AD-A056 080 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 6/4
SCRIPT APPLICATION: COMPUTER UNDERSTANDING OF NEWSPAPER STORIES--ETC(U)
JAN 78 R E CULLINGFORD N00014-75-C-1111
UNCLASSIFIED RR-116 NL

1 OF 3
ADA
056080

LEVEL # (10) 2

AD A056080

(6)

SCRIPT APPLICATION: COMPUTER UNDERSTANDING OF NEWSPAPER STORIES.

(11) January 1978 (12) 213 p.

(9) Research Report #116

(10) Richard Edward/Cullingford

(14) RR-116

(15) N00014-75-C-1111

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

AD No.
DDC FILE COPY

DDC
RECEIVED
JUL 11 1978
A

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

78 07 07 044

407 051

JOB

This work was presented to the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy.

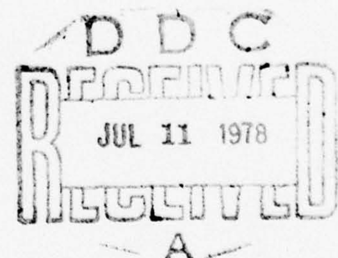
Research Rept. No. 116
Table of Figures, Page XIII is not a part of the report and should not have been listed per Mr. Gordon Goldstein, ONR/Code 437. Information was furnished by contractor.

SCRIPT APPLICATION: COMPUTER UNDERSTANDING OF NEWSPAPER STORIES

January 1978

Research Report #116

Richard Edward Cullingford



This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored under the Office of Naval Research under contract N00014-75-C-1111.

78 07 07 044

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #116	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Script Application: Computer Understanding of Newspaper Stories		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Edward Cullingford		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-1111
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University Department of Computer Science 10 Hillhouse Ave., New Haven, Conn. 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE January 1978
		13. NUMBER OF PAGES 203
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Program Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

text understanding, psychological modelling, knowledge structures, frame systems, conceptual dependency, pattern-directed inference

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The report describes a computer story understander which applies knowledge of the world to comprehend what it reads. The system, called SAM, reads newspaper articles from a variety of domains, then demonstrates its understanding by summarizing or paraphrasing the text, or answering questions about it.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

-- OFFICIAL DISTRIBUTION LIST --

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Code 102IP Arlington, Virginia 22217	6 copies
Office of Naval Research Branch Office - Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office - Chicago 536 South Clark Street Chicago, Illinois	1 copy
Office of Naval Research Branch Office - Pasadena 1030 East Green Street Pasadena, California 91106	1 copy
Mr. Steven Wong Administrative Contracting Officer New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy
Naval Research Laboratory Technical Information Division Code 2627 Washington, D. C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps Code RD-1 Washington, D. C. 20380	1 copy

Office of Naval Research
Code 455
Arlington, Virginia 22217
1 copy

Office of Naval Research
Code 458
Arlington, Virginia 22217
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, California 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research and Development Center
Computation and Mathematics Department
Bethesda, Maryland 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Board
OP-916D
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
OP-91T
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Advanced Research Projects Agency
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209
1 copy

Professor Omar Wing
Columbia University in the City of New York
Department of Electrical Engineering and
Computer Science
New York, New York 10027
1 copy

Office of Naval Research
Assistant Chief of Technology
Code 200
Arlington, Virginia 22217
1 copy

ABSTRACT

Script Application:
Computer Understanding of Newspaper Stories

Richard Edward Cullingford

Yale University 1977

This thesis describes a computer story understander which applies knowledge of the world to comprehend what it reads. The system, called SAM, reads newspaper articles from a variety of domains, then demonstrates its understanding by summarizing or paraphrasing the text, or answering questions about it. Since the knowledge structures SAM works with are conceptual and language-free, we have been able to add a limited machine-translation capability to SAM, as well.

SAM's knowledge of the world is encoded through the use of a representational construct called a Script. Scripts describe the stereotyped activities characteristic of socially ritualized situations such as going to stores, museums and restaurants, taking business trips and vacations, and attending banquets and birthday parties. SAM consults its Scripts to recognize the events a particular text refers to, to identify the participants in these events, and to fill in other events, not explicitly mentioned by a story, which can be plausibly inferred to have happened. In this process, it moves Scripts in and out of active memory on the basis of predictions it makes about what may be seen next.

SAM represents an attempt to build a complete, working story understander exploiting an important source of knowledge about the world to find the connections which make a text "coherent," and to build a memory representation for the text from which natural-language outputs can be generated which indicate a reasonable depth of understanding. Since Scripts model a knowledge structure that people constantly apply, both to cope with the world and to understand what they read, SAM embodies a theory of context and how context is to be used in the process of understanding.

ACCESSION for
NTIS White Section ☒
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

PREFACE

Stories constantly refer to people, places, things and events in the world. The task of story understanding, whether undertaken by a person or by a computer program, requires numerous sources of world knowledge and methods for getting at the appropriate parts of that knowledge as required. For example, we may need to know about how physical forces operate, or about the rules governing people's behaviour in various social circumstances.

A computer story understander, to achieve a reasonable depth of comprehension, has to do many of the things that a person does as he reads a text. It must recognize the context that a given story is set in. It has to identify each and every reference to an actor in the story. Most importantly, it must make explicit the things that are only implicit in a story, to fill in the things the storyteller left out. Only in this way will the representation the understander retains after the story has been read be sufficiently rich to allow it to fashion appropriate summaries, and to answer questions properly.

The process of inference in understanding a story is always informed and controlled by the understander's consulting what he knows to be true about the world. In what follows we describe an attempt to give the computer a source of detailed world knowledge, and procedures for using that knowledge to comprehend.

ACKNOWLEDGMENTS

First of all, I'd like to thank my thesis advisor, Professor Roger Schank, for suggesting the idea of the Script Applier to me, and then spending the time and effort to keep me on the straight and narrow. It's been a pleasure to work with a man who knows where he is going, and knows how to manage a large research team for maximum benefit to everybody. It's also been an intellectual adventure to learn, from him, the careful mixture of freewheeling intuition and disciplined programming that one needs to do research in Artificial Intelligence.

I owe a special debt of gratitude to Yale's Department of Engineering and Applied Science, especially to Professors Werner Wolf, K. S. Narendra and Franz Tuteur, for encouraging me as I pursued this topic in Artificial Intelligence, so unlike anything else the Department does. Professors Narendra and Tuteur also have my thanks for serving on my reading committee. At Yale, when a Department says it supports "interdisciplinary work," it means what it says.

Professor Alan Perlis got me interested in AI in the days before Professor Schank arrived at Yale, and was rewarded for his trouble by having to serve on my committee. Dr. Chris Riesbeck, by admonition and example, taught me many of the techniques of AI research in natural-language processing. He also read a draft version of this manuscript, and contributed several patches of clarity to my otherwise unwieldy prose.

Nearly everyone in the Yale Artificial Intelligence Project has worked at one time or another on the story understander, SAM, discussed here. Many of the results I present came out of marathon sessions of "crunching" by various teams of hackers. I want to express my thanks to my friends and co-workers: Jaime Carbonell, Gerald DeJong, Anatole Gershman, Richard Granger, Janet Kolodner, Wendy Lehnert, James Meehan, Warren Odom, Richard Proudfoot, Mallory Selfridge, Walter Stutzman and Robert Wilensky.

Finally, there's a special word of appeciation for the Yale DECsystem-10 computer, serial number 152, for a gritty performance in the role of Mother Nature.

The research described in this thesis was supported by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-1111.

For Sylvia:
Who helped me preserve my sanity,
at the expense of hers.

TABLE OF CONTENTS

ABSTRACT
PREFACE ii
ACKNOWLEDGMENTS iii
DEDICATION iv
TABLE OF CONTENTS v
~~TABLE OF FIGURES~~ . . . . ~~xiii~~

Chapter 1: What This Thesis Is About

1.1 A Computer Program That Understands Stories 1
1.2 Knowledge about Context: Scripts 2
1.3 What SAM Does 5
1.4 Scripts and Script Applying in Historical Perspective 7
1.5 An Overview of Script Application 10
1.6 An Annotated Example 19
1.7 Outline of the Thesis 30

Chapter 2: Script Structure

2.1 Introduction 33
2.2 Riding the Subway 35
2.3 Script Variables and Patterns 39
2.4 Episodes and Pathvalue 44
2.4.1 The Structure of Episodes 44
2.4.2 Connecting Episodes Together 48
2.4.3 Pathvalue 50
2.4.4 More About Patterns 52
2.5 Scenes and Tracks 53
2.6 Permanent Memory Structures 56
2.6.1 Setting and Point of View 57
2.6.2 Script Preconditions 60
2.6.3 Script Headers 62
2.6 3.1 Example Headers for $SUBWAY 66
2.7 Summing Up 67

Chapter 3: Managing Scripts

3.1 Why This is a Problem 69
3.2 Organizing Expectations about Stories 71
3.3 Connections Among Scripts 75
3.3.1 Scripts in Simple Sequential Relation 76
3.3.2 Scripts Occurring in the Same Place 78
3.4 Fitting Scripts Together 85
3.4.1 Scripts Involving Organizations and Forces 87
3.4.2 Transactions 88
3.4.3 Natural-Force Scripts 90
3.4.4 Situations 92

3.5 Summing Up 97

Chapter 4: Script Application: The Basic Cycle

4.1 Introduction 99
4.2 Story-Telling Conventions 100
4.3 Internalizing Conceptualizations 101
4.4 Choosing a Context 103
4 5 Pattern Matching 106
4.5.1 The Backbone Match 107
4.5.2 Rolefit 108
4.5.3 Rolemerge 109
4.6 Making and Unmaking Predictions 110
4.7 Instantiating Episodes 113
4.8 Changing Contexts 114
4.9 Processing Newspaper Stories 115

Chapter 5: Inferencing in SAM

5.1 Introduction 119
5.2 Classes of Inferences 120
5.3 Immediate-Result Inferences 120
5.4 Mental-Act Inferences 121
5.4.1 Perception and Remembering 121
5.4 2 Authority Announcements 122
5.5 Locational Inferences 123
5.5.1 Transitivity of Proximity 123
5.5 2 Enclosure 124
5.6 Movement Inferences 124
5.6.1 Movement of Personal Possessions 125
5.6.2 Conveyance Inferences 126
5 7 Agency Inferences 127

Chapter 6: A Very Detailed Example

6.1 Introduction 128
6 2 Understanding the Story 130
6.2.1 Finding the Inputs 130
6.2.2 Making the Story Representation 152
6.3 Answering Questions 158

Chapter 7: Representing Conceptual Nominals

7.1 Motivation 163
7.2 Actors 168
7.2 1 Persons 169
7.2.2 Groups 170
7.2.3 Organizations 173
7.2 4 Polities 174
7.2.5 Forces 175
7.3 Physical Objects 177
7.3.1 Simple Objects 177
7.3.2 Structured Objects 178
7.4 Places 179
7.4.1 Simple Locales 179

7.4.2 Geographical Features 180
7.4.3 Links 180
7.5 Miscellaneous PPs 181

Chapter 8 Finale

8.1 Why Did We Do This? 182
8.2 What Else Could Be Done? 183
8.2 1 A Laboratory for Inference 183
8.2.2 A Model of Reading 184
8.2.3 Scripts and Plans 185

APPENDIX 1· Representation and Notation 187

APPENDIX 2: More Output from SAM 195

BIBLIOGRAPHY 200

Chapter 1
What This Thesis Is About

1.1 A Computer Program That Understands Stories

What do you have to know, and how should you use what you know, in order to read a story? This thesis describes an attempt to answer these questions by building a computer model which applies knowledge of the world to understand texts it is given to read. The task of story understanding has two aspects which the model will have to account for: (1) the actual process of understanding; and (2) what the understander does with the information acquired. Our computer simulation should be able to read simple stories the way people do: "left-to-right," in one pass, a sentence at a time. And, since people can demonstrate their comprehension of stories by summarizing them, paraphrasing them, or answering questions about them, the simulation should be able to do the same.

If we wish to build a computer story understander, we must search for ways of classifying and organizing world knowledge so it can be used by a computer. Organization is the crucial issue for two closely related but opposing reasons: (1) a program which achieves a reasonable depth of understanding as it reads will certainly require an enormous amount of knowledge drawn from very different domains; but (2) the particular information needed at any point during comprehension must somehow be made accessible without a large amount of search.

This thesis is about the organization of world knowledge for the particular task of story understanding by computer. On the one hand, we describe ways of setting up a data base containing knowledge of the world without having an information explosion. On the other, we discuss how such a data base can be arranged for reasonably efficient retrieval. Our computer model, SAM, exploits knowledge of a particularly important kind to read stories referring to a variety of knowledge domains. It solves the problems that a text presents by finding the connections that make the text "coherent." Then it shows that it has "understood" the text, in a fairly deep sense, by answering questions about it, summarizing it, or paraphrasing it. Additionally, SAM's output can be expressed in a variety of natural languages, so SAM is a kind of Machine Translation system as well.

1.2 Knowledge about Context: Scripts

Broadly speaking, stories describe the working out of problems by actors in some context (Note 1). An intelligent story understander, therefore, will have to embody a theory of what constitutes a context, and how one is used. If competing contexts are available, the understander will also need rules to decide which takes precedence.

Our computer model, SAM, implements a partial theory of how context can be applied in text comprehension. The basis for the theory is a representational construct, devised by Schank and Abelson [34], called a Script. Use of the Script enables us to encode for the computer our culturally shared knowledge of the stereotyped events that occur in such socially ritualized activities as going to stores, restaurants and museums; riding trains and subways; attending plays and banquets; and playing games or driving cars. We believe that people themselves have Scripts, acquired by repeated exposure to situations, which tell them what can happen in a situation, what follows what, and when; what roles various people and things typically have; and, most importantly, what the person is expected to do. Aside from their "operational" value, people use Scripts in "cognitive" activities, e. g., to help in understanding references to the situation in the things they read.

As an example of a typical Script, consider the activities which go with eating in a restaurant. The ordinary course of affairs is that the patron enters, is seated, and orders a meal. The meal is then prepared and served, and the patron eats it. Finally, the patron pays the bill and leaves. Each of these activities is described by a stereotyped chain of events, which prescribes the order in which things happen, and the people and objects participating in the action. Entering the restaurant, looking for a table, walking over to one and sitting down comprise one such event-chain, or "episode." Episodes in Scripts are organized as causal chains [28,34]. Each event has resulting states which in turn become the enabling conditions for further events to occur. For example, one must physically be inside a restaurant before one can look for a table. Seeing an empty table enables walking over to it. As a result of walking to a table, one can sit down at it. The

----

1. Interpreting people's actions in terms of their underlying motivations is an extremely difficult problem for which no complete theory yet exists. The work of Schank and Abelson [34] represents one Artificial Intelligence (AI) approach to this domain, based on a small set of very general procedures called Plans. Plans characterize people's standard desires and their preferred methods for satisfying them. An early version of Plan theory is implemented in a computer story understander called PAM [41]. Schmidt and Sridharan [37] have developed a second representational system for dealing with this problem, based on a construct called a "plan schema." These researchers have been developing an AI system, called BELIEVER, which uses plan schemata to model in a psychologically plausible manner how people use their beliefs about other people to arrive at an intentional explanation of observed behavior.

episodes in Scripts are connected together at "turning points," where alternative paths for accomplishing a given Scriptal activity are available. A turning point in the restaurant Script occurs when the order is placed, since the order may either be accepted, or rejected for some reason.

Each event in a Script contains references to the people ("roles") who have well-known duties there, and the things ("props") they use when engaged in their duties. For example, "patron" fills a defined function in the restaurant Script, as does "waiter," "cashier" and "table." Since we believe that the memory structures which embody a person's understanding of events in the world are "conceptual," that is, language-free, Scriptal events are encoded according to the rules of Conceptual Dependency meaning representation [32]. Therefore, a Script is a large network of interconnected Conceptualizations, each containing "slots," or Script variables, with requirements on the real-world people and things which can fill the slots.

A given story about a situation will refer to, or "instantiate," only some of its episodes. Script-based story understanding, accordingly, is a process of constructing a "trace" or "scenario" through a given Script which contains both the events explicitly mentioned in the story, and those which can be inferred to have happened. The Script is used, that is, not only to recognize what has been read, but also to fill in what was left out. Continuing our restaurant example, let's suppose that a story referring to the restaurant Script begins:

John walked into a restaurant. He asked for some lasagna.

