias_ag\n");
    for (i=0;i<agsizeh;i++) printf("%f ",hbias_ag[i]);
    printf("\n obias_ag\n");
    for (i=0;i<agsizeo;i++) printf("%f ",obias_ag[i]);
      printf("\n");

} /* print_weight */

/* read input story from inputfile; batch mode  */
read_story()
{
 /* read single story into triple forms; bypass st-parser */
  int i;

  scanf("%s",repn); /* read script flag */
  if (strcmp(repn,"script") == 0) script = 1;
  else script = 0;

  inp = inputstory1;

  while(scanf("%s",repn) != EOF) {

  if (repn[0] == ';')  continue; /* comment */
  else if (strcmp(repn,"block:") == 0)  {   /* if block designator */
     strcpy(inp->evnum,repn);
     inp++;
     continue;
     }
  else {  /* triple */
    scanf("%s",linkn);
    scanf("%s",noden);

  /* load event name */
  strcpy(inp->evnum,repn);

  /* load object pointer  */
  i=0;
  while((strcmp(noden,objstore1[i].name) != 0) && (objstore1[i].name[0] != '\0')) i++;
  if (objstore1[i].name[0] != '\0') inp->object = i;
  else {
    inp->object = -1; /* event, not object */
    strcpy(inp->object2,noden);
  }

  /* load case rep */
  i=0;
  while((strcmp(linkn,casestore1[i].name) != 0) && (casestore1[i].name[0] != '\0')) i++;
  if (casestore1[i].name[0] == '\0') printf("\n error: no %s",linkn);
  else inp->case_role = i;

  inp++; /* next store */
  } /* else */

  } /* while */

   /* verify story */
   printf("\n input story\n");
   inp = inputstory1;
     while(inp->evnum[0] != '\0') {
     printf("%s %d %d %s\n",inp->evnum,inp->case_role,inp->object,inp->object2);
     inp++;
     }

}

/*************** propagations for each module ******************/
/* triple-encoder propagation */
prop_ev()
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<rsize;i++) in_ev[pos++] = rep[i];
  for(i=0;i<lsize;i++) in_ev[pos++] = link[i];
  for(i=0;i<rsize;i++) in_ev[pos++] = node[i];

 /* prop procedure */

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<rsize;i++)  {
       net = 0.0;
       for(j=0;j<iosize;j++)
 net = net + wih_ev[j][i]*in_ev[j];

 net = net + hbias_ev[i];
 hidden_ev[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<iosize;i++) {
       net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_ev[j][i]*hidden_ev[j];

 net = net + obias_ev[i];
 out_ev[i] = sigmoid(net);
 }

  /* print net  */
   printf("\n%s %s %s\n",repn,linkn,noden);
   printf("\n input\n");
   for(i=0;i<iosize;i++) printf("%.1f ",in_ev[i]);   printf("\n");
   printf("hidden\n");
   for (i=0;i<rsize;i++) printf("%.1f ",hidden_ev[i]); printf("\n");
   printf("output\n");
   for(i=0;i<iosize;i++) printf("%.1f ",out_ev[i]); printf("\n");

} /* prop_ev */

float sigmoid(x)
float x;
{
 double exp();
 return( 1.0 / (1.0 + exp(-x)));
}

/* plan-selector propagation */
prop_ps()
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<csize;i++) in_ps[pos++] = bank1_ps[i];
  for(i=0;i<rsize;i++) in_ps[pos++] = bank2_ps[i];

 /* prop procedure */
  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<pssizeh;i++)  {
       net = 0.0;
       for(j=0;j<pssizei;j++)
 net = net + wih_ps[j][i]*in_ps[j];

 net = net + hbias_ps[i];
 hidden_ps[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<pssizeo;i++) {
       net = 0.0;
       for(j=0;j<pssizeh;j++)
 net = net + who_ps[j][i]*hidden_ps[j];

 net = net + obias_ps[i];
 out_ps[i] = sigmoid(net);
 }

  /* print net  */
   printf("\n%s\n",bankn2_ps);
   printf("\n input\n");
   for(i=0;i<pssizei;i++) printf("%.1f ",in_ps[i]);   printf("\n");
   printf("hidden\n");
   for (i=0;i<pssizeh;i++) printf("%.1f ",hidden_ps[i]); printf("\n");
   printf("output\n");
   for(i=0;i<pssizeo;i++) printf("%.1f ",out_ps[i]); printf("\n");

} /* prop_ps */

/* decode into triple; at the moment, it is story31 dependent code */
/* output is in decodeout  */
decode_ev(vector)
float vector[rsize];
{
  /* if goal/plan/state/act then stop decoding
     if state(wanted) then embedded decoding */

  float embedrep[rsize]; /* holds embedded representation */
  float caserep[rsize],objrep[rsize];
  int pos,i,depth;
  int embed; /* 0: unembeded event, 1: embedded event */
  int trpc,trpc2;

  /* clear decodeout */
  clear_gpnode(decodeout);

  strcpy(decodeout.noden,"evgp");
  for(i=0;i<rsize;i++) decodeout.vecrep[i] = vector[i];

/* start decoding */
  trpc = 0;

 for(i=0;i<rsize;i++) hidden_ev[i] = vector[i];

 printf("\n decoding ");
 for(i=0;i<rsize;i++) printf("%.1f ",hidden_ev[i]);
 printf("\n");

  depth = 0;
  while (depth < 5)  {   /* infinite loop */

     decode_prop();

     /* copy and recirculate */
     pos = rsize;
     for(i=0;i<lsize;i++) caserep[i] = out_ev[pos++];
     for(i=0;i<rsize;i++) objrep[i] = out_ev[pos++];

     for(i=0;i<rsize;i++) hidden_ev[i] = out_ev[i];

     /* find symbol and store */
     strcpy(decodeout.evnum[trpc],"evgp");
     find_obj(objrep,trpc);
     find_case(caserep,trpc); /* find symbol and store into decodeout */

    if (trpc == 0) for(i=0;i<rsize;i++) embedrep[i] = objrep[i];

    if (strcmp(decodeout.casen[trpc],"goal") == 0 ||
        strcmp(decodeout.casen[trpc],"plan") == 0 ||
        strcmp(decodeout.casen[trpc],"state") == 0 ||
        strcmp(decodeout.casen[trpc],"act") == 0) break;

    trpc++;
    depth++;
  }  /* while */

      /* if embed is 1 then gpflag is 0  */
  if (strcmp(decodeout.casen[trpc],"state") == 0 &&
      strcmp(decodeout.objn[trpc],"wanted") == 0) embed = 1;

  if (embed == 1)  { /* do embedded event decoding */
    trpc++;
    for(i=0;i<rsize;i++) hidden_ev[i] = embedrep[i];
    depth = 0;
    while (depth < 5)  {   /* infinite loop */
     decode_prop();
     /* copy and recirculate */
     pos = rsize;
     for(i=0;i<lsize;i++) caserep[i] = out_ev[pos++];
     for(i=0;i<rsize;i++) objrep[i] = out_ev[pos++];
     for(i=0;i<rsize;i++) hidden_ev[i] = out_ev[i];

     /* find symbol and store */
     strcpy(decodeout.evnum[trpc],"evgp");
     find_obj(objrep,trpc);
     find_case(caserep,trpc); /* find symbol and store into decodeout */

    if (strcmp(decodeout.casen[trpc],"state") == 0 ||
        strcmp(decodeout.casen[trpc],"act") == 0) break;

    trpc++;
    depth++;
    } /* while */

    strcpy(decodeout.objn[0],"evgp");
    }  /* if embedded */

 /* verify decoding */
 printf("\n decoded output \n");
 printgpnode(decodeout);

}

/* event decoder propagation; only from hidden layer to the output layer  */
decode_prop()
{

  float net;
  float sigmoid();
  int i,j,pos,flag;


   /* from hidden to output */
     for(i=0;i<iosize;i++) {
       net = 0.0;
       for(j=0;j<rsize;j++)
 net = net + who_ev[j][i]*hidden_ev[j];

 net = net + obias_ev[i];
 out_ev[i] = sigmoid(net);
 }
```

```
   /* print net  */
    printf("hidden\n");
    for (i=0;i<rsize;i++) printf("%.1f ",hidden_ev[i]); printf("\n");
    printf("output\n");
    for(i=0;i<iosize;i++) printf("%.1f ",out_ev[i]); printf("\n");

 } /* decode_prop */

 find_obj(objrep,trpc)
 float objrep[rsize];
 int trpc;
 {

   int sympos,matchpos; /* matched symbol position */
   float dista;
   float prevdist = 12.0; /* max distance */
   int i;

  /* binary thresholding */
  for(i=0;i<rsize;i++)
    if (objrep[i] <= 0.5) objrep[i] = 0;
    else objrep[i] = 1;

 /*  ID+DSR search */
   sympos = 0;
   while(objstorel[sympos].name[0] != '\0')  {

     /* calculate euclidean distance */
      dista = 0.0;
      for (i=0;i<rsize;i++)
      dista = dista
    + (objstorel[sympos].crep[i] - objrep[i])
    * (objstorel[sympos].crep[i] - objrep[i]);

   /*  printf("\n distnace for %s  --> %f\n",objstorel[sympos].name,dista); */

     /* compare */
     if (dista < prevdist) {
      matchpos = sympos;
      prevdist = dista;   }

      sympos++;
     }

/* store */
strcpy(decodeout.objn[trpc],objstorel[matchpos].name);
printf("\n found symbol is %s in %f\n",decodeout.objn[trpc],prevdist);

}

/* find closed symbols for the case-role patterns */
find_case(caserep,trpc)
float caserep[rsize];
int trpc;
{

  int sympos,matchpos; /* matched symbol position */
  float dista;
  float prevdist = 12.0; /* max distance */
  int i;

/*  ID+DSR search */
  sympos = 0;
  while(casestorel[sympos].name[0] != '\0')  {

    /* calculate euclidean distance */
     dista = 0.0;
     for (i=0;i<rsize;i++)
```

```
      dista = dista
    + (casestorel[sympos].rep[i] - caserep[i])
    * (casestorel[sympos].rep[i] - caserep[i]);

   /*  printf("\n distnace for %s  --> %f\n",casestorel[sympos].name,dista); */

     /* compare */
     if (dista < prevdist) {
      matchpos = sympos;
      prevdist = dista;   }

      sympos++;
     }

/* store; data-dependent interpretation of goal/plan case  */
if (decodeout.objn[trpc][0] == 's' &&
    decodeout.objn[trpc][1] == '-' &&
    strcmp(casestorel[matchpos].name,"state") == 0)
    strcpy(decodeout.casen[trpc],"goal");
else if (decodeout.objn[trpc][0] == 'd' &&
    decodeout.objn[trpc][1] == '-' &&
    strcmp(casestorel[matchpos].name,"state") == 0)
    strcpy(decodeout.casen[trpc],"goal");
else if (decodeout.objn[trpc][0] == 'p' &&
     decodeout.objn[trpc][1] == 'b' &&
    decodeout.objn[trpc][2] == '-' &&
    strcmp(casestorel[matchpos].name,"act") == 0)
    strcpy(decodeout.casen[trpc],"plan");
else strcpy(decodeout.casen[trpc],casestorel[matchpos].name);

printf("\n found symbol is %s in %f\n",decodeout.casen[trpc],prevdist);

}

/* print node structure in the gp-tree */
printgpnode(node)
struct gpnode node;
{
 int i;

  printf("\n %s\n",node.noden);
  for(i=0;i<rsize;i++)
  printf("%.1f ",node.vecrep[i]);
  printf("\n");
  i = 0;
  while(node.evnum[i][0] != '\0') {
    printf("%s %s %s\n",node.evnum[i],node.casen[i],node.objn[i]);
    i++;
    }
}

/*********** gp-tree expanding, matching, searching... ***********/
/* generate gpnode to expand; get node from wmout, gp_prop, decode_ev,
and store into wmin */
gennode(count)
int count;
{
  int i;

  /* setup selfincreasing count */
  if (count == 0) {
     bank1_gp[0] = 0; bank1_gp[1] = 0; }
  else if (count == 1) {
     bank1_gp[0] = 0; bank1_gp[1] = 1; }
  else if (count == 2) {
     bank1_gp[0] = 1; bank1_gp[1] = 0; }
  else if (count == 3) {
     bank1_gp[0] = 1; bank1_gp[1] = 1; }
    else printf("\n wrong self increasing counter in gp-assoc.c \n");

    /* cpy bank2_gp  */
    for (i=0;i<rsize;i++) bank2_gp[i] = wmemory[wmout].vecrep[i];
    strcpy(bankn2_gp,wmemory[wmout].noden);

    prop_gp();

   /* copy output gp */
    for (i=0;i<rsize;i++) bank3_gp[i] = out_gp[i];

    decode_ev(bank3_gp);

 /* copy decodout to wmemory[wmin] */
    strcpy(wmemory[wmin].noden,decodeout.noden);
    for(i=0;i<rsize;i++) wmemory[wmin].vecrep[i] = decodeout.vecrep[i];
    wmemory[wmin].parent = decodeout.parent;
    for (i=0;i<10;i++) {
      strcpy(wmemory[wmin].evnum[i],decodeout.evnum[i]);
      strcpy(wmemory[wmin].casen[i],decodeout.casen[i]);
      strcpy(wmemory[wmin].objn[i],decodeout.objn[i]);
      }


   /* set up parent pointer */
   wmemory[wmin].parent = wmout;

 }

 /* unification matching routine; targetn vs wmemory[wmin] node;
  use objstorel to match var and inst; build-up binding list for backward
  binding propagation (non interesting symbol manipulation);
  forward binding prop already done by network (prop_gp)  */
 unimatch()
 {
    int match = 1;
    int trpc;

     /* targetn has instantiated forms, wmemory[wmin] has uninstantiated
     forms */
    trpc = 0;
    while(wmemory[wmin].evnum[trpc][0] != '\0') {
      if (!sameco(trpc)) match = 0;
      trpc++;
      }

     return(match);
 }

 /*  compare caserole and obj pointed by trpc index;
     when compare obj, use objstorel to do unification matching */
 sameco(trpc)
 int trpc;
 {
    if (strcmp(wmemory[wmin].casen[trpc],targetn.casen[trpc]) != 0) return(0);
    else {

        if (wmemory[wmin].objn[trpc][0] != '?')  /* not variable */
    if (strcmp(wmemory[wmin].objn[trpc],targetn.objn[trpc]) != 0)
      return(0);
    else return(1);
        else
  if (varinst(wmemory[wmin].objn[trpc],targetn.objn[trpc])) return(1);
  else return(0);
       }
 }