The Script is "invoked," or activated, by the first sentence, since the understander is now primed to hear more about things which go with eating in a restaurant. When the second sentence is read, we understand that the event being instantiated is one way of carrying out the "ordering" activity which always goes on in restaurants. (We call the collection of possible ways an important activity such as ordering can be accomplished a "scene" of the Script.) We assume that John is asking the restaurant to prepare some food for him which he will eat there. Note that "asked for some lasagna" would have quite a different meaning if John were in a food store.

The Script also tells us that the order is directed to a unnamed "waiter," that John is probably seated at a table, and that the waiter is standing there, too. In fact, we infer that all the actions which are appropriate for finding a table and sitting down have already taken place. The structure of the restaurant Script tells us which particular pieces of causal chain from the "seating" scene to fill in between the acts of entering and ordering.

Our computer model of this process is called a Script Applier. It is the heart of a Script-based story understander called SAM (Script Applier Mechanism), which reads newspaper articles referring to events such as car accidents, train wrecks and state visits, and then summarizes them or answers questions about them. We have tried to make SAM read newspaper stories in a way which simulates an average adult

reading the same material. We assume that the task is careful reading, rather than skimming; and that understanding relies on general world knowledge, embodied in the appropriate Script, rather than specialist knowledge of a domain. For example, a reference to an ambulance ride in a car-accident story would be understood in terms of an ordinary person's knowledge of ambulances and emergency rooms, not a doctor's. Additionally, we assume that the understander is emotionally neutral about the stories read. The people mentioned are not personally known to the reader, and the events described are not interpreted in terms of any personal belief system. (Note 2.)

Here is an example of the kind of stories SAM has read, which illustrates the level of understanding it can achieve. (Story 1.1 is a slightly simplified version of a newspaper article about a car accident from the New Haven Register.) The text shown is actual input and output from the computer (Note 3).

> Story 1.1:
> Friday evening a car swerved off Route 69. The vehicle struck a tree. The passenger, a New Jersey man, was killed. David Hall, 27, was pronounced dead at the scene by Dr Dana Blauchard, medical examiner. Frank Miller, 32, of 593 Foxon Rd, the driver, was taken to Milford Hospital by Flanagan Ambulance. He was treated and released. No charges were made. Patrolman Robert Onofrio investigated the accident.
>
> English Summary:
> AN AUTOMOBILE HIT A TREE NEAR HIGHWAY 69 FOUR DAYS AGO. DAVID HALL, AGE 27, RESIDENCE IN NEW JERSEY, THE PASSENGER, DIED. FRANK MILLER, AGE 32, RESIDENCE AT 593 FOXON ROAD IN NEW HAVEN, CONNECTICUT, THE DRIVER, WAS SLIGHTLY INJURED. THE POLICE DEPARTMENT DID NOT FILE CHARGES.
>
> Spanish Summary:
> UN AUTO CHOCO CONTRA UN ARBOL CERCA DE LA CARRETERA 69 EL VIERNES AL ANOCHECER. DAVID HALL, DE 27 ANOS, QUE VIVIA EN NEW JERSEY, EL PASAGERO, SE MURIO. FRANK MILLER, DE 32 ANOS, QUE VIVIA EN 593 CALLE FOXON EN NEW HAVEN, CONNECTICUT, EL CHOFER, QUEDO UN POCO HERIDO. EL DEPARTAMENTO DE POLICIA NO ACUSO A NADIE.

----

2. An early version of SAM, designed for simple stories about restaurants, busses and subways, is described in [36]. Methods developed for building and applying Scripts as SAM evolved are discussed in [10] and [11].

3. SAM is programmed in Irvine LISP and MLISP for a DECsystem-10 computer (KA-10 processor, 192K of core memory). Processing of Story 1.1, including summarization and question-answering, took about 8 minutes of CPU time, with most of the system's code executed interpretively.

Question-Answering Output:
Q1: Was anyone killed?
A1: YES, DAVID HALL DIED.

Q2: Was anyone hurt?
A2: YES, FRANK MILLER WAS SLIGHTLY INJURED.

Q3: Why was Miller hurt?
A3: BECAUSE THE AUTOMOBILE HIT THE TREE.

Q4: Did Miller go to the hospital?
A4: YES, HE WAS IN THE MILFORD HOSPITAL.

Q5: How did Miller get to the hospital?
A5: THE FLANAGAN AMBULANCE COMPANY TOOK HIM TO THE MILFORD HOSPITAL.

1.3 What SAM Does

What is SAM doing as it reads a story such as (1.1)? This text is understood by SAM as referring to the motor-vehicle-accident Script, $VEHACCIDENT (Note 4). The accident Script is more complicated than the restaurant Script, but the principles by which it is constructed are the same. There are causal chains, or episodes, containing well-defined roles, which characterize this situation, and the episodes are grouped into scenes by the importance of the activity in the accident context. There is, for example, a "crash" scene, describing the ways in which a vehicle can go out of control and run into something. Script variables in these events include the functions for "vehicle," "driver" and "obstruction."

Not only is $VEHACCIDENT built like a simpler Script, but stories about it are processed by SAM in the same way as stories referring, for example, to restaurants. The basic strategy is to recognize the event which the current input refers to, and then to fill in the things which have been left out, both missing roles and uninstantiated, connecting episodes. In Story 1.1, SAM has used its knowledge of typical happenings in a crash and its aftermath (treatment, investigation, dealings with the insurance company, etc.) to make explicit the connections, or _inferences_, which are only implicit in the text.

As we mentioned above, the most important kind of inference SAM makes is _filling in a causal chain_. The structure of $VEHACCIDENT tells the Script Applier which sequence of causally connected events to select and instantiate between explicitly mentioned events. In Story 1.1, we read about a crash, then about a person being taken to the hospital. How can these events be connected? SAM applies its knowledge about car

----

4. In this thesis, upper case names preceded by "$" are names of Scripts. Names preceded by "&" refer to Script variables, that is, to props and roles from the Script. For example, &BUSDRIVER is the "bus driver" role from the bus Script, $BUS.

accidents and the functions of ambulance companies (the ambulance Script) to fill in the probable causal relations that someone saw the crash and called an ambulance, that the ambulance came to the scene, that the ambulance attendants placed the person on a stretcher and put the stretcher into the ambulance, etc. It also makes the crucial connection, never stated in the story, that the person who was taken to the hospital in (1.1) must have been injured in the crash. The reason it can do this is because it "knows" what ambulances and hospitals are for, in the sense that the appropriate Scripts connect together for the purpose of aiding people who are sick or hurt, and cannot get to the hospital under their own power. A necessary part of filling in causal chains is role-instantiation: specifying the necessary properties a Picture Producer (PP: entities, such as people, places and things, having a "static" memory representation [32]) must have to fill a specified role in an event. An example of role-instantiation can be seen in the summary of (1.1), which asserts that the "police department," as the organization responsible for investigations and arrests, chose not to file charges in this instance.

Another basic class of inference is reference specification. The need for this process arises when a Script variable which has already been bound to a PP is mentioned in a subsequent input. At this point a decision has to be made: Can the new PP be an instance of an old one? The classic reference problem occurs with pronouns, e. g., can "he" be the "John" we heard about earlier? In newspaper stories, a more complicated reference problem arises because of what we call "paraphrastic reference:" the use of arbitrarily complex noun groups to refer to the same PP. An example is recognizing that the man from New Jersey mentioned in the third sentence of (1.1) must be David Hall, age 27.

SAM also uses the time/place setting of a story for inferences about where things are happening and how long they take. A Script's causal chains have associated default values for the length of time they typically use up, or where they would be expected to occur. SAM uses these defaults in Story 1.1 to infer that the crash must have occurred on the same day as the "swerve," namely Friday evening. Cars simply cannot stray from roads for very long (on the order of seconds) before encountering an obstruction. (SAM inserts the phrase "four days ago" because it is arranged, by convention, to be reading newspaper stories on Tuesdays.) Similarly, the crash must have occurred "near Route 69," although the story does not explicitly say so. This is because roads are provided with all sorts of nearby objects for cars to run into.

Finally, SAM makes various kinds of delayed inferences. Sometimes a story will leave a point of interest to a reader hanging for a while, only clearing up the problem in a later sentence. The inferences needed in these cases have the nature of "demons" [7], hovering around waiting for a feature of an input that satisfies their expectations. In car accidents, for example, we want to know whether anyone was killed; if someone was hurt, how badly; what the police did, etc. In Story 1.1, although we know that the man taken to the hospital was hurt -- this is what ambulances do --, we cannot initially be sure how seriously. Will the person be operated on and spend some time in the hospital? Will he be so badly damaged as to die there? This decision cannot be made until

the sentence about "treated and released" is seen, at which point SAM concludes that he must not have been too badly hurt.

1.4 Scripts and Script Applying in Historical Perspective

Early attempts to program computers to understand natural language, despite the initial optimism of the researchers, met with only limited success. For example, the problem of machine translation between languages was viewed as being essentially one of supplying the computer with dictionaries and grammars of sufficiently high quality. The ultimate failure of the translation projects of the 1950's, as described, for example, by Bar-Hillel [2], can be directly attributed to a reliance on formal, syntactic methods. Knowledge of the world was almost completely lacking in the systems actually built.

Natural-language processing research undertaken from the Artificial Intelligence (AI) viewpoint has approached the crucial knowledge problem from two distinct directions. One approach is to set up a "micro-world," that is, to limit the knowledge domain to be addressed enough that programs can be written which can "understand" inputs in some nontrivial sense, and generate appropriate responses. For example, Woods' LSNLIS [46] answers questions about the composition and properties of a collection of moon rocks. Winograd's SHRDLU [43] engages in conversation with its user concerning the manipulation of blocks in a simple visual scene. Brown and Burton's SOPHIE [6] tutors a student in troubleshooting a simulated electronic circuit via a "mixed-initiative" dialog.

AI systems like these clearly demonstrate the increases in power that can be achieved by incorporating more and more domain-specific knowledge. The obvious worry about "micro-worlds," however, is extensibility. It is never clear whether techniques which work well in one world will go over into another. The problem is compounded in SHRDLU and SOPHIE because most of the knowledge possessed by these programs is imbedded in procedures rather than in the data on which the procedures run.

A second approach is to attempt to develop generalized representational constructs which can handle large chunks of knowledge in a standardized way. One well-known scheme which takes this viewpoint is Minsky's Frame-system [19]. Frame-systems are built up out of static, hierarchically organized descriptions ("frames") of stereotyped situations, such as being in a store or seeing a well-known room, which are associated by virtue of sharing "terminals" or "slots" to be filled by objects with prespecified properties. A visual Frame-system for a cube, for example, would consist of frames describing the cube as seen from various orientations. Shifts in perspective are then mirrored by retrieval of the appropriate frame. The notion of the Script, although independently developed, takes the same standpoint as the Frame-system. It seeks to represent a variety of knowledge domains in terms of a hierarchical data structure containing slots or gaps which prescribe what things in an input can fill them; and how to construct a default for one if the input does not mention it at all.

What does a representational system like this buy us? First of all, since descriptions are naturally declarative in form, a system embodying a Script- or Frame-like approach should be easier to extend than a procedure-based system. Extending a system whose knowledge is procedurally imbedded always runs into the problem of side-effects. It is a commonplace among programmers that a minor change to an existing program can have far-reaching effects on other parts of the program. Extensive changes are even harder to implement, especially in AI applications, where often it is not clear how to split a given process into independent, "structured" subprocesses, or whether such a split is possible at all. A related problem is with the understandability of a procedure-based system. Modifying such a system becomes very difficult because a programmer (especially a new programmer who is trying to learn a large system) cannot foresee all the interactions among the system's procedures.

Adding a new knowledge area to a Frame-system, on the other hand, amounts to augmenting a database of interconnected declarations with a new set of assertions constructed in exactly the same way as the old ones. The most important advantage, however, is in the control of processing this kind of database affords. The sharing of features among pieces of a Script guides the understander on how to fill in gaps in the input description. The existence of defaults with their associated, prescribed features tells the process what inferences to make, and when. In our example of a Frame-system for a cube, effects of changes in the orientation of the viewer could be computed cheaply because of the terminals (edges, faces) which are known to be common to the two views; and because of the precompiled information, or defaults, available to help the processor identify new features of a scene.

Charniak [8] has sketched out what a Frame-system for understanding stories about shopping in supermarkets would look like. More recently [9], the same researcher describes how commonsense knowledge about the process of ordinary painting could be represented in a Frame-based format. Neither proposal, however, has yet been reduced to a working program which in some sense actually uses a data base to understand a text about supermarkets or painting a house. SAM, on the other hand, does just these things in its particular realm of newpaper stories. It actually applies Scripts to achieve a reasonable comprehension of what it reads. As a working system, therefore, it embodies a set of mechanisms which solve a problem which has often been discussed but only seldom attacked by an actual program, namely, the problem of making inferences as an aid to comprehension.

The importance of controlling inference in natural-language understanding can be seen by considering what happens in a system, such as Reiger's Conceptual Memory program [22], in which inference goes on "by reflex," without any control at all from higher-order processes. This program was designed to operate on Conceptual Dependency (CD) representations for single English sentences, generating all the inferences (commonsense assertions) that could be reasonably associated with the representation. Rieger's theory recognized sixteen classes of inferences needed to understand text, and the program could handle examples of each kind. Here are samples of some of the more important kinds; the inferences are in upper case:

(I1) Resultative Inferences:
John went to New York.
JOHN IS PROBABLY IN NEW YORK.

(I2) Feature Inferences:
Andy's diapers are wet.
ANDY IS PROBABLY A BABY.

(I3) Function Inferences:
John wants a magazine.
JOHN PROBABLY WANTS TO READ IT.

(I4) Situation Inferences:
John went to a masquerade party.
JOHN PROBABLY WORE A COSTUME.

Rieger's work greatly expanded our understanding of inference. As the heart of the MARGIE system [32], the Conceptual Memory program implemented a kind of "deep" understanding of sentences in a null context, by generating inferences and making memory connections on the basis of those inferences. The problem, of course, is the possibility of an inference explosion. Any Conceptualization can be the potential subject of up to sixteen inferences. The new inferences in turn become the starting points for further inferences. An example discussed in [22] shows how even a simple reference problem can lead to a potentially unstable inferencing process.

Suppose memory knows about two individuals named "Andy," one an adult, the other a baby; and this is all it knows about the world. Given the input "Andy's diapers are wet," the Conceptual Memory program went through two cycles of inferencing, each new set of inferences being handed to Memory's reference processor for a feature check. Although the output given does not specifically say, we may assume that on the order of twenty inferences were generated, of which only a few were actually relevant. In a memory with more facts, an undirected process clearly runs the risk of becoming clogged up by its own inferencing.

SAM controls its inference process by building the most important inferences into the database itself. Its episodes are in fact causal chains [28]: event sequences in which each action has results which in turn become the springboards for further actions. Suppose, for example, a patron has just taken a seat in a restaurant. The episodes about ordering a meal make use of the fact that he is sitting in the dining room to connect the ordering event to the seating event. The waiter may see that the patron is seated. Because the waiter has a definite role in the restaurant Script, this perception must be followed by the waiter's going to the table. Another way the act of coming to the table can be started up is by the patron's calling out to the waiter. Again, the facts that the patron is in the dining room, and that this is where a waiter is expected to be, are combined to initiate the act of taking an order.

Note that we are not saying that all sixteen classes of Riegerian inference may not be needed to understand arbitrary inputs, just that some of them commonly function to interconnect events in a causal chain,

while others hardly occur at all. Consider, for example, the inference needed to understand why the following sentence [22] is peculiar:

John told me he (i. e., John) killed himself.

We know that an actor, to initiate a communication, must actually be alive, and the apparent violation of this simple condition is what makes the example sound strange.

The real challenge is not providing the capability for making inferences like these, but in only making them when needed. In SAM, the commonest inferences, such as (I1)-(I4), are either built into the Script, predict that a particular Script will be referenced in a story, or themselves depend on a Script. Inference (I1), for example, is available from the causal chain. Inference (I2) is handled, in a given Script context, by SAM's reference processor, which "knows" who can be doing what because of the associated Script variable's place in its context. The function inference, (I3), is used by SAM to make predictions about inputs to follow, through the Script(s) which are most strongly associated with an object. For a "magazine," SAM would predict the occurrence of the read-Script, which defines where reading can go on, how to hold a book and turn the pages, etc. Finally, the situational inference, (I4), is nothing less than all the knowledge the Script itself organizes. For masquerades, this includes getting and wearing a costume, perhaps buying a present, etc.

1.5 An Overview of Script Application

This section is an introduction to Script-based story understanding. We discuss the pieces of the SAM system, concentrating on its central module, the Script Applier. The purpose is to orient the reader to the data structures that go into Scripts, and the methods developed to apply them to texts. An example of SAM's processing of a short newspaper article is given in the next section, as an illustration of the depth of comprehension it can achieve.

The version of the Script Applier Mechanism described in this thesis operates as a set of three to six separate but intercommunicating modules, the number varying according to the task. We do not believe that human story understanding can be neatly divided into modules concerned with parsing, inference, generation, etc., functioning independently. On the contrary, the psychological evidence, limited as it is, argues for close integration of processes.

The original reasons for dividing SAM into modules were logistical ones. Versions of the Conceptual Analyzer and Generator were already in existence when the SAM project started. Furthermore, the programs were big enough that running them as a single process soon became impossible. However, once the problem of communicating among the modules was solved, we saw that the division made sense for functional reasons, as well. Each part of SAM is intended to model one kind of process or knowledge source that seems to be needed in story understanding; or, alternatively, to solve one specific problem. We will briefly discuss these problems below.

We said that we wanted SAM to be a "complete" story understander. For any particular story, this means the system runs in three distinct passes, or phases:

1. Understanding Phase: the actual reading of the text and formation of a memory representation for the story.

2. Summary/Paraphrase Phase: production of a summary or paraphrase of the story from the memory representation, and its generation in some natural language.

3. Question-Answering Phase: reading of questions, search of the memory representation and auxiliary inferencing to find an answer, and expression of the result.

During story comprehension (see Figure 1.1), SAM is configured as a set of three modules: one (ELI) for analyzing the text into a meaning representation; one (PP-MEMORY) for tagging and identifying references to Picture Producers (PPs); and one (APPLIER) for applying Scripts.

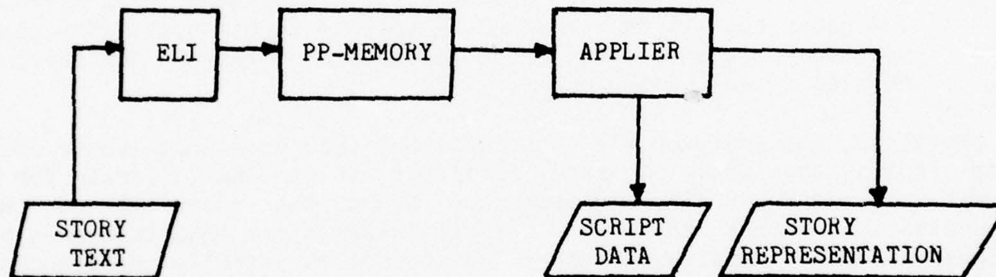


Figure 1.1
SAM: Understanding Phase

Control moves around among the modules of SAM in a co-routine fashion. One program may run for a while, send elsewhere for some information it needs to continue, and eventually regain control. Since SAM is designed to read stories making heavy use of Scripts, the basic job of understanding is performed by the Script Applier. However, Script knowledge can be, and is, exploited by the other modules as well.

Transforming the English text of a story into a CD meaning representation is the job of the Analyzer, ELI (English Language Interpreter). This is an extremely complicated program whose operation is described in [24] and [26]. Script Application, proper, works with the language-free output of ELI, and so, although ELI is an integral part of SAM, we will not discuss it in any detail in this thesis.

A few comments about its function in the story understander may be helpful, however. ELI is the only module of SAM that is concerned with linguistic input as such, that is, with the particular ways that English signals meaning through the choice of word senses, how words are ordered and inflected, etc. ELI's job in SAM is to extract from an English sentence only conceptual elements which are explicitly there, avoiding inference as far as possible. In SAM, we have deliberately reserved the task of filling in things which an input leaves out for the "memory" routines (PP-Memory and the Script Applier), rather than ELI. This is because inferences of this type depend on the use of world knowledge, rather than on the superficial semantic information ELI possesses as part of its knowledge of English.

ELI builds only conceptual entities which are explicitly flagged by the surface string. It constructs its meaning representations by filling slots in conceptual structures derived, usually, from the surface verb. A particular sentence invoking the structure will fill some, but not all, of its slots. As an illustration of conceptual slots which may or may not be filled, consider how ELI would handle the following simple story:

> Story 1.2:
> John took the BMT to Manhattan to see a play. At the theatre, he walked over to the ticket counter and asked for a ticket. The usher took it from him and showed him to his seat. The play was so offensive that John decided to leave. The theatre refused to refund his money.

In Story 1.2, the Analyzer would not make any inference about where John came from in the first sentence. Similarly, in parsing "...asked for a ticket," it would not make any assertion about the recipient of the communication, other than the default assumption that this must be "higher animate." Finally, ELI does not attempt to specify references, pronominal or otherwise. Inferences needed to fill empty slots or to make needed reference specifications depend on detailed world knowledge, and so are more properly performed by the "memory" routines.

ELI maps the surface text into a Conceptualization: a piece of data expressing the "inference-free" meaning of the sentence. From this point on SAM deals directly with the meaning representation. The details of the sentence as it actually appeared in the story are lost. This mode of operation is consistent with our claim that understanding is language-free, in a deep sense. This claim is supported by many psychological experiments on the recall of text (e. g., [3,16,17]) which indicate that the representation of stories in memory is "conceptual," or "propositional." What is remembered are the ideas in a text, not the actual words used. ELI sends its result to PP-Memory, the second module of SAM, which is a memory for Picture Producers (PPs).

Conceptualizations are built out of two ingredients: PPs, and propositions about PPs whose central elements are the primitive ACTs and STATEs of Conceptual Dependency. (See Appendix 1 for a description of CD representation.) PP-Memory's job is to find the PPs in the Conceptualization and assign tokens to them. The tokens are tags or handles by which the PPs will be known to the rest of SAM. This module

also supplies tokens for roles which the Script Applier has encountered in the course of instantiating a Script path, but which were not mentioned in the input. In Story 1.2, the Applier would tell PP-Memory to create a token for the cashier who is implicitly introduced by the Conceptualization for "...asked for a ticket."

A PP in an ELI Conceptualization may refer either to something SAM has seen before, or something "new." Therefore, PP-Memory has to deal with the problem of "reference:" is a new PP in a Conceptualization an instance of one already seen, or a reference to a "permanent" token known to the system, or a pointer to someone, something, someplace, etc., not seen before? The data structures possessed by PP-Memory encode "time-invariant" facts about PPs such as the "conceptual class" they belong to (human, physical object, organization, etc.), what roles they have in different Script contexts, and certain assertions about them which are true in any context. For example, the PP *CHAIR* denotes a "physical object" which people sit on in a variety of contexts, realized as "chair" in a restaurant environment, and as "seat" in the bus or subway environment (since it presumably can't be moved). Attached to *CHAIR* in a real human memory would be additional things such as its "visual image" (perhaps the chair at the person's desk), and such facts as that it usually has legs, a seat and a back. SAM's memory for PPs does not have much in the way of the latter kinds of information, because we frankly don't know how to represent images and quantified assertions too well. What it does have is data about how PPs are used in Script contexts. PP-Memory is a memory for Scriptal roles and props.

At any point in understanding, then, SAM has a list of tokens already identified, and a set of new tokens from the current Conceptualization. Some of the new tokens correspond to pronouns in the surface sentence, and we have the usual problem of pronominal reference. In Story 1.2, for example, there are several references to John using "he." A more difficult reference problem is created by the occurrence of "it" in "the usher took it from him..." Here, recourse to detailed world knowledge about the duties of ushers in $THEATER is needed to enable the correct assignment of "it" to the "ticket" John presumably got from the cashier by asking for one. In each of these cases, the reference problem is solved by the Script Applier.

Another class of tokens refers to well-known people and things in the world. These are called "permanent" tokens. Examples of permanent tokens in Story 1.2 are "the BMT" and "Manhattan." Although permanent tokens are identified as such immediately by PP-Memory, they may have differing Script roles in different stories. "Chairman Mao," for example, might be the head of the state which made the invitation in a "visiting-dignitary" story, or the deceased Very Important Person in a "state-burial" story. As in the case of pronominal reference, therefore, PP-references of this kind can only be settled by detailed examination of context: the PP with its associated Conceptualization, and what has been read before. The reference problem is always solved cooperatively in SAM, by PP-Memory and by the Script Applier, the module which knows about contexts and what can be a role in a context.

When PP-Memory is finished processing the Analyzer's output, it sends the result to the Script Applier. This program has three fundamental problems to solve as it processes a new Conceptualization: (1) locating a new input in its database of Scripts; (2) setting up predictions about likely inputs to follow; and (3) instantiating the appropriate segments of the Script up to the point referred to by the input. Of these three problems, the first one, which we call the Script-management problem, is the most important. To solve this problem, we had to answer questions such as: Which pieces of SAM's episodic knowledge are relevant at any given point in processing? When should a context be removed, and what should take its place? We discuss how SAM manages its Scripts in Chapter 3.

The Script Applier controls the comprehension process by consulting its collection of Scripts. Each Script has several important parts. First, there are the Script's characteristic event-chains, or episodes. Since an event in a Script may be realized in the world in many ways, the events in Script episodes are patterns. These are data structures containing constant parts which are expected to appear exactly in an input; and variable parts which define a range of alternative inputs.

In the theater Script, for example, which (1.2) accesses repeatedly, any member of the public can fill the role of the patron, and the story may give the name of the theater or not. The pattern looking for a patron to enter a theater must be able to recognize the Conceptualizations corresponding to the sentences "John/a man went into a playhouse" and "Mary entered a theater/the Yale Repertory" as instances of the same event from the Script. The Script is also responsible for identifying references in a story to its various roles and props. "The usher," "his seat" and "the theater" are examples from Story 1.2. Sometimes a story will refer to a chain of events comprising a sub-Script from the Script, and the understander must be able to recognize it, as well. In (1.2), for example, we see a reference to "the play," one of the things which happens in a theater.

A special set of patterns are the Script "preconditions," those global facts which SAM assumes to be true when a Script is entered for the first time, unless it reads something to the contrary. When the restaurant-Script is activated, for example, the Script Applier will assert that the person is hungry, and has money to pay for the meal. If the text has indicated that the patron does not have any money (if, for example, he left his wallet home), the violation of the precondition will trigger a prediction that the patron will have trouble when it comes time to pay the bill.

Another important piece of a Script is the definitions of the Script Variables appearing in the patterns. Each definition has a dual purpose. First, it must define the range of features an input PP can have and still satisfy the Script's requirements. The "patron" role in $RESTAURANT, for example, must be capable of accepting either a single person or several persons, since we may hear either of "John" or "the Gavin family" going to a restaurant. Secondly, a Script variable must be capable of giving directions to PP-Memory during instantiation of a role which was not explicitly mentioned in a story. If we hear about a parade during a visiting-dignitary story, the Script Applier must be

able to create a token for an organization whose "occupation" is the Script $LIMOUSINE, to stand for the limousine service which presumably carried the dignitaries around.

The last important part of each Script is the information which is always in active memory. This includes static data such as an initial list of patterns which activate the Script; how related episodes are combined into chunks or "scenes;" time- and place-setting data for the Script; and how other, simpler Scripts may be used as units in it.

The Script Applier's control structure is sketched in Figure 1.2. The three main procedures are a Pattern-Matcher, a Predictor, and an Instantiator. All the procedures run under an Executive, and all have access to Script data. The Pattern-Matcher consists of a routine which sets up desired Script contexts, one at a time, the Matcher proper, and a set of auxiliary inference processes. The Predictor adds and removes event-patterns based on the Pattern currently active and what has gone before.

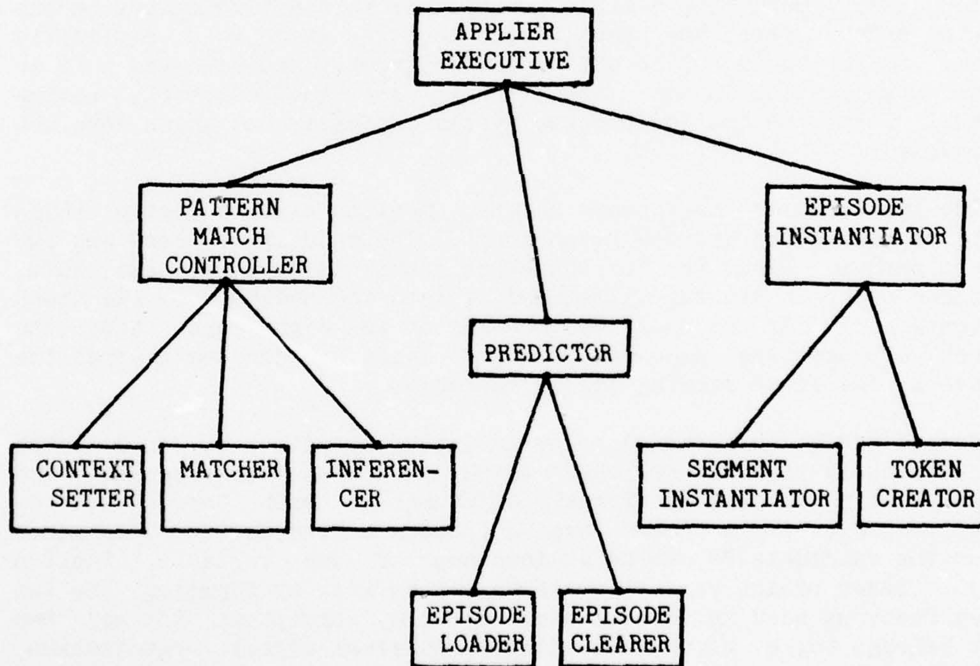


Figure 1.2
Script Applier Control Structure

An "active" Script in SAM defines a context which consists of:

1. A list of patterns which predicts what inputs will be seen at a given point in a story.

2. A binding list which links the tokens for PPs produced by PP-Memory with Script variables.

3. A record of the Script scenes which are currently active.

4. A list of Scriptal interferences -- events which have happened which interfere with the normal flow of activity in the Script -- which are currently outstanding.

5. A Script-global "strength" indicator which SAM uses to flag how strongly it "believes" in its inferences.

The process of Script Application is described in detail in Chapter 4. The Script Applier's basic cycle is to call in these Script contexts one at a time, and to attempt to locate an input in the context invoked (Section 4.4). Candidate Scripts are brought into active memory in the following order: first are those Script contexts which were explicitly referred to by the input or which were indirectly accessed via a PP or sub-Conceptualization in the input; next are the currently active Scripts; last are the Scripts the system possesses but which have not been invoked.

The Applier uses its Pattern Matcher (Section 4.5) to decide which Script is being referenced by an input. The matching process has two distinct phases. First the "backbone" of the pattern, i. e., the ACTs, STATEs and other constants, is matched against the backbone of the input (Section 4.5.1). If the input backbone is of the right type, then the features of the PPs appearing in the input are checked against the features of the corresponding Script variables.

Script variables referred to by an input may either be ones which were previously bound, or ones which have not been accessed. The feature-checking process is slightly different in each case. If the Script variable is a "new" one, a process called Rolefit determines whether the candidate PP can be an instance of the variable (Section 4.5.2). Since Script variables are really defined by function, the two primary features used in Rolefit are: (1) the conceptual "class" the object belongs to, e. g., human, animate, physical object, organization, etc.; and (2) any indicator of the function the PP might have. If the PP is a person, for example, Rolefit would look for an occupation, title or associated Script.

If the variable is an "old" one, there already exists a PP-token which has to be compared to the new one. The comparison is carried out by a procedure called Rolemerge (Section 4.5.3). This looks at the conceptual class and function of the input, as before, then checks secondary features of the input PP and the previous one, for example, residence, age, color, etc., looking for contradictions. The Rolemerge process is SAM's method of doing reference specification.