 /*  see the instance matches to the variable
```

```
code assumes that global-dict are var1,inst11,inst12...var2 form */
varinst(var,inst)
char var[nsize],inst[nsize];
{

  if (strcmp(var,inst) == 0) return(1);

  symp_obj = objstorel;
  while(strcmp(symp_obj->name,var) != 0) symp_obj++;
  symp_obj++;
  while(symp_obj->name[0] != '?' && symp_obj->name[0] != '\0') {
    if (strcmp(symp_obj->name,inst) == 0) return(1);
    symp_obj++;
 }

 return(0);

}

/* check nulll pattern */
allzero(vector)
float vector[rsize];
{
   int i;
   float sum;

   sum = 0;
   for(i=0;i<rsize;i++)
     sum = sum + vector[i];

   if (sum < 1.0)  return(1);
   else return(0);
}

/* make output goal/plan chain */
makegpchain()
{
  int i,j,k;

  i = 0;
  j = wmin;
  while(j != 0) {

/* cpy wmemory[j] to gpchain[i] */
  strcpy(gpchain[i].noden,wmemory[j].noden);
  for(k=0;k<rsize;k++) gpchain[i].vecrep[k] = wmemory[j].vecrep[k];
  gpchain[i].parent = wmemory[j].parent;
  for (k=0;k<10;k++) {
    strcpy(gpchain[i].evnum[k],wmemory[j].evnum[k]);
    strcpy(gpchain[i].casen[k],wmemory[j].casen[k]);
    strcpy(gpchain[i].objn[k],wmemory[j].objn[k]);
    }

    j = wmemory[j].parent;
    i++;
    }  /* while */

/* cpy wmeory[0] to gpchian */
  strcpy(gpchain[i].noden,wmemory[j].noden);
  for(k=0;k<rsize;k++) gpchain[i].vecrep[k] = wmemory[j].vecrep[k];
  gpchain[i].parent = wmemory[j].parent;
  for (k=0;k<10;k++) {
    strcpy(gpchain[i].evnum[k],wmemory[j].evnum[k]);
    strcpy(gpchain[i].casen[k],wmemory[j].casen[k]);
    strcpy(gpchain[i].objn[k],wmemory[j].objn[k]);
    }

}
```

```
 /* simplified routine; get next unsuccessful plan */
 getnextstartn()
 {
    int i,j;

    j = 1; /* skip target */
    while(!isplan(gpchain[j])) j++;

   strcpy(startn.noden,gpchain[j].noden);
   for(i=0;i<rsize;i++) startn.vecrep[i] = gpchain[j].vecrep[i];
   startn.parent = gpchain[j].parent;
   for (i=0;i<10;i++) {
     strcpy(startn.evnum[i],gpchain[j].evnum[i]);
     strcpy(startn.casen[i],gpchain[j].casen[i]);
     strcpy(startn.objn[i],gpchain[j].objn[i]);
     }


}

/* is node a plan? */
isplan(node)
struct gpnode node;
{
  int i;
  i = 0;
  while(node.evnum[i][0] != '\0') {
    if (strcmp(node.casen[i],"plan") == 0) return(1);
    i++;
   }

  return(0);
}

/********* propagations for each module; action-generator, gp-assoc *******/
/* action-generator propagation */
prop_ag()
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<csize;i++) in_ag[pos++] = bank1_ag[i];
  for(i=0;i<rsize;i++) in_ag[pos++] = bank2_ag[i];

 /* prop procedure */
  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<agsizeh;i++)  {
       net = 0.0;
       for(j=0;j<agsizei;j++)
net = net + wih_ag[j][i]*in_ag[j];

net = net + hbias_ag[i];
hidden_ag[i] = sigmoid(net);
}

  /* from hidden to output */

    for(i=0;i<agsizeo;i++) {
      net = 0.0;
      for(j=0;j<agsizeh;j++)
net = net + who_ag[j][i]*hidden_ag[j];
```

```
 net = net + obias_ag[i];
 out_ag[i] = sigmoid(net);
 }

  /* print net  */
   printf("\n%s\n",bankn2_ag);
   printf("\n input\n");
   for(i=0;i<agsizei;i++) printf("%.1f ",in_ag[i]);   printf("\n");
   printf("hidden\n");
   for (i=0;i<agsizeh;i++) printf("%.1f ",hidden_ag[i]); printf("\n");
   printf("output\n");
   for(i=0;i<agsizeo;i++) printf("%.1f ",out_ag[i]); printf("\n");

} /* prop_ag */

/* gp-tree expansion; gp-associator propagation */
prop_gp()
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */

  pos = 0;
  for(i=0;i<scsize;i++) in_gp[pos++] = bank1_gp[i];
  for(i=0;i<rsize;i++) in_gp[pos++] = bank2_gp[i];

 /* prop procedure */
  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<gpsizeh;i++)  {
       net = 0.0;
       for(j=0;j<gpsizei;j++)
 net = net + wih_gp[j][i]*in_gp[j];

 net = net + hbias_gp[i];
 hidden_gp[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<gpsizeo;i++) {
       net = 0.0;
       for(j=0;j<gpsizeh;j++)
 net = net + who_gp[j][i]*hidden_gp[j];

 net = net + obias_gp[i];
 out_gp[i] = sigmoid(net);
 }

  /* print net  */
   printf("\n%s\n",bankn2_gp);
   printf("\n input\n");
   for(i=0;i<gpsizei;i++) printf("%.1f ",in_gp[i]);   printf("\n");
   printf("hidden\n");
   for (i=0;i<gpsizeh;i++) printf("%.1f ",hidden_gp[i]); printf("\n");
   printf("output\n");
   for(i=0;i<gpsizeo;i++) printf("%.1f ",out_gp[i]); printf("\n");

} /* prop_gp */
```

APPENDIX G

DYNASTY analysis code

G.1 Performance statistics

This section lists a group of codes which were used to calculate the performance statistics tables listed in section 5.2. Each module has separate statistics gathering programs. Most of the duplicate codes are omitted.

```
/*********** for global-dictionary *********************************/
/* performance statistics for global-dict network module; ascii-to-dsr
   training data: global-dict into memory */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define isize_a2d nsize /* input layer size */
#define hsize_a2d 15 /* hidden layer size */
#define osize_a2d rsize /* output layer size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */

#define sfile "global-dict" /* input training data file */
#define wfile "weight-a2d"  /* loaded weight file */

/* global dictionary -- training data  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;


/* main network */
/* current network holder */
char bankn1_a2d[nsize];
float bank1_a2d[nsize],bank2_a2d[rsize]; /* input rep holder  */

/* gd main network */
float in_a2d[isize_a2d],out_a2d[osize_a2d],hidden_a2d[hsize_a2d],
teach_a2d[osize_a2d];
float wih_a2d[isize_a2d][hsize_a2d],who_a2d[hsize_a2d][osize_a2d];
float hbias_a2d[hsize_a2d],obias_a2d[osize_a2d];

/* global file pointer  */
FILE *fopen(),*sfp,*wfp;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  read_td();  /* read from initial gd as training data; not listed  */

  load_weight(); /* load previous weight for continuous training; not listed  */

/* initialize para  */
cosym = 0;
tosym = 0;
nounit = 0;
toerror = 0;