A form of pattern-directed function invocation [15] is used to check on special features of the input which SAM may be interested in at any given point. Suppose, for example, the system is reading a story about a car accident. At some point in this context, a pattern will become active which is conceptually equivalent to the surface form "someone was hurt." When this pattern is matched, the Script Applier automatically calls a function to check on the value on the HEALTH scale indicated in the input to see whether the actual event referred to was equivalent to "slightly hurt," "seriously injured" or even "dead." The result of the function call would be to modify predictions about future inputs, e. g., how long the subsequent stay in a hospital is likely to be.

Once an input has been located in a Script context, the Instantiator links it up with what has gone before in that context (Section 4.7), and then checks on the effect this may have on other active contexts (Section 4.8). If a Script is being referenced for the first time, the Applier checks on the Script Preconditions to see whether a Script is being entered normally, or whether some unusual events are to be expected in the new context because of a previous event. If more than one context is current, the Applier may be able to update the story representation on the basis of the static information that is always available for the Scripts. For example, the bus Script, $BUS, contains the information that this Script is "sequential" with the train Script $TRAIN. That is, if $BUS is active when $TRAIN is first invoked, the instantiation of $BUS must be completed before $TRAIN is started. On the other hand, $TRAIN contains the information that a reference to the $RESTAURANT context via "dining car" in an existing $TRAIN context defines a "parallel-nested" relationship. Inputs to follow may refer to either Script, but $RESTAURANT should be completed before $TRAIN.

Many transitions between component Scripts are handled by the more complex Scripts which define the "global" context of the story. For example, $BUS, $TRAIN, $PLANE, etc., are known to be "instrumental" means of reaching or leaving the place where the "goal" activity of a trip takes place. The global $TRIP Script may be explicitly introduced, as in "John went to Miami on a business trip;" or implicitly referenced by one of its instruments, as in "John took a train to Miami." Script Situations, as these global Scripts are called, provide the most important machinery for the solution of what we have called the Script-management problem. (Script Situations are described in Chapter 3.)

When the linking process has been completed, SAM updates its predictions about the context based on the new input and what has gone before by merging the specific incremental predictions associated with the pattern that was matched with the Script global search list (Section 4.6). The updated context is then stored, and the next round of processing is started with a call to ELI. After the whole text has been absorbed, the Script Applier constructs a representation of the story that is used by all the postprocessing routines. The representation is a network of causally connected Conceptualizations: both those which were explicitly accessed by an input, and those which could be inferred to have happened. "Header" information is also provided which the

summarizing and question-answering modules use to get at the important events in the network, the global structure of the story in terms of the Scripts which were referred to, and the details of the Script role bindings.

SAM's summary and paraphrase methods are discussed in detail in [36] and [34]. Briefly, these routines access the story representation and pick out the "interesting" events recorded there. Since the CD structures SAM deals with internally are interlingual, output can be generated in any language whose speakers have the requisite world knowledge. As a Machine Translation system, SAM can express summaries or paraphrases in Mandarin Chinese and Spanish, or simulate "simultaneous translation" of an English story into Chinese [39]. The generators used by SAM are modifications of Goldman's BABEL program [14].

The summary/paraphrase task is one of choosing what it seems appropriate to say. Even if we have the means for expressing any conceivable Conceptualization memory may have access to, there is the prior problem of deciding which ones are the best response in a given situation. That is, the summary/paraphrase process has to answer these two questions: (1) which of the Conceptualizations marked by the Script Applier as being important are "interesting" enough to be expressed? and (2) what time- or place-setting information should be included to form the result into a connected whole?

The ability to answer questions about a story that has been read is in many ways the most crucial test of whether the story has really been understood. The theory underlying SAM's question-answering (Q/A) capability is provided by Lehnert's QUALM, discussed in detail in [18]. For orientation purposes, we briefly outline the relevant features of QUALM here.

In the work on SAM, it was desired not only to get an "acceptable" answer to a question, but to get the answer a person would consider "best." To do this, QUALM attempts to mimic the way the person would find that answer. That is, QUALM is designed to answer these two questions: (1) what kinds of search processes on the story representation, and what other kinds of inferences, are needed to find "acceptable" answers to questions? and (2) what methods are there which determine which, out of a number of possible answers, is the most appropriate one?

There is just no limit to the number of reasonable (or unreasonable!) questions that can be asked about a text. The Q/A processor may need to get at every bit of information in the representation to construct an adequate response to a query. Indeed, the answer to a question may refer to an event which did not happen in the story, and active processing may be needed at Q/A time. This is why the Q/A Phase of SAM may involve a computing load comparable to that required for understanding.

In Q/A, an English question to be answered is analyzed into a CD representation by the Analyzer, and tokenized by PP-Memory, as usual. The conceptual question is then passed to the Q/A module, which selects

a search strategy based on the conceptual "type" of the question, examines the story representation for an answer, then sends the answer to the generator to be expressed.

1.6 An Annotated Example

This section presents some excerpts from the processing log SAM kept as it read a three-line story about a state visit of the Premier of Albania to China. Lines shown in upper case are from the program. Each few lines of computer output are accompanied by commentary explaining what's going on.

We've included this example to give the reader some feeling for how SAM goes about its job. However, because SAM is an extremely complicated system, we have not attempted to describe everything it has to do here. For those heroes who really want to know, we have included a more complete, detailed example in Chapter 6. Appendix 2 gives input and output for other stories SAM has read.

Our sample story is understood by SAM as referring to the Script Situation $VIPVISIT, which has component Scripts for travelling ($PLANE and $SHIP), parades ($PARADE), banquets ($BANQUET), etc. First is output for the understanding phase of SAM. Then a summary for the story is expressed in English and Spanish. Finally SAM answers several questions about the story.

The text of the story is as follows:

> Story 1.3:
> Sunday morning Enver Hoxha, the Premier of Albania, and Mrs Hoxha arrived in Peking at the invitation of Communist China. The Albanian party was welcomed at Peking Airport by Foreign Minister Huang. Chairman Hua and Mr Hoxha discussed economic relations between China and Albania for three hours.

Though superficially quite simple, (1.3) contains very real problems of analysis, inference and generation, as will be indicated below. Although the log contains output from every module of SAM, the emphasis will be on the interactions of the "deep-memory" modules, viz., PP-Memory and the Script Applier, rather than on the internal workings of the other modules. The Conceptual Analyzer, in particular, carries a large processing load in SAM. Except for the most complex stories, it uses the most CPU time of all the routines. However, the log will contain only the sentences input to it and the LISP CD representations it computes. (Appendix 1 discusses the particular form of Conceptual Dependency representation used here.) For a detailed description of how the Analyzer operates, see [24] and [26].

The log from which these excepts were taken was made under a DECsystem-10 utility program called OPSER, which has facilities for starting up and controlling several jobs, and sending any output from the jobs to a single terminal. Lines beginning with a "!" are OPSER messages indicating the module from which succeeding lines of output came. PARSER is the Conceptual Analyzer, TOK is PP-Memory, and APPLY is

the Script Applier.

SAM starts up with the Script Applier in control:

```
!(APPLY)
SCRIPT APPLIER MECHANISM...VERSION 4.1...12 JULY 1976
PROCESSING NEWSPAPER TEXT (TEXT . V3)
AVAILABLE SCRIPTS:
($TRAINWRECK $VIPVISIT $VEHACCIDENT)
GETTING NEW INPUT
```

The version of the Script Applier running here is arranged for processing newspaper stories about train wrecks, state visits and motor-vehicle accidents. It contains special procedures for handling lead sentences, making predictions from what it finds in the lead, and taking care of the reference problems that the complicated noun groups found in these stories cause.

PARSER gets control and analyzes the first sentence. The Conceptualization for "arrived" says that a group consisting of Premier and Mrs Hoxha PTRANSed themselves into the city of Peking, and that the arrival happened in some temporal relation to an invitation by Communist China. (PTRANS is the CD action primitive for events which contain a change in physical location. The MODE specifier on the Conceptualization indicates that the PTRANS ceased ("arrived" vs. "went") in Peking.)

Next TOK replaces references to PPs in the Conceptualization with tokens which name the property-list structures that the memory modules use. If the PP is a "permanent token," a well-known person or place in the world, TOK also copies the information from the permanent token onto the new token created for this story. "Enver Hoxha," "Peking" and "Albania" are examples of permanent tokens.

```
!(PARSER)
Sunday morning Enver Hoxha, the Premier of Albania, and Mrs Hoxha
arrived in Peking at the invitation of Communist China.

CONCEPT: GN1
((ACTOR TMP32 <=> (*PTRANS*) OBJECT TMP32
              TO (*INSIDE* PART (#POLITY POLTYPE (*MUNIC*)
                                         POLNAME (PEKING)))
              FROM (NIL) INST (NIL))
MODE (MOD1) TIME (TIM2))

MOD1 = (*TF*)

TIM2 = ((WHEN TMP7)(DAYPART MORNING)(WEEKDAY SUNDAY))

TMP7 = ((<=>($INVITATION INVITER (#POLITY POLTYPE (*NATION*)
                                          POLNAME (COMMUNIST CHINA))
                  INVITEE (NIL)
                  INVITOBJ (NIL))))
```

```
TMP32 = (#GROUP MEMBER (#PERSON GENDER (*FEM*)
                  LASTNAME (HOXHA))
                MEMBER (#PERSON GENDER (*MASC*)
                  FIRSTNAME (ENVER) LASTNAME (HOXHA)
                  TITLE (PREMIER)
                  POLITY (#POLITY POLNAME (ALBANIA) POLTYPE
                                  POLTYPE (*NATION*))
                  REF (DEF)))

!(TOK)
top level PARSER atom is: GN1

processing PP:
(#GROUP MEMBER TMP24 MEMBER TMP28)
creating new token: GROUP0

processing PP:
(#PERSON GENDER TMP25 LASTNAME TMP26)
creating new token: HUM0

processing PP:
(#PERSON GENDER TMP11 FIRSTNAME TMP12 LASTNAME TMP13 TITLE TMP19
         POLITY TMP20 REF TMP23)
creating new token: HUM1

processing PP:
(#POLITY POLNAME TMP21 POLTYPE TMP22)
creating new token: POLIT0

processing PP:
(#POLITY POLTYPE TMP59 POLNAME TMP60)
creating new token: POLIT1

processing PP:
(#POLITY POLTYPE TMP74 POLNAME TMP75)
creating new token: POLIT2

PERMANENT TOKEN IDENTIFIED:
POLIT2 IS !POL101
PERMANENT TOKEN IDENTIFIED:
POLIT1 IS !POL100
PERMANENT TOKEN IDENTIFIED:
POLIT0 IS !POL103
PERMANENT TOKEN IDENTIFIED:
HUM1 IS !HUM100

top level TOK atom for GN1 is MEM0

!(APPLY)
NEW INPUT: MEM0
```

At this point, the Script Applier has received the tokenized Conceptualization underlying the first sentence of the story. It is important to note what PARSER and TOK have done, and what they did not do. PARSER has not attempted any inference about where the Hoxhas came from, or how they got to Peking, although world knowledge suggests that they probably flew from their homeland, Albania, to China. Furthermore, though we know that invitations characteristically *precede the invited* person's actually going somewhere, PARSER has suggested only that there is some temporal relation between the inviting and arriving events. The surface string does not directly state who was invited or what the reason for the invitation was, so PARSER leaves the corresponding slots empty.

TOK has provided tokens for the PPs appearing in the Conceptualization, for example, the Premier and his wife. It has also recognized that the Premier, Peking, Albania and China are permanent tokens. Like PARSER, however, TOK does not have the knowledge required to infer that the group that arrived is the one that was invited, so it leaves this slot alone. Finally, TOK has not been able to suggest a possible Script that the PPs in this Conceptualization may be participating in. This is because people, cities and nations are associated with so many contexts that it is impossible to make a processing suggestion.

APPLY searches first for the imbedded "invitation."

```
!(APPLY)
FINDING IMBEDDED CDS: (MEM6)

SEARCHING FOR MEM6 IN SCRIPT $TRAINWRECK
SEARCHING FOR MEM6 IN SCRIPT $VIPVISIT


LOCATED AT VAR1
BOUND SCRIPT VARIABLE: &INVGSTATE TO POLIT2
TRACK $VIP1 OF $VIPVISIT ACTIVATED

SETTING PARSER WORD-SENSES FOR $VIPVISIT
```

APPLY searches for the input in $TRAINWRECK. However, since invitations don't initiate train crashes but often start up state-visit stories, it finds an appropriate pattern in $VIPVISIT. In this pattern, the entity which made the invitation, China, is identified, and APPLY makes a note of this fact.

Now the system is primed to read more about events from the $VIPVISIT domain. APPLY was able to identify the nature of the invitation from the fact that a nation, China, was doing the inviting, rather than a private citizen. Since SAM now assumes that it knows which context is active, it biases the Analyzer to check first on words and phrases which are appropriate for the state-visit context.

APPLY now looks for the "arrival" the input mentioned:

```
SEARCHING FOR MEMO IN SCRIPT $VIPVISIT

PATTERN BACKBONE MATCHED AT VAR3
VARIABLE BINDING CONTRADICTION IN  (&PTRORG . GROUP0)
TRYING INFERENCE TYPE CONVEY ON VAR3
PATTERN BACKBONE MATCHED ON DERIVED PATTERN:
((ACTOR &INVDGRP <=> (*PTRANS*) OBJECT &INVDGRP TO
        (*INSIDE* PART &INITDEST)))
SUCCESSFUL MATCH ON DERIVED PATTERN


LOCATED AT VAR3
BOUND SCRIPT VARIABLE:
&INITDEST TO POLIT1
&INVDGRP TO GROUP0

GETTING NEW INPUT
```

Now that the first sentence has been read, a number of predictions about what will come next have been made, and some of the Script's variables have been identified. The pattern-match on the "arrival" event contains a typical example of the auxiliary inferencing processes SAM uses to reconcile small differences between an input and a pattern which encodes a specific expectation about what will be read. One of the predictions which is active when the top-level Conceptualization is accessed is for a journey by the invited Very Important Person, specifically, a pattern for an organization such as an airline company or a passenger shipping line moving this person, or a group containing this person, to the country making the invitation. What SAM gets instead is a Conceptualization in which a VIP party moves itself there. In this circumstance, SAM makes a Conveyance inference, and assumes that the VIPs were moved by an organization which wasn't mentioned. If this pattern were instantiated for inclusion in the story representation at this time, APPLY would assume that an airline was, in fact, the organization that did the moving. (Conveyance inferences, and other kinds of auxiliary inferences, are discussed in Chapter 5.)

SAM now starts on the second sentence of (1.3). PARSER interprets this as a "state welcome" since $VIPVISIT is active:

```
!(PARSER)
The Albanian party was welcomed at Peking Airport by Foreign
Minister Huang.

CONCEPT: GN7
((<=> ($VIPWELCOME WELCOMER (#PERSON TITLE (FOREIGN MINISTER)
                                     FIRSTNAME (HUANG))
                   WELCOMEE TMP55)))
 TIME (TIM3) MODE (MOD7))

TMP55 = (#GROUP RESIDENCE (#POLITY POLTYPE (*NATION*)
                                   POLNAME (ALBANIA))
              REF (DEF))
```

```
TIM3 = ((WHEN TMP72))

TMP72 = ((ACTOR TMP55 IS (*LOC* VAL (*PROX*
                                    PART (#LOCALE LOCTYPE (*AIRPORT*)
                                                  LOCNAME (PEKING)))))
```

As in the first sentence, the top-level Conceptualization for "welcomed" is related to another Conceptualization by a temporal link. The modifying Conceptualization states that "the Albanian party" (a group pronoun) was in the proximity of "Peking Airport." After TOK has processed PARSER's output, APPLY looks for the latter event:

```
!(APPLY)

SEARCHING FOR MEM15 IN SCRIPT $VIPVISIT

TRYING INFERENCE TYPE IMRES ON VAR4
PATTERN BACKBONE MATCHED ON DERIVED PATTERN:
((ACTOR &INVDGRP IS (*LOC* VAL (*PROX* PART &PTRTERM))))

SUCCESSFUL MATCH ON DERIVED PATTERN
LOCATED AT VAR4

RUNNING PATTERN FUNCTION (RF1VAR4)


!(TOK)
creating new token: ORGO


!(APPLY)
GOT TOKEN ORGO FOR &PTRORG
BOUND SCRIPT VARIABLE:
&PTRTERM TO LOCO
```

This sequence illustrates a typical interaction between APPLY and TOK in the process of inferencing during pattern-matching. An Immediate-Result inference is calculated because of the mismatch between the pattern looking for the VIP group to come to the arrival point (e. g., an airport), and the stative for "at Peking Airport." Because an airport has been mentioned, APPLY now knows the identity of the transporting organization. It calls TOK to obtain a token of the right type, and binds it to the appropriate Script variable.

Now APPLY searches for the "welcoming" event:

```
SEARCHING FOR MEM11 IN SCRIPT $VIPVISIT
PATTERN BACKBONE MATCHED AT WEL4
CHECKING GROUPS:  (GROUP1 GROUPO)
POSSIBLE REFERENCE FOUND: GROUP1 IS GROUPO
LOCATED AT WEL4

MERGING TOKENS ((GROUP1 . GROUPO))
```

Here we see the first case of a reference that needs to be filled in. When TOK identified the Hoxhas as permanent tokens, it copied the information it had about these individuals onto the tokens created for them in this story. Additionally, the Hoxha family group was marked as having the same residence as its members, Albania. The group for "the Albanian party" is merged with the original group on this basis. Since SAM keeps only one token around for each Script variable, it tells TOK to copy any new information available from the second token it made for the Hoxhas onto the first, and throw the second one away.

This TOK does, and SAM starts on the final sentence of the story:

```
!(PARSER)
Chairman Hua and Mr Hoxha discussed economic relations between
China and Albania for three hours.

CONCEPT: GN13
((ACTOR TMP171 <=> (*MTRANS*)
 MOBJECT (*CONCEPTS* REGARDING
             (#CONTRACT
                 TYPE (*ECONOMY*)
                 PARTY (#GROUP MEMBER (#POLITY POLTYPE (*NATION*)
                                               POLNAME (ALBANIA))
                               MEMBER (#POLITY POLTYPE (*NATION*)
                                               POLNAME (CHINA)))))
  INST ((ACTOR TMP171 <=> (*SPEAK*)))
   FROM (*CP* PART TMP171) TO (*CP* PART (NIL)))
TIME (TIM7) MODE (MOD3))

TMP171 = (#GROUP MEMBER (#PERSON GENDER (*MASC*)
                                 LASTNAME (HOXHA))
                 MEMBER (#PERSON TITLE (CHAIRMAN)
                                 LASTNAME (HUA)))
```

PARSER interprets this event as a dual-MTRANS involving a group consisting of Hoxha and Hua, about an "economic contract" (agreement of some sort) between the countries of Albania and China. After tokenization, APPLY searches for the event in $VIPVISIT:

```
!(APPLY)
SEARCHING FOR MEM20 IN SCRIPT $VIPVISIT
PATTERN BACKBONE MATCHED AT TALK2


LOCATED AT TALK2

BOUND SCRIPT VARIABLE:
&INVITOBJ TO CNTRCTO
&GRP1 TO GROUP2
```

Now all three sentences have been recognized within $VIPVISIT. The final Conceptualization has realized one of the main Conceptualizations (Maincons) of the Script, since it is a possible reason for the state

visit. Other reasons, not instantiated in this story, include the signing of a treaty or the issuance of an official communique. APPLY is prepared at this point to hear about other official ceremonies, or about the VIPs leaving Peking for home.

There are no more story inputs, so APPLY builds a memory representation for the story:

```
BUILDING STORY REPRESENTATION FOR (TEXT . V3)

MAKING STORY SEGMENT FOR SUBSCENE $VARRIVE1 IN $VIPVISIT
MAKING STORY SEGMENT FOR SUBSCENE $VWELCOME1 IN $VIPVISIT
MAKING STORY SEGMENT FOR SUBSCENE $VTALK1 IN $VIPVISIT

RUNNING PATTERN FUNCTION (RFTLK1)
BINDING SCRIPT VARIABLE &MTGPLC TO POLIT1


EVENT GRAPH:
((EVNT1 EVNT2 EVNT3 EVNT4)
 (EVNT5 EVNT6 EVNT7)
 (EVNT8 EVNT9))
```

Story (1.3) has instantiated three episodes from $VIPVISIT: (1) an episode in which a VIP group travels to Peking; (2) an official-greeting episode, acted out, in this case, at Peking Airport; and (3) an instance of the "official talks" episode. APPLY uses the Script-variable bindings it has accumulated during the recognition part of the run to instantiate the events in these episodes. Along the way, it has made inferences about variables which the story did not explicitly mention. For example, the place where Hua and Hoxha met for their talks (the Script variable &MTGPLC) was not explicitly stated, so APPLY assumes it was in the city where the Hoxha party arrived. The result of the instantiation process is the "event graph" of instantiated, interconnected episodes. The event graph, the details of the Script variable bindings, and other information about the story are stored in permanent memory.

At a later time, this information is loaded by the summary postprocessor of SAM, which selects events of interest and passes them to a Generator for expression. In the output shown below, ENGLSH is the English generator, SPANSH is the Spanish generator. (The summarizer SAM uses was programmed by Jerry DeJong. The Spanish output was obtained from a modification of Goldman's BABEL, programmed by Jaime Carbonell, Jr.)

```
!(ENGLSH)

CONSTRUCTING SUMMARY FROM STORY (TEXT . V3)

PREMIER ENVER HOXHA, THE ALBANIA GOVERNMENT HEAD, AND
CHAIRMAN HUA KUO-FENG, THE CHINA GOVERNMENT HEAD, DISCUSSED
ALBANIA COMMUNIST-CHINA ECONOMIC AFFAIRS IN PEKING, CHINA
TWO DAYS AGO.
```

For this simple story, the summarizer has chosen to express only the Script Maincon. The Maincon has been augmented by setting information provided by the Script Applier. For example, APPLY has inferred that the official discussions took place in Peking, China. The information that TOK had as part of its knowledge of the permanent tokens "Enver Hoxha" and "Chairman Hua" has been reflected in the summary by the additional information about their names and occupations that ENGLSH has been instructed to express.

Now the Spanish generator expresses the summarizer output:

```
!(SPANSH)

CONSTRUCTING SUMMARY FROM STORY (TEXT . V3)

EL JEFE DEL GOBIERNO DE CHINA, HUA, Y EL PRIMER MINISTRO
DE ALBANIA, HOXHA, DISCUTIERON UN TRATADO SOBRE ASUNTOS
ECONOMICOS EN PEKING.
```

Finally, SAM answers some questions about the story it has read. In the Q/A configuration of SAM, PARSER, TOK and ENGLSH are the Analyzer, PP-Memory and English Generator, as before. QA is the question-answering module. The details of the Q/A strategies incorporated in SAM can be found in [18].

PARSER analyzes the first question:

```
!(PARSER)
Who went to China?


CONCEPT: GN1002
((ACTOR TMP8 <=> (*PTRANS*) OBJECT TMP8
    TO (*PROX* PART (#POLITY POLTYPE (*NATION*)
                              POLNAME (CHINA)))
    FROM (NIL) INST (NIL))
MODE (MODO) TIME (TIM2))

TMP8 = (*?*)


!(TOK)
top level PARSER atom is: GN1002

processing PP:
(#POLITY POLTYPE TMP39 POLNAME TMP40)
creating new token: POLIT101

top level TOK atom for GN1002 is MEM101
```

PARSER interprets this as a question, marked by (*?*), about the ACTOR of a PTRANS which ended up in the vicinity of China. TOK assigns a token to China in the usual way. Then QA gets control:

```
!(QA)
NEXT QUESTION:
((ACTOR (*?*) <=> (*PTRANS*) OBJECT (*?*)
              TO (*PROX* PART POLIT101)))


(QUESTION TYPE IS CONCCOMP)
(SEARCHING $VIPVISIT-SCRIPT STRUCTURE)
(FOUND AT SCRIPT STRUCTURE LEVEL)

THE ANSWER IS:
(GROUPO)


!(ENGLSH)
PREMIER ENVER HOXHA AND MRS HOXHA
```

QA takes the Conceptual question, identifies it as a query about a role in a Conceptualization, and searches the story representation for an answer. It finds the needed role-filler, GROUPO, and instructs ENGLSH to express this answer.

The answer to the question "Who went to China?" depends, as always, on an inference. First of all, the story does not explicitly say that anyone went to China, only that an official party arrived in Peking. The answer depends on a causal-chain inference, that an arrival in the capital of a country, or anywhere else in the country, for that matter, must be preceded by entering the country the city is part of. The Script Applier, as part of its world knowledge about official visits, built this information into the top level of the story representation. Why the top level? The Script $VIPVISIT, as will be explained in Chapter 3, is a special case of the trip-Script, $TRIP. People, especially Very Important Persons, are always taking trips, using standard means of transportion (that is, Scripts whose Maincons are based on a PTRANS) to get to and from the places they want to visit. $VIPVISIT has the general structure of a trip. Therefore, the Script Applier keeps a record of the "going" part of the trip, which the question-answering module accesses to find the answer to the question "Who went to China?"

```
!(PARSER)
How did they get to China?

CONCEPT: GN1008
((ACTOR TMP103 <=> (*PTRANS*) OBJECT TMP103
              TO (*PROX* PART (#POLITY POLTYPE (*NATION*)
                                       POLNAME (CHINA)))
              FROM (NIL) INST (*?*))
MODE (MOD1) TIME (TIM5))

TMP103 = (#GROUP MEMBER (#PERSON) REF (DEF))
```

```
!(QA)
NEXT QUESTION:
((ACTOR GROUP101 <=> (*PTRANS*) OBJECT GROUP101
TO (*PROX* PART POLIT102) INST (*?*)))

(QUESTION TYPE IS INSTPROC)
(SEARCHING $VIPVISIT-SCRIPT STRUCTURE)

THE ANSWER IS:
((ACTOR ORGO <=> (*PTRANS*) OBJECT GROUPO
             TO (*PROX* PART POLIT2)))

!(ENGLSH)
MRS HOXHA AND PREMIER ENVER HOXHA FLEW TO COMMUNIST-CHINA
```

The answer to the question "How did they get to China?" depends on three crucial inferences that the Script Applier has made. First, "they" must be recognized as the Hoxha party, a reference problem which TOK and QA solve by using the same methods as the Script Applier used to determine the reference for "the Albanian party" in the story input. Secondly, "China" was recognized as "Communist China" by the Script Applier, rather than "Nationalist China," on the basis of the knowledge of where "Peking" is. Finally, the instrumental means that enabled the Hoxhas to get to China was determined by a Role-instantiation inference that the Script Applier made when the phrase "...welcomed at Peking Airport" was read. As in the answer to "Who went to China?", the answer to "How did they get there?" is stored in the story representation as the instrument of the Conceptualization which summarizes the "going" part of the State-Visit Trip $VIPVISIT.

```
!(PARSER)
Why did Enver Hoxha go to China?

CONCEPT: GN1015
((CON (*?*) LEADTO
     ((ACTOR TMP160 <=> (*PTRANS*) OBJECT TMP160
                      TO (*PROX* PART
                                (#POLITY POLTYPE (*NATION*)
                                         POLNAME (CHINA)))
                      FROM (NIL) INST (NIL))
                      MODE (MOD2) TIME (TIM6))) MODE (NIL))

TMP160 = (#PERSON GENDER (*MASC*) FIRSTNAME (ENVER)
                                  LASTNAME (HOXHA))

!(QA)
NEXT QUESTION:
((CON (*?*) LEADTO
        ((ACTOR HUM102 <=> (*PTRANS*) OBJECT HUM102
                      TO (*PROX* PART POLIT103))))))

(QUESTION TYPE IS CAUSANT)
(SEARCHING $VIPVISIT-SCRIPT STRUCTURE)
```

```
THE ANSWER IS:
(BECAUSE ((CON ((ACTOR GROUP2 <=> (*MTRANS*)
                          INST ((ACTOR GROUP2 <=> (*SPEAK*)))
                          MOBJECT (*CONCEPTS* REGARDING CNTRCTO)
                          FROM (*CP* PART GROUP2)) TIME (TIME10))
           IS (*GOAL* PART (GROUP2))) TIME (TIME10)))

!(ENGLSH)
BECAUSE CHAIRMAN HUA KUO-FENG AND MR ENVER HOXHA WANTED TO DISCUSS
CHINA ALBANIA ECONOMIC AFFAIRS.
```

This concludes our sample computer run of SAM on a newspaper story. Chapters to follow will describe in detail the structure of the Scripts, such as $VIPVISIT, which SAM has, and how they are used in story understanding. To finish our discussion here, let's highlight once more some of the important inference processes which actually came into play as Story 1.3 was read.

As the first sentence of Story 1.3 is processed, one procedure, called Rolefit, checks that at least one of the group that arrived is a Very Important Person, that is, that this event fits in $VIPVISIT rather than, say, $TOURIST. An allied process, Rolemerge, is used in the second sentence to decide that "the Albanian party" is the same as the group that was mentioned in the first sentence, namely Premier and Mrs Hoxha. Note, in the second sentence, the special sense of "welcome" that is used. This kind of "welcome," which typically involves bands, speeches, etc., is itself a Script, $VIPWELCOME, imbedded in $VIPVISIT, and this is how SAM, operating in the state-visit context, analyzes it. Similarly, "party" is interpreted as a group-pronoun rather than as a Script.

Other inferences which SAM makes can be seen in the outputs produced for Story 1.3. The need for time- and place-setting information can be seen in the summary, which asserts that the meeting between Hua and Hoxha occurred in Peking on the same day as the arrival, even though the story does not explicitly say this. (SAM inserts the phrase "two days ago" because it is arranged, by convention, to be reading newspaper articles on Tuesday.) The answer to "Who went to China?" depends on a causal-chain inference, that going to Peking must be preceded by entering the country that Peking is part of. Finally, a role-instantiation inference is required in answering the question "How did they get to China?", which assumes that visiting VIPs who are greeted at airports were transported by plane. This is a typical useful, but only "probably" true, natural inference that story understanders make all the time.

1.7 Outline of the Rest of the Thesis

Chapter 2, "Script Structure," presents the details of the data structures which are used in SAM's Scripts. To motivate the discussion, we discuss how a typical Script, the subway Script, is constructed, and what it looks like in the computer. The presentation is an extension of the work of Roger Schank and Robert Abelson [34], which was intended to

model a human memory structure useful, among other things, for story understanding. SAM's Scripts embody this theory in networks of patterns containing roles with requirements on what can fill these roles. THE STORY REPRESENTATION BUILT BY THE SCRIPT APPLIER IS A TRACE THROUGH THE SCRIPT STRUCTURES ACCESSED BY A STORY.

Chapter 3 is a discussion of the various ways in which Scripts can interconnect to account for a story. We consider two different kinds of connections: (1) hierarchical; and (2) temporal/causal. Newspaper stories are understood by means of a generalization of the Script idea, called the Situation. The Situation is a way of classifying how Scripts can be used in a larger context. For example, the Vehicle-Accident Situation prescribes how component Scripts such as "ambulance," "emergency-room" and "police-investigation" will occur: that is, what order these event-chunks will happen in, how the Script roles interface, etc. A Situation is essentially a precompiled set of processing-time suggestions for how Scripts are to be brought into and removed from active memory.

In Chapter 4 we turn from the consideration of the Script as a static data structure, to the discussion of how Scripts are used by the Script Applier in the process of story understanding. We describe the basic Pattern-Match and Instantiate cycle the Script Applier goes through, and the method of "prediction by incremental context" it uses to follow a story through a Script. We then discuss how the nature of the reading task can modify the way in which expectations are aroused and used. For example, reading a newspaper article differs from reading a regular narrative in that the newspaper tells you first what the most important things are. This is what the "lead" sentence is for. Having gotten the lead out of the way, the article then goes over the events in one or more additional passes, adding more details each time.

Chapter 5 describes a set of auxiliary inferences which the Script Applier uses as it tries to recognize a new story input, which are responsible for ironing out discrepancies between an expectation and inputs which, though conceptually "equivalent" to the expectation, vary slightly from an exact match. We contend that low-level inferences of the kinds described are an essential part of any story understander that relies, as SAM does, on expectations about what will be read.

For those interested in the gory details of how SAM goes about its work, we include in Chapter 6 an annotated log of a run of SAM on a car-accident story that actually appeared (in a slightly more complex form) in the New Haven Register.