/* for every training pair in the global-dict */
symp_obj = objstorel;
while (symp_obj->name[0] != '\0') {  /* for all the words */

  strcpy(bankn1_a2d,symp_obj->name);

/* normalize ascii for a(97) -- z(122) into 0 -- 1
   if less than 0; then ? or -  or space       */

  for (i=0;i<nsize;i++)
     bank1_a2d[i] = (float) (symp_obj->name[i] - 96) / 26.0;

  for(i=0;i<rsize;i++)
     bank2_a2d[i] = symp_obj->crep[i];

prop(total); /* verify -- weight freezed */

   symp_obj++;
   } /* while */

   /* performance statistics */
   correct = (float) cosym / (float) tosym;
   eavg = toerror / (float) nounit;
   printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
   printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* main */

/******* propagate while caculating performance statistics ******/
prop(total)
int total;
{
  float net,pesig;
  float sigmoid();
  int i,j,pos,cycle,flag;

/* load data */
  /* input layer */
  for(i=0;i<nsize;i++) in_a2d[i] = bank1_a2d[i];

 /* teach layer : hetero-associative network  */
  for(i=0;i<rsize;i++) teach_a2d[i] = bank2_a2d[i];

  /* forward prop */
   /* from input to hidden forward */

     for(i=0;i<hsize_a2d;i++)  {
       net = 0.0;
       for(j=0;j<isize_a2d;j++)
 net = net + wih_a2d[j][i]*in_a2d[j];

 net = net + hbias_a2d[i];
 hidden_a2d[i] = sigmoid(net);
 }

   /* from hidden to output */

     for(i=0;i<osize_a2d;i++) {
        net = 0.0;
        for(j=0;j<hsize_a2d;j++)
  net = net + who_a2d[j][i]*hidden_a2d[j];

  net = net + obias_a2d[i];
  out_a2d[i] = sigmoid(net);
  }

   /* statistics gathering; epsilon is 0.15 */
   flag = 0;
   for (i=0;i<osize_a2d;i++) {
    if (fabs(teach_a2d[i] - out_a2d[i]) > 0.15) flag = 1;
    toerror = toerror + (float) fabs(teach_a2d[i] - out_a2d[i]);
    nounit++;
    }

   if (flag == 0) cosym++;
   tosym++;
    printf("\n %d %d %f %d\n",cosym,tosym,toerror,nounit);

   /* print net  */
    printf("\n %d %s \n",total,bankn1_a2d);
    printf("input layer\n");
    for (i=0;i<isize_a2d;i++) printf("%.1f ",in_a2d[i]);
    printf("\nhidden layer\n");
    for (i=0;i<hsize_a2d;i++) printf("%.1f ",hidden_a2d[i]);
    printf("\noutput-teach pair\n");
    for(i=0;i<osize_a2d;i++) printf("%.1f ",out_a2d[i]); printf("\n");
    for(i=0;i<osize_a2d;i++) printf("%.1f ",teach_a2d[i]);   printf("\n");
}

/************** for triple-encoder ********************/
/* get performance statistics for triple-encoder network module
   data: tdata-te into memory
   input symbol dictionary: global-dict, conbol-case
   dynstat3r                                     */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define iosize lsize+2*rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 100 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_te 400 /* number of triples in tdata-te  */

#define dfile "tdata-te"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
```

```
struct tdata_te {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_te_l[nums_td_te],*tdp1;

/* previous event rep holder  */
char preev[nsize];
float preevr[rsize];

/* ARPDP network accessory */
char repn[nsize],linkn[nsize],noden[nsize]; /* input name holder */
float rep[rsize],link[lsize],node[rsize]; /* input rep holder  */

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int max,total,mtotal;
  int i;
  char tempev[nsize];
  float ran; /* random number output */

  read_to_store();  /* read from initial gd and case to internal data
        structure; not listed    */
  load_weight(); /* load previous weight for continuous training; not listed  */

 /* print initial weight to the log */
  print_weight(); /* not listed */

 /* read traning data into memory; object hold -1 if embedded */
 read_td_te(); /* not listed */

/* initialize para  */
cosym = 0;
tosym = 0;
nounit = 0;
toerror = 0;

/* triple encoding through all data; one epoch */
strcpy(tempev,"xxx");
tdp1 = tdata_te_l;
while(tdp1->evnum[0] != '\0') { /* while 2 */

strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
 rep[i] = 0.5; /* init ev rep */
 }

        for (i=0;i<lsize;i++)
 link[i] = casestorel[tdp1->case_role].rep[i];

 if (tdp1->object >= 0)  /* no recursive event */
           for (i=0;i<rsize;i++)
      node[i] = objstorel[tdp1->object].crep[i];
          else if (strcmp(tdp1->object2,preev) == 0)
   for (i=0;i<rsize;i++)
      node[i] = preevr[i];
          else printf("\error no %s\n",noden);

        prop_ev(total);

        for (i=0;i<rsize;i++)
           rep[i] = hidden_ev[i];

       strcpy(tempev,repn); /* for event group checking */
       tdp1++;
} /* while */

    /* performance statistics */
    correct = (float) cosym / (float) tosym;
    eavg = toerror / (float) nounit;
    printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
    printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* main */

/************* for plan-selector *****************/
/* performance statistics for plan-selector network module
   data: tdata-ps into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te  */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 30 /* context size */
#define isize csize+rsize
#define osize rsize+1
#define iosize rsize+lsize+rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ps 1000 /* number of triples in tdata-te  */
#define nums_td_in 400 /* number of internal data */

#define dfile "tdata-ps"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"
#define wfile2 "weight-ps"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_ps {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_ps_l[nums_td_ps],*tdp1;

/* internal training data */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize];
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn1[nsize],bankn2[nsize],bankn3[nsize],bankn4[nsize];
float bank1[csize],bank2[rsize],bank3[rsize],bank4; /* input rep holder  */
/* plan-selector main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;

  load_weight_te(); /* triple encoder weight file; not listed  */

  read_to_store();  /* read from initial gd and case to internal data
       structure; not listed    */
  load_weight(); /* load previous weight for continuous training; not listed  */

 /* read training data into memory; object hold -1 if embedded */
 read_td_ps(); /* not listed */

triple_encode(total); /* internal data build up; not listed  */

/* initialize para  */
cosym = 0;
tosym = 0;
nounit = 0;
toerror = 0;

/* for every training pair in the tdata_internal */
tdpi = tdata_internal_l;
```

```
 while (tdpi->key_word[0] != '\0')  {  /* while  3 */

 tdpi2 = tdpi;
 /* then part assign for bank3 and bank4  */
 while(strcmp(tdpi2->key_word,"then") != 0) tdpi2++;
     if (strcmp(tdpi2->key_word,"then") == 0) {
  strcpy(bankn3,tdpi2->evnum);
  for (i=0;i<rsize;i++)  bank3[i] = tdpi2->trep[i];
  }
     else printf("\n no then part");
    tdpi2++;
    if (strcmp(tdpi2->key_word,"and") == 0) {
    strcpy(bankn4,tdpi2->evnum);
    bank4 = tdpi2->trep[0];
    }
    else printf("\n no then part");