Chapter 7 presents a keystone of the representational system that underlies SAM's ability to understand: the representation of Picture Producers. The system for representing PPs, like the Conceptual Dependency system of which it is an extension, posits a small number of "primitive" classes of PPs, into which the people, places and things appearing in stories are mapped. The effect of this system on reference specification in SAM is also discussed.

Finally, Chapter 8 makes some general comments on the work and how it fits into the scheme of things in AI. A concluding section is also given on how SAM could, and should, be incorporated into a more general understander having knowledge of Plans and Goals as well as Scripts; and how SAM could be used as a laboratory for memory organization and inference techniques.

Chapter 2
Script Structure

> All the world's a stage, and all the men and women merely players. They have their exits and their entrances; and one man in his time plays many parts...
>
> Shakespeare, As You Like It, II, 7.

2.1 Introduction

To implement a knowledge-based story understander on the computer, we need good answers to two questions: what is the nature and structure of the program's knowledge? and how does the program get at the information it needs when it needs it? In SAM, these questions are addressed, respectively, by Scripts, and the Script Applier.

In this chapter, we discuss how Scripts, considered as data, are encoded for use by SAM. The process of Script Applying is described in detail in Chapter 4.

SAM's Scripts come in several varieties. They range in complexity from the relatively simple things that a person does when serving in a Script role, such as a waiter or bus driver, to the interlinked activities of police, ambulance service, hospital, etc., after a car accident. A classification of Scripts is given in Chapter 3, based on how complicated their structure is, how rigid the associated event-chains are, and what their characteristic actors and objects are. We present here a set of techniques for building Scripts of all kinds. These techniques have been tested by SAM in the process of reading stories from a number of different domains, both simple made-up stories and actual newspaper articles. The Scripts that SAM has applied at one time or another include:

$VEHACCIDENT	motor-vehicle accidents
$VIPVISIT	state visits
$TRAINWRECK	train crashes and derailments
$PLANECRASH	plane crashes
$EARTHQUAKE	earthquakes
$OILSPILL	oil spills at sea
$TRIP	generalized trips
$KIDNAPRANSOM	kidnappings
$RESTAURANT	eating in a restaurant
$BUS	bus rides
$SUBWAY	subway rides
$TRAIN	passenger/cargo train rides
$PLANE	passenger/cargo plane rides
$MARRIAGECEREMONY	marriage ceremony
$OBITUARY	obituaries
$MARRIAGEANNOUNCE	marriage announcements
$PREPARE	preparing food
$DRIVE	driving a car

In the next section, we'll pick a typical Script (the subway Script) from this collection and consider what it looks like in the computer. Before we start on the details, however, some general comments about Scripts should be made. First of all, we need to be clear about what a Script models and what it does not. A Script is intended to describe events which "belong" to a situation in a stereotyped way. For subways, we are interested in what people typically do when they ride on subways. The Script does not care that subway cars in New York have graffiti painted on them, or that the tunnels people walk in are made out of yellow brick and have advertisements posted on them. Static facts of this kind, while true, are simply not relevant to the Script, because they normally have no effect on events. The events in a story that to refer to a Script are part of the normal course of affairs in the situation. They must not be idiosyncratic, but must occur "all the time." In a story such as:

> John went to the cashier's window in the subway station. He offered the cashier a credit card.

we would not consider that the second sentence would fit the Script, because, though credit cards can be used in place of money in most situations, cash is all a subway cashier will take. It would be impossible to build in special paths for novel circumstances like this. There are too many of them. The best a Script Applier could do would be to pass the problem on to a higher-level processor, capable of inferencing in novel situations, with a note that a recognized Script activity appears to have been instantiated, but that an expected feature of one of the props (the fare) has been contradicted.

On the other hand, consider:

> John got on the subway. All the seats were taken, so he had to hold on to a strap.

Here the second sentence is clearly in the Script, although it would sound funny in isolation. Our world knowledge about subways, incorporated in the subway Script, tells us that people become "straphangers" when a seat isn't available.

A second comment has to do with point of view. Most of us know about subways because we have used them to get somewhere. That is, we executed the Script from the standpoint of a subway rider, rather than, say, the conductor or trainman. The subway Script as embodied in SAM takes the point of view of an outside observer, using general knowledge of the situation, such as a rider of subways possesses, to understand what's going on in a story. In most cases, the Script has a designated "main character," for example, the person riding the subway, and this is the role whose actions the observer is watching. We incorporated this "third-person" point of view in Scripts because this is the typical story-telling mode adopted in simple texts and newspaper stories.

Finally, there is the question of the level of detail which is appropriate for Scripts. The subway Script is a device for helping a computer to comprehend ordinary, boring stories about subway rides, not a set of procedures that a robot could actually use to take a ride on

the subway. Scripts are encoded in Conceptual Dependency (CD), a system designed for describing events as they appear in the sentences of a natural language. Script structure is therefore biassed toward recognizing events in the "chunks" with which people describe them. It does not have the fineness of detail that a robot would need to participate in the events. Such information is not needed for ordinary story understanding. Most people can comprehend a story about an airplane pilot, for example, without knowing how to fly a plane.

2.2 Riding the Subway

Having made these general remarks, let's look at the structure of one of SAM's Scripts, $SUBWAY, in detail.

Subways, as everybody knows, are a form of mass transit found in large cities around the world. There are, for example, subway systems in New York, Boston, San Francisco, Tokyo, Paris and Moscow. A ride on the subway in these places differs in various ways. The Boston subway, until recently, used single cars which look more like a bus or a trolley than a train. Tokyo's subways employ special people who cram commuters into cars at rush hour. San Francisco's BART was designed to run under computer control, with a human operator as a back-up.

How can we handle this diversity? The most important observation is that the subway "ride" itself is very similar in each of these cases. This suggests that a single knowledge structure can be devised which captures the similarities, with various pieces of the structure being "overwritten" depending on which city's subways are being ridden. For example, the definition of $SUBWAY's variables would be slightly different if the story were set in Boston. The capacity of the train would be reduced, and the conductor and trainman roles would be merged. In Tokyo, we would have to create a new role for the train-jammers, and add a special episode to the standard "get on the train" scene. In San Francisco, the conductor and trainman would disappear entirely. Once we have the standard "ride," the modifications needed are of two major kinds: (1) roles and props may be redefined slightly, coalesce or go away altogether; and (2) situation-specific episodes may have to be added or deleted.

What does a trip on the subway look like? Let's consider, for example, an ordinary subway ride in New York. A patron enters the station and goes to a turnstile. Next, the patron puts a token in, passes through, and goes to the appropriate platform. Eventually, the train comes. The patron enters and finds a seat. After a number of stops, the destination is reached, the patron leaves the train and exits from the station.

The stereotyped sequence of events just described is the backbone of the subway Script, $SUBWAY, as understood and used by millions of commuters in New York, and, with minor variations, in other cities as well. In each case, what we have is an organization (the subway company or Authority) providing a certain kind of transportation to a member of the public in return for money (Note 1, overleaf). The details of the Script backbone change from instance to instance. For example, there are several different ways a patron can pay for the ride. In Boston

(and New York's PATH system), the patron puts coins directly into the turnstile. In Paris, magnetized strips are used in place of tokens. A comprehensive subway Script will have to contain paths for handling each of these possibilities. The basic structure has to be modified as we discussed above depending on which city's subways are being used.

Let's sketch out the kinds of things the fundamental subway Script has to contain. First of all, there is a cast of characters ("roles"); the objects they use while going about their business ("props"); and the places ("settings") where the Script's activities happen. The roles, props and settings of a Script taken together make up the Script variables, which are matched up against the real-world people, places and things that a story contains.

Here are the roles of the subway Script:

&PATGRP	a group of subway riders
&CASHIER	the cashier
&CONDUCTOR	the conductor
&DRIVER	the person controlling the train
&SUBORG	the subway organization

Picture Producers (PPs) which fill Script roles must belong to one of the "primitive," higher-animate PP-classes described in Chapter 7. For example, the patron role in $SUBWAY must be filled by a PP of the class "person" or "group," since we want to be able to accept both "John Smith" and "Mr. and Mrs. Smith." The subway company providing the service, for example, "the BMT," must belong to the class "organization."

We said that the settings of a Script are the places where the Script's events happen. Settings belong to the PP-class "locale." In $SUBWAY, the three most important settings are the originating station, the inside of the car the patron selects, and the destination station. A more complete roster of settings is as follows:

&STATION1	originating station
&CONCOURSE1	concourse of originating station
&PLATFORM1	platform of originating station
&INSIDECAR	the car the patron rides on
&STATIONn	an intermediate station
&STATION2	destination station
&CONCOURSE2	concourse of destination station
&PLATFORM2	platform of destination station

A special kind of setting, belonging to the PP-class "link," is the collection of structures which connect settings. For a subway station

----

1. In the classification scheme of Chapter 3, the subway Script is an instance of a commercial Transaction between the public and a special kind of organization, a PTRANS-Organization. PTRANS is the Conceptual Dependency "action primitive" for physical transfer of location. See Appendix 1 for a discussion of CD representation.

in which the concourse is on a different floor from the tracks, these would be:

&STAIRWAY11	stairs or escalator between street and concourse (originating station)
&STAIRWAY12	stairs or escalator between concourse and platform (originating station)
&CARDOOR	door of selected car
&STAIRWAY21	stairs or escalator between street and concourse (destination station)
&STAIRWAY22	stairs or escalator between concourse and platform (destination station)

In a station in which the cashier's booth is actually on the subway platform, &STAIRWAY11 and &STAIRWAY12 would coalesce into &STAIRWAY1.

The props of a Script are associated either with the Script's roles or its settings. Examples of the former are small objects, such as tokens and coins (PP-class "money"), which people handle and carry around. The latter props have the job of "furniture" in a setting. For example, the cashier's booth and the turnstile are furniture in the subway concourse. Seats and bubble gum machines are furniture on a platform. (These PPs have the class "physical object.") A special prop in $SUBWAY is the train itself. This is an example of a "structured" physical object whose parts, the cars, are important locations for Script activity in their own right. The props of $SUBWAY include:

&TOKEN	a token
&FARE	money paid for a token
&TURNSTILE	a turnstile
&PLATSEAT	a seat on the platform
&SUBWAY	the train itself
&SUBWAYCAR	one of the cars
&CARSEAT	a seat on the car
&STRAP	a strap for the patron to grasp
&EXITGATE	the gate leading from the platform at the destination station

The most important parts of $SUBWAY are the events, involving the roles, props and settings, which belong to it. An example is the patron's giving money to the cashier. The events in all of SAM's Scripts are based on a single CD ACT or STATE primitive, with appropriate Script variables filling the slots in the conceptual structure. The event described above would look like:

```
((ACTOR &PATGRP <=> (*ATRANS*)
   OBJECT &FARE TO &CASHIER))
```

(This uses the CD action primitive ATRANS, which signals an abstract transfer of possession or control. The LISP CD format for events used in this thesis is described in Appendix 1.)

Several things need to be emphasized about events in Scripts. First of all, they are <u>language-free</u>. The CD representation of an event provides a canonical form into which SAM's Analyzer maps the many

surface strings which are conceptually "equivalent." We would use the same form whether the sentence were: "John gave 50 cents to the cashier," "the cashier got 50 cents from John," or "50 cents was received from John by the cashier." The use of CD representation thus cuts down tremendously on the size of the Script, since only the conceptual content of sentences need be considered, and on the amount of processing SAM needs to do, since the needed inferences can be tied directly to the conceptual events, rather than having to be duplicated for each "equivalent" surface string. (This argument is presented in detail by Schank in [32].)

Another point about Script events is that they contain both "constant" parts (e. g., ACTOR and *ATRANS* in the example given above) and "variable" parts (e. g., &PATGRP and &FARE). Each event, therefore, is really a <u>pattern</u>, a data structure designed to match an arbitrary range of real-world events. We know, for example, that any member of the public can ride on the subway, and so the corresponding slot in the Script's events cannot be fixed but must accept any person or group that comes along in a story. In the "paying the cashier in the subway" activity, we need a way to specify the things which are always true. For example, this event has a person handing over an amount of money to another person who is an agent of the subway organization. We also have to provide for things which can vary in small details. The fare may be expressed as "fifty cents" or a "half dollar," "John" may pay the cashier, or "John and Mary" may pay.

Activities in Scripts are stereotyped. Events follow one another in one of a small set of recognized ways. On entering the subway, for example, the patron may either proceed directly to the turnstile, or stop to buy a token. A chain of events describing one of these well-understood activities is called an <u>episode</u>. "Buying a token" is an episode consisting of the events: "enter the station," "see the cashier's cage," "go to it," "ask for a token," "be told the fare," and "pay the fare." Note that the Script demands that the fare be paid <u>before</u> the token is handed over. This is how this episode is always structured in the subway Script, although the actions can be reversed in other Scripts, for example, if a person were buying an ice cream cone.

The events of an episode don't merely follow one another in time; they are <u>causally</u> connected, as well. "Causally connected" means that the results of one event set up the enabling conditions for the next event in the sequence to occur. That is, the events comprise a <u>causal chain</u> [28]. In the episode we are considering here, the result of entering the station, viz., being at the station's "concourse," is a necessary condition for the patron's locating the cashier's counter. Going to the counter, in turn, enables asking for a token. The principles of causal-chaining used in Scripts are set out in [34], and are briefly described in Appendix 1.

How long can the causal chains in episodes be? It would be possible to define all the events from entering the subway to leaving the destination station as a single episode, but we would clearly be ignoring important facts about the structure of $SUBWAY. One such fact is that some parts of the subway ride are more important, more central to the situation, than others. Getting a token, for example, is an

important activity in $SUBWAY because without one a patron can't get through the turnstile to get his ride. Taking a seat on the platform, on the other hand, is not so important because it doesn't really have any effect on whether the patron can get on the train. Another fact is that sometimes different ways to do the same thing may be available. Having a token before the ride, asking for one at the counter, or showing the cashier a special pass are all possible ways of procuring a ride. In this case, we have a set of episodes which seem to go together.

The activities of a Script which always have to occur for us to recognize that the Script has in fact been instantiated are called its <u>scenes</u>. The scenes of $SUBWAY are:

$SUBWAYENTER	enter the station and wait at the platform
$SUBWAYRIDE	enter the train and ride to destination
$SUBWAYEXIT	leave the station at destination

"Entering," including buying and using a token, is a scene of the subway Script because it is necessary to procure a ride. "Riding" is a scene because this is the transporting activity that the Script is all about. "Leaving" is a scene because we can't be sure that the ride is over until the patron exits from the station. He may just be transferring between subway lines. Each scene of a Script is defined by a set of episodes which (1) describe the different ways the important activity the scene organizes can happen, and (2) give other, less important, actions, such as sitting down on the platform, which can be interlinked with the main episodes, but which don't contribute directly to their accomplishment. The use of "$" before scene names is meant to suggest that each scene (and each component episode) shares some of the features of the Script it belongs to. In particular, each scene and episode consists of causal chains, with specified Script variables and a main activity each chain accomplishes or contributes to.

A final point about the structural properties of Scripts is that their scenes occur in some characteristic order. As with the events in episodes, the scenes are interconnected both temporally and causally. One temporal-causal sequence defining an independent means of accomplishing a complete Script activity is called a <u>track</u> of the Script. The tracks of the subway Script are its manifestations in the different cities which have subway systems. As we mentioned before, these tracks differ from one another only in how one or more scenes differ in details. There is one basic track, using the scene for "paying"," for example, which is specific to the city which is being considered. Another Script having more clearly distinct tracks is the restaurant Script, $RESTAURANT, which has "fast-food," "cafeteria," "regular sit-down restaurant," and "fancy" tracks.

2.3 Script Variables and Patterns

The units from which Scripts are built are patterns or templates for events, Conceptualizations having slots containing references to Script variables. The Script uses patterns because SAM needs to recognize different real-world events as being examples of the action

the Script is interested in. (An event-Conceptualization from a story which is an instance of the pattern is said to match or instantiate the pattern.) Let's look at the variables and patterns of Scripts more closely.