  /* if part for bank1 and bank2 */
  while(strcmp(tdpi->key_word,"then")!= 0) { /* while 4 */
  if (strcmp(tdpi->key_word,"if") == 0 ||
      strcmp(tdpi->key_word,"follows") == 0) {
 strcpy(bankn2,tdpi->evnum);
         for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
 }
  else printf("\n no if part");

  if (strcmp(tdpi->key_word,"if") == 0)
      for (i=0;i<csize;i++) bank1[i] = 0;


 prop(total); /* verify -- weight freezed */

        for (i=0;i<csize;i++) bank1[i] = hidden[i];

       tdpi++;
 } /* while  4 */

   while(strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') tdpi++;
   } /* while  3 */

    /* performance statistics */
    correct = (float) cosym / (float) tosym;
    eavg = toerror / (float) nounit;
    printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
    printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* main */

/************ for gp-associator ********************/
/* performance statistics for gp-associator network module
   data: tdata-gp into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te  */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define hsize 10 /* hidden size */
#define scsize 2  /* counter size */
#define isize scsize+rsize
#define osize rsize
#define iosize rsize+lsize+rsize
#define idsize 2 /* id bit size */

/* max data size */
```

```
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_gp 1000 /* number of triples in tdata-gp  */
#define nums_td_in 30 /* number of internal data for ONE gp */

#define dfile "tdata-gp"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"
#define wfile2 "weight-gp"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_gp {
  char evnum[nsize];
  int case_role; /* hold counter value  0 to 3 too */
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_gp_l[nums_td_gp],*tdp1,*tdp2;

/* internal training data */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize]; /* first two array elts holds counter value */
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn2[nsize],bankn3[nsize];
float bank1[scsize],bank2[rsize],bank3[rsize]; /* input rep holder  */
/* gp-assoc main network */
float in[isize],out[osize],hidden[hsize],teach[osize];
float wih[isize][hsize],who[hsize][osize];
float hbias[hsize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int total,mtotal;
  int i,j;
  float ran;
```

```
 /* previous event rep holder for triple-encoder  */
 char preev[nsize];
 float preevr[rsize];
 char tempev[nsize];

   load_weight_te(); /* triple encoder weight file; not listed  */

   read_to_store();  /* read from initial gd and case to internal data
        structure; not listed    */
   load_weight(); /* load previous weight for continuous training; not listed  */

  /* read training data into memory; object hold -1 if embedded */
  read_td_gp(); /* not listed */

 /* initialize para  */
 cosym = 0;
 tosym = 0;
 nounit = 0;
 toerror = 0;

 /* for all data in tdata-gp  */
 tdp1 = tdata_gp_1;

 while (tdp1->evnum[0] != '\0')  {  /* while 100 at the end of data */
 tdp1++; /* skip coment */

 /* random id assign for this gp association; assign to only var in if part
  clear id in objstore; every var is up front in objstore */
 symp_obj = objstorel;
 while(symp_obj->name[0] == '?') {
   for (i=0;i<idsize;i++) symp_obj->crep[i] = 0;
   symp_obj++;
   }

 tdp2 = tdp1;
 tdp2++;
 tdp2++;
 while(strcmp(tdp2->evnum,"then") != 0) {  /* while 22 */
 if (tdp2->object != -1 && objstorel[tdp2->object].name[0] == '?')
  for (i=0;i<idsize;i++) {
     ran = rand()/32767.0;
     if (ran < 0.51) objstorel[tdp2->object].crep[i] =  0.0;
     else objstorel[tdp2->object].crep[i] = 0.0;
     }

tdp2++;
} /*  while 22 */

/* start triple encode here */
if (total == 0)
  printf("\n triple-encode verify for epoch 0\n");

/* clear tdata_internal */
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0') {
  tdpi->key_word[0] = '\0';
  tdpi++;
  }

/* get triple representation for this id; store into internal
data file; tdata_internal  */

tdpi = tdata_internal_1;
while (tdp1->evnum[0] != ';' && tdp1->evnum[0] != '\0')  { /* while 101 */
     if (strcmp(tdp1->evnum,"if") == 0) {
       strcpy(tdpi->key_word,tdp1->evnum);
       if (tdp1->case_role == 0) { tdpi->trep[0] = 0; tdpi->trep[1] = 0; }
       else if (tdp1->case_role == 1) { tdpi->trep[0] = 0; tdpi->trep[1] = 1; }
       else if (tdp1->case_role == 2) { tdpi->trep[0] = 1; tdpi->trep[1] = 0; }
       else if (tdp1->case_role == 3) { tdpi->trep[0] = 1; tdpi->trep[1] = 1; }
       else printf("\n invalid counter ");
       tdpi++; tdp1++;
       }
     else if (strcmp(tdp1->evnum,"and") == 0 ||
      strcmp(tdp1->evnum,"then") == 0) { /* else if 1 */
         strcpy(tdpi->key_word,tdp1->evnum);
 tdp1++;

        if (strcmp(tdp1->evnum,"nil") == 0) {
   strcpy(tdpi->evnum,tdp1->evnum);
   tdp1++; tdpi++;
   }

        else {   /* else 10 */  /* triple encoding */

 strcpy(tempev,"xxx");
 while (strcmp(tdp1->evnum,"if") != 0 &&
strcmp(tdp1->evnum,"then") != 0 &&
tdp1->evnum[0] != ';' &&
tdp1->evnum[0] != '\0') { /* while 2 */

/* event encoding until key word */

strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];

if (tdp1->object >= 0)  /* no recursive event */
          for (i=0;i<rsize;i++)
     node[i] = objstorel[tdp1->object].crep[i];
        else if (strcmp(tdp1->object2,preev) == 0)
  for (i=0;i<rsize;i++)
     node[i] = preevr[i];
        else printf("\error no %s\n",noden);

       prop_ev(total);

       for (i=0;i<rsize;i++)
          rep[i] = hidden_ev[i];

      strcpy(tempev,repn); /* for event group checking */
      tdp1++;
} /* while 2  */

      strcpy(tdpi->evnum,repn);
      for (i=0;i<rsize;i++)
tdpi->trep[i] = rep[i];
tdpi++;
        }  /* else 10  end triple */
  } /* else if 1 */

    else printf("\n input data error: no if and then format");

  } /* while 101 */

  /* verify encoding internal data */
  if (total == 0) {
  printf("\n tdata_internal");
  tdpi = tdata_internal_1;
  while (tdpi->key_word[0] != '\0') {
    printf("\n%s  %s  ",tdpi->key_word,tdpi->evnum);
    for (j=0;j<rsize;j++)
      printf("%.1f ",tdpi->trep[j]);
      tdpi++;
      }
  }

  /* do gp-association training */

  /* for every training pair in the tdata_internal */
  tdpi = tdata_internal_1;
  while (tdpi->key_word[0] != '\0')  {  /* while  3 */

   /* if part for bank1 and bank2 */
   if (strcmp(tdpi->key_word,"if") == 0)  {
        for(i=0;i<scsize;i++)  bank1[i] = tdpi->trep[i]; /* first two bits */
        tdpi++;
        }
   else printf ("\n no if part ");

   if (strcmp(tdpi->key_word,"and") == 0) {
        strcpy(bankn2,tdpi->evnum);
        for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
        tdpi++;
        }
   else printf("\n no if part");

  /* then part for bank3 */

  if (strcmp(tdpi->key_word,"then") == 0) {
      strcpy(bankn3,tdpi->evnum);
      if (strcmp(tdpi->evnum,"nil") == 0)
  for (i=0;i<rsize;i++) bank3[i] = 0;
      else for (i=0;i<rsize;i++) bank3[i] = tdpi->trep[i];
      tdpi++;
      }
   else printf("\n no then part");

prop(total); /* verify -- weight freezed */

   } /* while  3 */

} /* while 100 for all data */

   /* performance statistics */
   correct = (float) cosym / (float) tosym;
   eavg = toerror / (float) nounit;
   printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
   printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* main */

/************ for action-generator ****************/
/* performance statistics for action-generator network module
   data: tdata-ag into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te */
```

```
#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 30 /* context size */
#define isize csize+rsize
#define osize rsize
#define iosize rsize+lsize+rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ag 1000 /* number of triples in tdata-te  */
#define nums_td_in 400 /* number of internal data */

#define dfile "tdata-ag"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"
#define wfile2 "weight-ag"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_ag {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_ag_l[nums_td_ag],*tdp1;

/* internal training data */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize];
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn1[nsize],bankn2[nsize],bankn3[nsize];
float bank1[csize],bank2[rsize],bank3[rsize]; /* input rep holder  */

/* action-gen main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];
```

```
 /* global file pointer  */
 FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

 /* statistics parameter */
 int cosym,tosym,nounit;
 float correct,toerror,eavg;

 /* driver */
 main()
 {
   int total,mtotal;
   int i,j;
   float ran;

   load_weight_te(); /* triple encoder weight file; not listed  */

   read_to_store();  /* read from initial gd and case to internal data
        structure; not listed    */
   load_weight(); /* load previous weight for continuous training; not listed  */

  /* read training data into memory; object hold -1 if embedded */
  read_td_ag(); /* not listed */

 triple_encode(total); /* internal data build up; not listed  */

 /* initialize para  */
 cosym = 0;
 tosym = 0;
 nounit = 0;
 toerror = 0;

 /* for every training pair in the tdata_internal */
 tdpi = tdata_internal_1;
 while (tdpi->key_word[0] != '\0')  {  /* while  3 */

 /* if part assign for bank2  */
     if (strcmp(tdpi->key_word,"if") == 0) {
  strcpy(bankn2,tdpi->evnum);
  for (i=0;i<rsize;i++)  bank2[i] = tdpi->trep[i];
  }
     else printf("\n no if  part");

    tdpi++;

  /* then  part for bank1 and bank2 */
  while(strcmp(tdpi->key_word,"if")!= 0 && tdpi->key_word[0] != '\0') { /* while 4 */
  if (strcmp(tdpi->key_word,"then") == 0 ||
      strcmp(tdpi->key_word,"follows") == 0) {
strcpy(bankn3,tdpi->evnum);
        for (i=0;i<rsize;i++) bank3[i] = tdpi->trep[i];
}
 else printf("\n no then part");

 if (strcmp(tdpi->key_word,"then") == 0)
     for (i=0;i<csize;i++) bank1[i] = 0;

prop(total); /* verify -- weight freezed */

       for (i=0;i<csize;i++) bank1[i] = hidden[i];

      tdpi++;
} /* while  4 */

  } /* while  3 */

   /* performance statistics */
   correct = (float) cosym / (float) tosym;
   eavg = toerror / (float) nounit;
   printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
   printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* main */
```

G.2 Weight damage resistance

This section lists a group of codes which were used for weight damage resistance experiments described in section 5.4.2. These programs are slight variations of the statistics-gathering programs, so only two programs are listed here as examples of necessary modifications.

```
/*********** for triple-encoder *****************************/
/* weight damage resistance for triple-encoder network module
   data: tdata-te into memory
   input symbol dictionary: global-dict, conbol-case */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define iosize lsize+2*rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 100 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_te 400 /* number of triples in tdata-te  */

#define dfile "tdata-te"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_te {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_te_l[nums_td_te],*tdp1;

/* previous event rep holder  */
char preev[nsize];
float preevr[rsize];

/* ARPDP network accessory */
char repn[nsize],linkn[nsize],noden[nsize]; /* input name holder */
float rep[rsize],link[lsize],node[rsize]; /* input rep holder  */
```

```
 /* event encoder/decoder network */
 float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
 float wih_ev[iosize][rsize],who_ev[rsize][iosize];
 float hbias_ev[rsize],obias_ev[iosize];

 /* global file pointer  */
 FILE *fopen(),*dfp,*sfp,*lfp,*wfp;

 /* statistics parameter */
 int cosym,tosym,nounit;
 float correct,toerror,eavg;

 /* driver */
 main()
 {
   int max,total,mtotal;
   int i;
   char tempev[nsize];
   int maxi,maxo,ran1,ran2; /* random number output */

   read_to_store();  /* read from initial gd and case to internal data
        structure; not listed    */
   load_weight(); /* load previous weight for continuous training; not listed  */

  /* read traning data into memory; object hold -1 if embedded */
  read_td_te(); /* not listed */

 /* start weight damage here */
 /* for all damage cycle */
 for (total=0;total<6;total++)  { /* for23 */

 /* initialize para  */
 cosym = 0;
 tosym = 0;
 nounit = 0;
 toerror = 0;

 /* 1% random damage to weight  */
 if (total != 0) {
 maxi = (36*12)/400;
 maxo = (12*36)/400;
 for (i=0;i<maxi;i++) {
 ran1 = (rand()/32767.0)*35;
 ran2 = (rand()/32767.0)*11;
 wih_ev[ran1][ran2] = 0;
 }
 for (i=0;i<maxo;i++) {
 ran1 = (rand()/32767.0)*11;
 ran2 = (rand()/32767.0)*35;
 who_ev[ran1][ran2] = 0;
 }
 } /* if total */

 /* triple encoding through all data; one epoch */
 strcpy(tempev,"xxx");
 tdp1 = tdata_te_1;
 while(tdp1->evnum[0] != '\0') { /* while 2 */

 strcpy(repn,tdp1->evnum);
 strcpy(linkn,casestorel[tdp1->case_role].name);
 if (tdp1->object >= 0) /* object not embedded */
 strcpy(noden,objstorel[tdp1->object].name);
 else
 strcpy(noden,tdp1->object2); /* event; embedded */

 /* if new event */
 if (strcmp(tempev,repn) != 0) {
        strcpy(preev,tempev); /* for embedding */
        for (i=0;i<rsize;i++)
 preevr[i] = rep[i];
```

```
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}

       for (i=0;i<lsize;i++)
link[i] = casestorel[tdp1->case_role].rep[i];

if (tdp1->object >= 0)  /* no recursive event */
          for (i=0;i<rsize;i++)
     node[i] = objstorel[tdp1->object].crep[i];
        else if (strcmp(tdp1->object2,preev) == 0)
  for (i=0;i<rsize;i++)
     node[i] = preevr[i];
        else printf("\error no %s\n",noden);

       prop_ev(total);

       for (i=0;i<rsize;i++)
          rep[i] = hidden_ev[i];

      strcpy(tempev,repn); /* for event group checking */
      tdp1++;
} /* while */

   /* performance statistics */
   printf("\n %d percent damage\n",total);
   printf("\n maxi %d maxo %d ran1 %d ran2 %d\n",maxi,maxo,ran1,ran2);
   correct = (float) cosym / (float) tosym;
   eavg = toerror / (float) nounit;
   printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
   printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* for23 */