As with most aspects of Script structure, the nature of Script variables can't really be described in complete isolation from how the Script is used. It's enough at this point, however, to note that, in the process of Script Applying, Picture Producers (PPs) appearing in a text are checked to see whether they can be instances of variables. As discussed in Chapter 7, every PP in SAM must belong to one of a small number of "primitive" classes, each class defined by a roster of conceptual "slots" giving a typical property of the class. Here, for example, is the definition of a typical person as it would exist in property-list format in PP-Memory:

"Dr Marcus Welby, 53, of 45 Orchard St, New York"

```
HUMO:
    CLASS (#PERSON)
    TITLE (DOCTOR)
    OCCUPATION (*MD*)
    PERSNAME (MARCUS)
    SURNAME (WELBY)
    AGE (53)
    GENDER (*MASC*)
    RESIDENCE (LOCO)

LOCO:
    CLASS (#LOCALE)
    LOCTYPE (*ADDRESS*)
    STREETNUMBER (43)
    STREETNAME (ORCHARD STREET)
    POLITY (POLO)

POLO:
    CLASS (#POLITY)
    POLTYPE (*MUNIC*)
    POLNAME (NEW YORK)
```

(We're assuming that "Marcus Welby" is a permanent token known to PP-Memory. This is the basis for the specification of "medical-doctor" under the OCCUPATION property. Similarly, "New York" would be known to be a city.)

Suppose that Marcus Welby is taking a ride on the subway. At some point, the token HUMO might be matched up against the Script variable for the patron role, viz., &PATGRP. This Script variable has the following property-list definition:

```
&PATGRP:
   CLASS (#PERSON #GROUP)
   DUMMY T
   SFUNCTION (*NONE*)
```

This means that the atom &PATGRP is a dummy variable to be bound to PPs belonging either to the PP-class "person" or "group." Welby is a person, so far so good. The property SFUNCTION states that a candidate PP should not have a function which can be interpreted as belonging to $SUBWAY. Indicators of the functions people can have are to be found on the OCCUPATION and FUNCTION properties of the corresponding token. Here, the occupation *MD* is not one which is internal to $SUBWAY, so again the PP checks out. If, however, the PP were "Marcus Welby, the driver," the token would contain a FUNCTION (*DRIVER*) flag, and the possible role of Welby as the "driver" of the subway would prevent the acceptance of the PP as an instance of the patron role.

This simple example of the relation between a Script variable and the real PPs that may instantiate it illustrates two basic facts about roles and props in Scripts. First is the observation that they are "abstract" or "generalized." In our example, it doesn't matter that Welby is a masculine adult, since all kinds of people ride on subways. Nor does it matter (except, perhaps, for inferring what town the subway ride is being taken in) where he lives. This leads to a second fact, namely, that Script variables are really defined by <u>function</u>. Concentrating on what people do and what objects are for (this might be called an "episodic" approach), rather than on the details of their structure as bundles of features representing various abstract classes (a "semantic" approach) makes psychological sense. When asked what a waiter is, for example, people invariably reply on some such functional basis as "a person who takes orders in a restaurant and brings people their food."

This is not to say that finding out whether a given PP can fulfil the function prescribed by a variable is always as simple as checking property lists. Suppose, for example, we read:

John's car swerved off the road and struck an X.

The class of things we would be willing to accept as X's can be characterized by the function-word "obstruction." Some PPs, like bridge abutments and cement walls, are massive enough to serve as obstructions for any kind of motor vehicle. Others can obstruct only certain kinds of vehicles. A motorcycle would be demolished if it hit a guardrail, but a garbage truck would demolish the rail. What this means computationally is that whether a candidate PP can function as the Script requires can be decided in the last resort only by running a function, attached to the pattern, taking into account a number of relevant facts. In the "motor-vehicle accident" Script, $VEHACCIDENT, for example, such a function accepts "absolute" obstructions such as abutments, but compares the mass and speed of the vehicle with the mass and degree of immobility of the candidate obstruction. This process of pattern-directed function invocation [15] is an important mechanism of Script Application, to be discussed in Chapter 4.

Every Script pattern belongs to one of three classes, corresponding to the type of Conceptual Dependency action primitive it contains. First are patterns which use one of the eleven CD ACTs. An example of this kind of pattern is:

"Patron enters station"

```
((ACTOR &PATGRP <=> (*PTRANS*)
                OBJECT &PATGRP
                TO (*INSIDE* PART &STATION1)))
```

Next are patterns based on a CD STATE. For example, the subway Script has a STATive pattern:

"Patron waits on platform for a short period"

```
((ACTOR &PATGRP IS (*LOC* VAL
                       (*TOPOF* PART &PLATFORM1))
                MANNER ((DURATION &DELTIM1)))
```

In the above pattern, the variable &DELTIM1 would be bound to an indicator of the length of the wait, e. g., "a few minutes," "half an hour," etc.

The third kind of pattern is based upon a Script. Ideally, Script patterns should contain only ACTs and STATEs. However, there are many cases when we need a way to refer, not to a unit event, but to a cluster of events which is named by a Script. For example, we might have:

"Trainman drives train"

```
((ACTOR &TRAINMAN <=> ($DRIVE DRIVER &TRAINMAN
                              VEHICLE &SUBWAY)))
```

There are two reasons for this. First of all, there are many low-level Scripts, such as the "drive-subway" Script, which are used instrumentally in larger Scripts. These Scripts contain a level of detail and complication which stories rarely get into. Consider, for example, the process of starting a car. We can specify the necessary actions roughly as follows. First, the driver GRASPs the key, PTRANSes it into the keyhole, and "turns" it. When the engine catches (an MTRANS event for the driver), the driver unGRASPs the key. At the same time, the driver might have to PROPEL the clutch with one foot, the gas pedal with the other. On some cars, all this would have to be preceded by PROPELling the choke. The fact is, however, that a text will rarely say more than "John started the car," so we have not bothered to fill in the details of low-level Scripts like these.

A more important reason for having patterns containing Scripts is that stories often refer to them directly as a means of summarizing or pointing to a particular section of a larger Script. Consider a sentence such as: "John's subway ride took an hour." This sentence refers to the duration of the ride scene of the subway Script, and to the person doing the riding. To recognize it, the Script needs a pattern of the form:

```
((<=> ($SUBWAYRIDE PATRON &PATGRP))
   MANNER ((DURATION &DELTIM2)))
```

As another example, consider the state-visit Script, $VIPVISIT. Suppose

we have a sentence such as "The official party was welcomed at the airport." The word "welcome" points to the entire structure of possible episodes in a "state welcome," including bands, speeches, 21-gun salutes, etc. The pattern for this event uses the name of the scene to pick out the participants:

```
((<=> ($VIPWELCOME WELCOMER &WELCOMER
                   WELCOMEE &VIPGRP)))
```

This pattern would also be instantiated by the Conceptualizations for:

> The Secretary of State greeted Chairman Hua.
> Politboro spokesmen welcomed the visitors.
> The official party was received.

but not by:

> Aunt Fanny welcomed her grandchildren.

which is a welcome of a very different kind.

No matter which class a pattern belongs to, the basic idea is to include only the minimum amount of information needed to uniquely identify the event. Let's consider again, for example, the pattern for:

"Patron enters station"

```
((ACTOR &PATGRP <=> (*PTRANS*)
                OBJECT &PATGRP
                TO (*INSIDE* PART &STATION1)))
```

This pattern would be matched by the Conceptualizations corresponding to inputs such as:

(2.1) John and Mary went into a subway station.
(2.2) John walked into a subway station.
(2.3) John strolled out of a restaurant up the street into a subway station.
(2.4) John went into the BMT.

Example (2.1) would instantiate the pattern because, as we explained above, John and Mary form a personal group undistinguished by function. The Instrumental Conceptualization for "walked" in (2.2) would be ignored by the pattern. It would also accept "sauntered," "rambled," "ran" or even "came in on a skateboard." In (2.3), where John came from is of no interest to $SUBWAY, although, in this case, it would constitute a signal to $RESTAURANT (presumably active at this point) that this Script should be closed before $SUBWAY is opened (Note 2, overleaf). Similarly, the phrase "up the street" would trigger a fleeting reference to the "Walk on city street" Script. Finally, (2.4) would instantiate the pattern because PP-Memory would contain a permanent token for "BMT," which is marked as a subway organization.

2.4 Episodes and Pathvalue

2.4.1 The Structure of Episodes

Patterns for events are connected together into chains, called episodes, which describe one typical activity from a Script. Every episode has a main Conceptualization, or Maincon, which is the goal, or point, of the episode. Here are the episodes (marked with "E") and Maincons (marked with "M") of $SUBWAY:

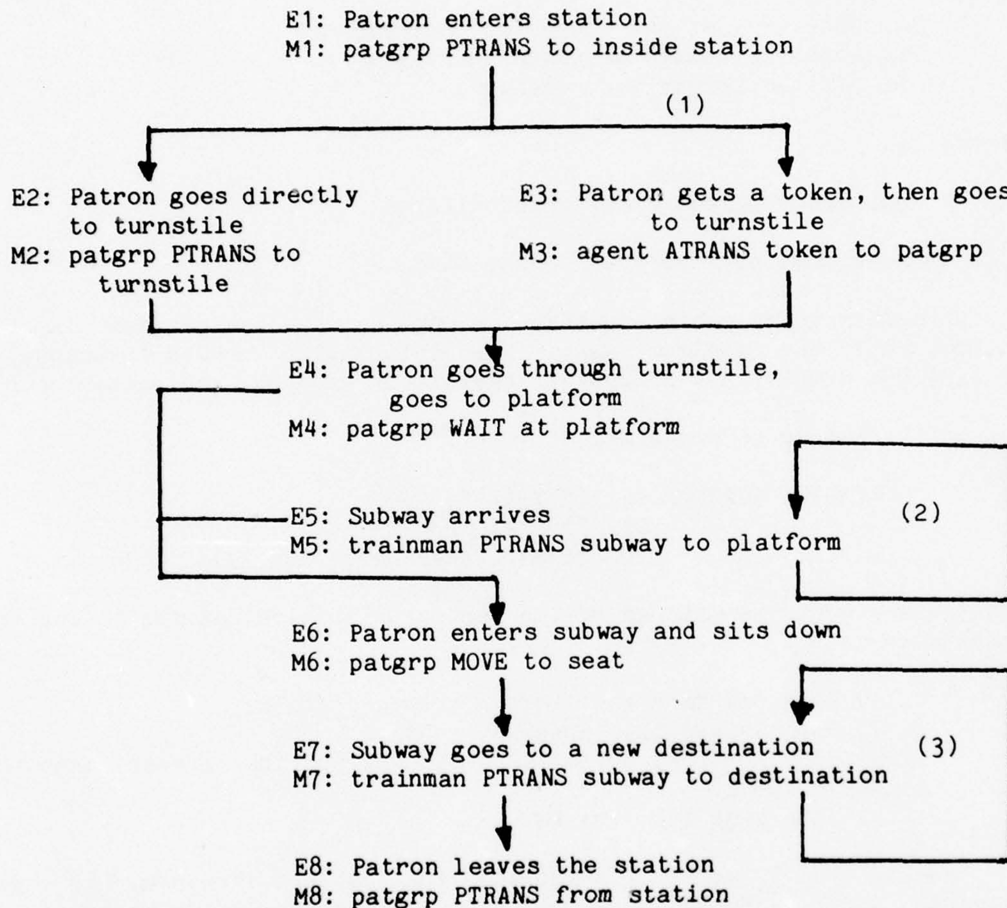


Figure 2.1
Episodes of the Subway Script

----

2. The processing of sentence (2.3) would be handled by reference to the trip-Script, $TRIP, which contains information about the means people use to get to, and return from, places where "goal" activities such as going to restaurants and museums take place. $TRIP is an example of a Script Situation, to be discussed in Chapter 3.

In Figure 2.1, the branching of paths at (1) leads to the subsequent episodes E2 and E3, which describe alternative ways of arranging to get through the turnstile. A branch of this type is called a _turning point_. The "loops" at points (2) and (3) are for the _cyclic_ episodes E5 and E7, that is, for episodes which may happen several times in succession. At (2), for example, several trains may arrive before the one the patron wants, and so the patron must continue to wait. At (3), we have the possibility of the train's stopping at several intermediate stations, before the one the patron wants to get to is reached.

Where does an episode begin and where does it end? An episode begins directly after an event which can be followed by several different event-chains, only _one_ of which can be instantiated by a given story. We call these events turning points because the action of the Script can flow in several possible, mutually exclusive, channels after they occur. As we indicate in Figure 2.1, a turning point in $SUBWAY occurs when the patron enters the station concourse. This can be followed by two alternative episodes. Either the patron will proceed directly to a turnstile, if he has a token, or go to the cashier to buy one. Arriving in a restaurant is an example of a turning point in $RESTAURANT. Possible outcomes of this event include finding and going to a table directly, being taken to a table by the hostess, or having to wait until a table is ready. As a final example, a driver's losing control of his car is a turning point in $VEHACCIDENT, since this may be followed by the car's going off the road, through the guard rail on a bridge, or into the oncoming lane of traffic.

An episode ends when the next possible turning point of the Script occurs, or when a scene boundary is reached. (We discuss the latter case below.) The first and last events in an episode are called its Entrycon and Exitcon, respectively. Turning points are often triggered by MTRANS events, since an act of perception, remembering or communication can lead to a decision by an actor to do one of several things.

Let's look at the episodes from $SUBWAY which describe the patron's entering the station, followed by a forking of events depending on whether he has a token or not. In Figure 2.2(a), we sketch how the events in these episodes fit together. In Figure 2.2(b), we give the definitions of the events in more detail. The episodes shown in Figure 2.2 are labelled $SUBWAYENTER1, $SUBWAYENTER2 and $SUBWAYENTER3, three episodes from the $SUBWAYENTER scene. (The labels on the arrows connecting events in Figure 2.2(a) indicate the causal relation between the events. The types of causal relations used in SAM are briefly discussed in Appendix 1.)

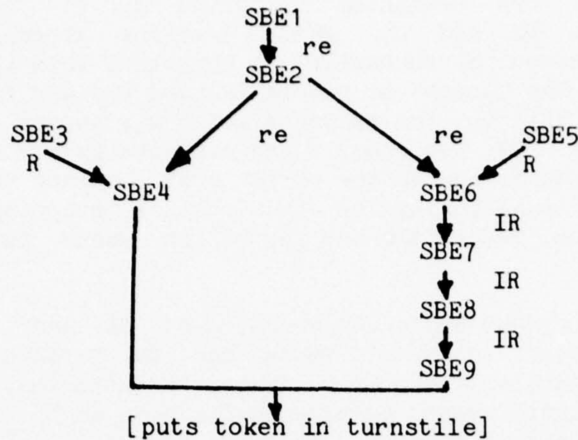


Figure 2.2(a)
Interconnections among Events in $SUBWAYENTER

```
Events of $SUBWAYENTER1:
     SBE1: [Patron uses steps]
           ((ACTOR &PATGRP <=> (*PTRANS*) OBJECT &PATGRP VIA
             (*TOPOF* PART &STAIRWAY11 DIRECTION &DIR11)))
     SBE2: [arrives at concourse]
           ((ACTOR &PATGRP <=> (*PTRANS*)
             OBJECT &PATGRP TO (*INSIDE* PART &CONCOURSE1)))

Events of $SUBWAYENTER2:
     SBE3: [Patron knows he has token]
           ((CON ((ACTOR &TOKEN IS (*POSS* VAL &PATGRP)))
            TOWARD (*MLOC* VAL (*CP* PART &PATGRP))))
     SBE4:  [goes to turnstile]
            ((ACTOR &PATGRP <=> (*PTRANS*)
             OBJECT &PATGRP TO (*PROX* PART &TURNSTILE)))

Events of $SUBWAYENTER3:
     SBE5: [Patron knows he doesn't have token]
           ((CON ((ACTOR &TOKEN IS (*POSS* VAL &PATGRP)) MODE (*NEG*))
             TOWARD (*MLOC* VAL (*CP* PART &PATGRP))))
     SBE6: [goes to cashier]
           ((ACTOR &PATGRP <=> (*PTRANS*)
            OBJECT &PATGRP TO (*PROX* PART &CASHIER)))
     SBE7: [gives cashier the fare]
           ((ACTOR &PATGRP <=> (*ATRANS*) OBJECT &FARE TO &CASHIER))
     SBE8: [Cashier hands over the token]
           ((ACTOR &CASHIER <=> (*ATRANS*) OBJECT &TOKEN TO &PATGRP))
     SBE9: [Patron goes to turnstile]
           ((ACTOR &PATGRP <=> (*PTRANS*)
            OBJECT &PATGRP TO (*PROX* PART &TURNSTILE)))
```

Figure 2.2(b)
Episodes from $SUBWAYENTER Scene

The collection of episodes given in Figure 2.2 is tied together at the event of the patron's arriving at the subway concourse (the pattern labelled SBE2). The trigger for the turning point is the patron's realization that he either does or doesn't possess a token. At this point, the action branches. Eventually, however, the patron ends up at the turnstile, and the branching paths converge. Why is having a token more important at this point in the subway Script than, say, having the Daily Racing Form? The answer is simply that the patron, having executed $SUBWAY many times, knows that without a token he will not be able to get through the turnstile (legally, that is), hence will not be able to get on the subway. The foreknowledge of an obstruction to the Script, based on possession of a token, is what makes a subway rider check his pockets as he enters the station. It's interesting to note that riders often buy tokens in pairs, one for the trip out, one for the return, to avoid having to get a token in both directions. This is because using the subway is one way of doing the trip-Script, which always includes a "going" part, and a "return" part, with the favored means of returning being the same as was used to go.