} /* main */

/************ for plan-selector ************************/
/* weight damage resistance for plan-selector network module
   data: tdata-ps into memory
   input symbol dictionary: global-dict, conbol-case
   input weight file: weight-te  */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define csize 30 /* context size */
#define isize csize+rsize
#define osize rsize+1
#define iosize rsize+lsize+rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 150 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_ps 1000 /* number of triples in tdata-te  */
#define nums_td_in 400 /* number of internal data */

#define dfile "tdata-ps"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"
#define wfile2 "weight-ps"

/* global dictionary  */
struct objstore {
   char name[nsize];
```

```
    float crep[rsize]; /* current rep  */
    } objstorel[nums_obj],*symp_obj;

 struct casestore {
    char name[nsize];
    float rep[lsize];
    } casestorel[nums_c],*casep;

 /* training data store ; each entry is array number in the dictionary */
 struct tdata_ps {
   char evnum[nsize];
   int case_role;
   int object;
   char object2[nsize]; /* for embedded event */
   } tdata_ps_l[nums_td_ps],*tdp1;

 /* internal training data */
 struct tdata_internal {
   char key_word[nsize];
   char evnum[nsize];
   float trep[rsize];
   } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

 /* triple encoder related network */
 /* for triple encoder and intput name holder */
 char repn[nsize],linkn[nsize],noden[nsize];
 float rep[rsize],link[lsize],node[rsize];

 /* event encoder/decoder network */
 float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
 float wih_ev[iosize][rsize],who_ev[rsize][iosize];
 float hbias_ev[rsize],obias_ev[iosize];

 /* main network */
 /* current network holder */
 char bankn1[nsize],bankn2[nsize],bankn3[nsize],bankn4[nsize];
 float bank1[csize],bank2[rsize],bank3[rsize],bank4; /* input rep holder  */
 /* plan-selector main network */
 float in[isize],out[osize],hidden[csize],teach[osize];
 float wih[isize][csize],who[csize][osize];
 float hbias[csize],obias[osize];

 /* global file pointer  */
 FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

 /* statistics parameter */
 int cosym,tosym,nounit;
 float correct,toerror,eavg;

 /* driver */
 main()
 {
   int total,mtotal;
   int i,j,jj;
   float ran;
   int maxi,maxo,ran1,ran2; /* random number output */

   load_weight_te(); /* triple encoder weight file */

   read_to_store();  /* read from initial gd and case to internal data
        structure   */
   load_weight(); /* load previous weight for continuous training */

  /* read training data into memory; object hold -1 if embedded */
  read_td_ps();

/* for all damage cycle */
for (total=0;total<6;total++)  { /* for23 */
```

```
/* 1% random damage to weight  */
if (total != 0) {
maxi = (42*30)/100;
maxo = (30*13)/100;
for (i=0;i<maxi;i++) {
ran1 = (rand()/32767.0)*41;
ran2 = (rand()/32767.0)*29;
wih[ran1][ran2] = 0;
}
for (i=0;i<maxo;i++) {
ran1 = (rand()/32767.0)*29;
ran2 = (rand()/32767.0)*12;
who[ran1][ran2] = 0;
}
} /* if total */

/* initialize para  */
cosym = 0;
tosym = 0;
nounit = 0;
toerror = 0;

triple_encode(total); /* internal data build up */

/* for every training pair in the tdata_internal */
tdpi = tdata_internal_1;
while (tdpi->key_word[0] != '\0')  {  /* while  3 */

tdpi2 = tdpi;
/* then part assign for bank3 and bank4  */
while(strcmp(tdpi2->key_word,"then") != 0) tdpi2++;
    if (strcmp(tdpi2->key_word,"then") == 0) {
 strcpy(bankn3,tdpi2->evnum);
 for (i=0;i<rsize;i++)  bank3[i] = tdpi2->trep[i];
 }
    else printf("\n no then part");
   tdpi2++;
   if (strcmp(tdpi2->key_word,"and") == 0) {
   strcpy(bankn4,tdpi2->evnum);
   bank4 = tdpi2->trep[0];
   }
   else printf("\n no then part");

 /* if part for bank1 and bank2 */
 while(strcmp(tdpi->key_word,"then")!= 0) { /* while 4 */
 if (strcmp(tdpi->key_word,"if") == 0 ||
     strcmp(tdpi->key_word,"follows") == 0) {
strcpy(bankn2,tdpi->evnum);
        for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
}
 else printf("\n no if part");

 if (strcmp(tdpi->key_word,"if") == 0)
     for (i=0;i<csize;i++) bank1[i] = 0;


prop(total); /* verify -- weight freezed */

       for (i=0;i<csize;i++) bank1[i] = hidden[i];

      tdpi++;
} /* while  4 */

  while(strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') tdpi++;
  } /* while  3 */

   /* performance statistics */
   printf("\n %d percent damage\n",total);
   printf("\n maxi %d maxo %d ran1 %d ran2 %d\n",maxi,maxo,ran1,ran2);
```

```
      correct = (float) cosym / (float) tosym;
      eavg = toerror / (float) nounit;
      printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
      printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);
  } /* for */

  } /* main */
```

G.3 Unit damage resistance

This section lists a group of codes which were used for unit damage resistance experiments described in section 5.4.1. These programs are also slight variations of the statistics-gathering programs, so only two programs are listed.

```
/********* for triple-encoder ***************************/
/* unit damage resistance for triple-encoder network module
   data: tdata-te into memory
   input symbol dictionary: global-dict, conbol-case */

#include <stdio.h>
#include <math.h>

#define nsize 20 /* number of characters in name */
#define rsize 12 /* representation size */
#define lsize 12 /* case-role size  */
#define iosize lsize+2*rsize
#define idsize 2 /* id bit size */

/* max data size */
#define nums_obj 100 /* number of object symbol in the dictionary */
#define nums_c 30 /* number of case-role */
#define nums_td_te 400 /* number of triples in tdata-te  */

#define dfile "tdata-te"
#define sfile "global-dict"
#define lfile "conbol-case"
#define wfile "weight-te"

/* global dictionary  */
struct objstore {
   char name[nsize];
   float crep[rsize];  /* current rep  */
   } objstorel[nums_obj],*symp_obj;

struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_te {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_te_l[nums_td_te],*tdp1;

/* previous event rep holder  */
char preev[nsize];
float preevr[rsize];

/* ARPDP network accessory */
```

```
char repn[nsize],linkn[nsize],noden[nsize]; /* input name holder */
float rep[rsize],link[lsize],node[rsize]; /* input rep holder  */

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int max,total,mtotal;
  int i,jj;
  char tempev[nsize];
  float ran; /* random number output */

  read_to_store();  /* read from initial gd and case to internal data
       structure   */
  load_weight(); /* load previous weight for continuous training */

 /* read traning data into memory; object hold -1 if embedded */
 read_td_te();

/* start unit damage here */
/* for all damage cycle */
for (total=0;total<6;total++)  { /* for23 */

/* initialize para  */
cosym = 0;
tosym = 0;
nounit = 0;
toerror = 0;

/* damage one unit */
jj = 12-total;
symp_obj = objstorel;
while(symp_obj->name[0] != '\0') {
  if (jj <12) symp_obj->crep[jj] = 0.5;
  symp_obj++;
}


/* triple encoding through all data; one epoch */
strcpy(tempev,"xxx");
tdp1 = tdata_te_1;
while(tdp1->evnum[0] != '\0') { /* while 2 */

strcpy(repn,tdp1->evnum);
strcpy(linkn,casestorel[tdp1->case_role].name);
if (tdp1->object >= 0) /* object not embedded */
strcpy(noden,objstorel[tdp1->object].name);
else
strcpy(noden,tdp1->object2); /* event; embedded */

/* if new event */
if (strcmp(tempev,repn) != 0) {
       strcpy(preev,tempev); /* for embedding */
       for (i=0;i<rsize;i++)
preevr[i] = rep[i];
       for (i=0;i<rsize;i++)
rep[i] = 0.5; /* init ev rep */
}
```

```
        for (i=0;i<lsize;i++)
 link[i] = casestorel[tdp1->case_role].rep[i];

 if (tdp1->object >= 0)  /* no recursive event */
           for (i=0;i<rsize;i++)
      node[i] = objstorel[tdp1->object].crep[i];
         else if (strcmp(tdp1->object2,preev) == 0)
   for (i=0;i<rsize;i++)
      node[i] = preevr[i];
         else printf("\error no %s\n",noden);

        prop_ev(total);

        for (i=0;i<rsize;i++)
           rep[i] = hidden_ev[i];

       strcpy(tempev,repn); /* for event group checking */
       tdp1++;
 } /* while */

    /* performance statistics */
    printf("\n %d unit damage\n",total);
    correct = (float) cosym / (float) tosym;
    eavg = toerror / (float) nounit;
    printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
    printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

 } /* for23 */

 } /* main */


 /*********** for plan-selector ************************/
 /* unit damage resistance for plan-selector network module
    data: tdata-ps into memory
    input symbol dictionary: global-dict, conbol-case
    input weight file: weight-te  */

 #include <stdio.h>
 #include <math.h>

 #define nsize 20 /* number of characters in name */
 #define rsize 12 /* representation size */
 #define lsize 12 /* case-role size  */
 #define csize 30 /* context size */
 #define isize csize+rsize
 #define osize rsize+1
 #define iosize rsize+lsize+rsize
 #define idsize 2 /* id bit size */

 /* max data size */
 #define nums_obj 150 /* number of object symbol in the dictionary */
 #define nums_c 30 /* number of case-role */
 #define nums_td_ps 1000 /* number of triples in tdata-te  */
 #define nums_td_in 400 /* number of internal data */

 #define dfile "tdata-ps"
 #define sfile "global-dict"
 #define lfile "conbol-case"
 #define wfile "weight-te"
 #define wfile2 "weight-ps"

 /* global dictionary  */
 struct objstore {
    char name[nsize];
    float crep[rsize]; /* current rep  */
    } objstorel[nums_obj],*symp_obj;
```

```
struct casestore {
   char name[nsize];
   float rep[lsize];
   } casestorel[nums_c],*casep;

/* training data store ; each entry is array number in the dictionary */
struct tdata_ps {
  char evnum[nsize];
  int case_role;
  int object;
  char object2[nsize]; /* for embedded event */
  } tdata_ps_l[nums_td_ps],*tdp1;

/* internal training data */
struct tdata_internal {
  char key_word[nsize];
  char evnum[nsize];
  float trep[rsize];
  } tdata_internal_l[nums_td_in],*tdpi,*tdpi2;

/* triple encoder related network */
/* for triple encoder and intput name holder */
char repn[nsize],linkn[nsize],noden[nsize];
float rep[rsize],link[lsize],node[rsize];

/* event encoder/decoder network */
float in_ev[iosize],out_ev[iosize],hidden_ev[rsize],teach_ev[iosize];
float wih_ev[iosize][rsize],who_ev[rsize][iosize];
float hbias_ev[rsize],obias_ev[iosize];

/* main network */
/* current network holder */
char bankn1[nsize],bankn2[nsize],bankn3[nsize],bankn4[nsize];
float bank1[csize],bank2[rsize],bank3[rsize],bank4; /* input rep holder  */
/* plan-selector main network */
float in[isize],out[osize],hidden[csize],teach[osize];
float wih[isize][csize],who[csize][osize];
float hbias[csize],obias[osize];

/* global file pointer  */
FILE *fopen(),*dfp,*sfp,*lfp,*wfp,*wfp2;

/* statistics parameter */
int cosym,tosym,nounit;
float correct,toerror,eavg;

/* driver */
main()
{
  int total,mtotal;
  int i,j,jj;
  float ran;

  load_weight_te(); /* triple encoder weight file */

  read_to_store();  /* read from initial gd and case to internal data
       structure   */
  load_weight(); /* load previous weight for continuous training */

 /* read training data into memory; object hold -1 if embedded */
 read_td_ps();

/* start unit damage here */
/* for all damage cycle */
for (total=0;total<6;total++)  { /* for23 */

/* damage one unit */
jj = 12-total;
symp_obj = objstorel;
```

```
  while(symp_obj->name[0] != '\0') {
    if (jj <12) symp_obj->crep[jj] = 0.5;
    symp_obj++;
  }

  /* initialize para  */
  cosym = 0;
  tosym = 0;
  nounit = 0;
  toerror = 0;

  triple_encode(total); /* internal data build up */

  /* for every training pair in the tdata_internal */
  tdpi = tdata_internal_1;
  while (tdpi->key_word[0] != '\0')  {  /* while  3 */

  tdpi2 = tdpi;
  /* then part assign for bank3 and bank4  */
  while(strcmp(tdpi2->key_word,"then") != 0) tdpi2++;
      if (strcmp(tdpi2->key_word,"then") == 0) {
   strcpy(bankn3,tdpi2->evnum);
   for (i=0;i<rsize;i++)  bank3[i] = tdpi2->trep[i];
   }
      else printf("\n no then part");
    tdpi2++;
    if (strcmp(tdpi2->key_word,"and") == 0) {
    strcpy(bankn4,tdpi2->evnum);
    bank4 = tdpi2->trep[0];
    }
    else printf("\n no then part");

  /* if part for bank1 and bank2 */
  while(strcmp(tdpi->key_word,"then")!= 0) { /* while 4 */
  if (strcmp(tdpi->key_word,"if") == 0 ||
      strcmp(tdpi->key_word,"follows") == 0) {
 strcpy(bankn2,tdpi->evnum);
        for (i=0;i<rsize;i++) bank2[i] = tdpi->trep[i];
 }
  else printf("\n no if part");

  if (strcmp(tdpi->key_word,"if") == 0)
      for (i=0;i<csize;i++) bank1[i] = 0;


 prop(total); /* verify -- weight freezed */

        for (i=0;i<csize;i++) bank1[i] = hidden[i];

       tdpi++;
 } /* while  4 */

  while(strcmp(tdpi->key_word,"if") != 0 && tdpi->key_word[0] != '\0') tdpi++;
  } /* while  3 */

   /* performance statistics */
   printf("\n %d unit damage\n",total);
   correct = (float) cosym / (float) tosym;
   eavg = toerror / (float) nounit;
   printf("\n cosym: %d tosym: %d correct: %.5f\n",cosym,tosym,correct);
   printf("\n toerror: %.3f nounit: %d eavg: %.7f\n",toerror,nounit,eavg);

} /* for */

} /* main */
```