Patterns from episodes come in three different varieties. First are patterns belonging to an event-chain which describes the main flow of action through a Script. Secondly, there are patterns providing an auxiliary condition which is necessary to keep the main action going. Last, there are patterns for possible, but not necessary, outcomes of main events, which require a confirming episode before they are inferred to have occurred. This distinction among patterns becomes important when the episodes referenced by a story are instantiated for inclusion in the final story representation, since it tells the processor which inferences to make.

A pattern which is part of the "backbone" of an episode, that is, one which is on the direct path of the action described by the episode, is called a "mainpath" pattern. Mainpath patterns are always instantiated, since they provide the necessary connection between events which the story explicitly introduced. Patterns SBE1 and SBE2 in Figure 2.2 are mainpath patterns. Patterns giving auxiliary conditions which are needed for a mainpath event to occur are called "auxiliary-inference" patterns. These patterns are instantiated whenever the mainpath pattern they support is instantiated. Patterns SBE3 and SBE5 are examples of this type. Note that auxiliary-inference patterns may themselves be formed into causal chains of indefinite length. Consider, for example, the episode from the Script describing the patron's entering the subway. The important events in this episode, the main activity it describes, are the patron's waiting at the platform, and getting on the train when it stops. A necessary condition for his entering the subway, however, is that the train be there and the door be open. The chain consisting of the trainman bringing the train into the station, stopping it at the platform, and the conductor's opening the doors is an inference chain that terminates on the patron's entering the car. This configuration of events is sketched in Figure 2.3.

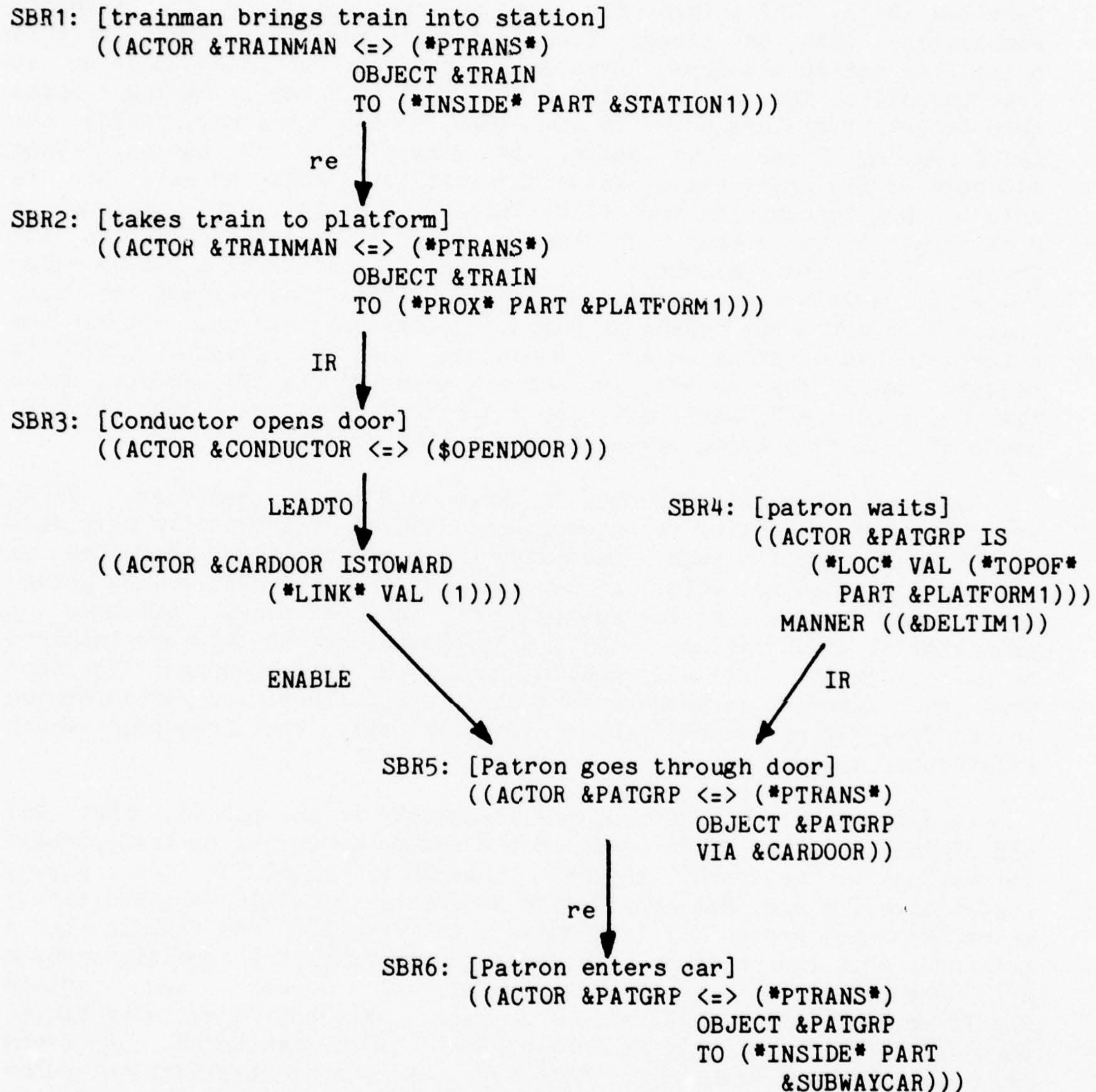


Figure 2.3
Inference Chain in $SUBWAYRIDE Scene

2.4.2 Connecting Episodes Together

Script-based story understanding relies on identifying Script episodes which a story references. Once these episodes have been recognized and instantiated, the understander must make inferences to interconnect the episodes. In our work with SAM, inferences of two radically different types seemed to be required. We call them "short-range" and "long-range" inferences.

Short-range inferences preserve causal continuity between adjacent episodes. Long-range inferences have more of the character of "demons" [7], hanging around in wait for some interesting occurrence. As an example of both kinds of inferences, consider the cluster of events shown in Figure 2.4 which describes a crash in the accident Script, $VEHACCIDENT.

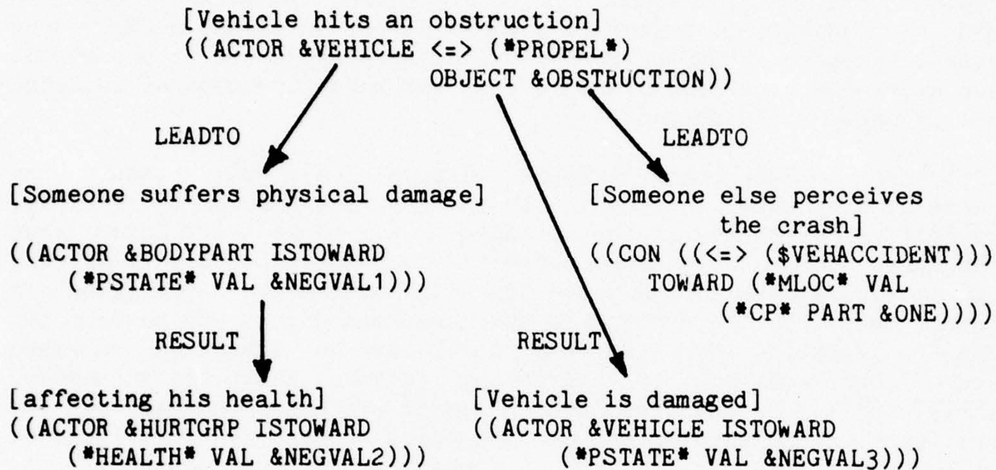


Figure 2.4
Inferences from a Crash in $VEHACCIDENT

One possible result of the crash (the *PROPEL* event) shown in Figure 2.4 is that someone sees or hears it. This perception triggers a call to police, ambulance company, medical examiner, etc., that is, to the episodes which immediately follow a crash. Therefore, this event is a short-range inference, since it connects the crash to the things which happen next.

Another possible outcome sketched in Figure 2.4 is that a bodypart of someone in the car may undergo a negative physical state-change, with a resulting change in the person's health. In the Script, however, nothing can be done about this until a doctor is called and arrives on the scene. Note that the doctor's arrival depends on the short-range inference described above, since he will not be called until someone has reported the crash, and the police and ambulance act. Once the doctor is on the scene, the decrement of the person's *HEALTH* leads to several possible decisions. For example, he may be sent to the hospital by ambulance, first aid may be administered at the scene, or, in the worst case, the body may go to the morgue. This is a typical long-range inference: an event which hovers around and has important, though not, immediate repercussions in the Script. The final result of the crash, the negative change in the physical state of the car, is also a long-range inference, since the amount of the damage won't become important until the owner of the car begins negotiating with his insurance company.

When should inferences like these be instantiated, and when not? In our example of a car crash, the inference about damage to the car should clearly always be made. It is an immediate result of the *PROPEL* event. This type of inference is called an "immediate forward inference," since it is instantiated when its causal predecessor, the crash, is. An analogous inference is the "immediate backward inference," which is made when its causal successor is instantiated. An example of this kind of inference occurs in the restaurant Script, when a Conceptualization about the patron's leaving a large tip is processed. In this case, the backward inference that the patron remembered that the service was good would be made.

Returning to the crash example (Figure 2.4), when should the inference be made about someone in the car's having been hurt? Clearly, when a confirming episode is instantiated. We would not infer that someone was hurt just because there was a crash. However, if we read that an ambulance came to the scene and took someone away, the inference should be made. In the terminology we have been using, the pattern for the Conceptualization "someone was hurt" is an immediate backward inference connected to the confirming episode about the ambulance. Similarly, the inference about someone seeing the crash and calling the authorities is a backward inference from episodes describing the police, ambulance, emergency squad, etc., appearing at the scene of the accident.

2.4.3 Pathvalue

In considering the episodes of a Script, we come up against the fact that some of them seem to occur nearly all the time, while others don't happen so often, and, as a result, are not as expected as the more stereotyped episodes. For example, when reading about eating in a restaurant, we always expect to hear something about ordering, since this is a necessary prologue to eating. Expecting to hear about a big tip, on the other hand, is contingent on fast or polite service, and would not be predicted as strongly as a normal tip, which follows from ordinary service. This observation motivates us to introduce the notion of _pathvalue_, a measure of weight or importance assigned to an episode which indicates roughly how normal or expected it is in the Script.

SAM's Scripts use several pathvalues. The most highly expected happenings are called _default_. These are filled in whether the story mentions them or not. For example, suppose we have:

> John entered a subway station. He waited at the platform for five minutes.

This sequence makes no mention of John's passing through the turnstile and walking to the platform. Since these events would commonly be assumed in this situation, the Script Applier needs a way to fill them in in the story representation. This is done by marking these episodes with the pathvalue "default." The main heuristic followed in assigning the default path in a Script is economy. In reading the above story, for example, the Script Applier would not fill in the possible, but extra, episode in which the patron buys a token. Similarly, in a story

such as:

John went to a restaurant. He ordered lasagna.

the Applier would infer that John found a table by himself and sat down; not that a hostess took him to one, or that he had to wait for a table.

These examples illustrate the basic principle of Script-based story understanding: the filling in of useful -- though possibly mistaken -- connecting inferences, but not too many of them, based on the pathvalues provided by the person who wrote the Script. Inferences are an integral part of understanding. They are needed, for example, to answer questions about things which were left out of a story. If we read:

John went to a restaurant. He had some lasagna.

the answer to the question "What did he order?" depends on an inference about what is usual in restaurants.

On the other hand, any inference may be in error. A subway rider in a hurry may go over the turnstile and run down the steps. Filling in the default episode would then result in the mistaken inference that the turnstile was used in the ordinary way. But it is precisely the unexpectedness of a patron's going under the turnstile that would make us expect that the story-teller would say something about it. For this reason, we would not want our understander to fill in too many things, since the possiblity for a mistake increases. We would not want to assume, in reading the story above, that John had to wait a while for a table, since this event, though possible, seems relatively improbable. The presence of the default path enables SAM to connect up the sentences of a story with plausible episodes. Since the episodes have been "pruned," that is, contain the minimum number of inferences needed to establish a causal connection between adjacent events, the Script provides a built-in control on the amount of inferencing the story understander must perform.

Another pathvalue, called nominal, is assigned to episodes which are typical of the normal passage of events in a Script, but which don't occur often enough to be called "default." Consider, for example, another story about $RESTAURANT:

John went into a restaurant. The hostess seated John and gave him a menu. He asked for some lasagna.

The Conceptualizations for the first and last sentences of this story refer to the default path through the Script. The middle two sentences, however, instantiate "nominal" deviations from the default path.

Sometimes things happen in a Script which depart from the highly expected, normal course of affairs there. This kind of episode is called an interference path. In $SUBWAY, the fact that the patron doesn't have a token initially is a mild interference to executing the Script. In $RESTAURANT, having to wait for a table is a blockage, again, one that's not too serious. Having no money is a more serious interference to both these Scripts. In $SUBWAY, it means that a subway

ride can't be gotten at all (in this case we say the Script has "aborted"). In $RESTAURANT, the patron will be in trouble because he can't pay for the meal he already has eaten. An "abort" of $RESTAURANT can occur if the patron has to wait a long time for service, gets mad, and leaves the restaurant.

Many interference episodes in Scripts are provided with resolution episodes which clear up the blockage, and get the Script back on to its normal track. Patrons in a restaurant, for example, will eventually get a table if they wait long enough. The fabled resolution for having no money in a restaurant -- has it ever really happened? -- is to wash dishes.

We must emphasize once again that not all possible events which can interfere with a Script are actually in it. Only those events which have repeatedly held up the normal flow of action in a situation will be handled by the Script. As an illustration, consider the following story fragment:

John was sitting in a theater. The roof caught fire.

We know that the roof's catching fire will abort the theater Script in a rather dramatic way. Nevertheless, $THEATER could not deal with the second sentence at all, aside from identifying what "the roof" is. A Conceptualization about a fire could only be handled by a larger fire Script, which would know about the causes and results of fires, how organizations such as police and fire department react, etc. (The fire Script is an example of a Script Situation. Situations are discussed in Chapter 3.)

2.4.4 More About Patterns

At this point, we're ready to describe more completely what the patterns in a Script look like. Every pattern is specified by a set of properties. There is a VALUE (the pattern itself); a LABEL which names the episode the pattern belongs to; a TOP which gives the Script; a LASTEVENT pointer to previous events which are connected to it; a NEXTEVENT pointer to subsequent events; a CAUSATION property which defines the respective causal links to the NEXTEVENTs; and pathvalue (PV) and pathtype (PT) properties. Consider the turning point labelled (1) in Figure 2.1, whose component patterns, as shown in Figure 2.2, are SBE2, SBE5 and SBE6. In property-list format, these are:

```
SBE2:   [patron enters subway station]
     VALUE ((ACTOR &PATGRP <=> (*PTRANS*)
                               OBJECT &PATGRP
                               TO (*PROX* PART &CONCOURSE1)))

     LABEL $SUBWAYENTER1       [the episode SBE2 belongs to]
     TOP   $SUBWAY             [the Script the pattern is part of]
     PV DEF                    [the pattern is in a "default" episode]
     PT MAIN                   [the pattern is on a main path through
                                 $SUBWAY]
```

```
     LASTEVENT (SBE1)          [SBE1 is this event's immediate causal
                                  predecessor]
     NEXTEVENT (SBE3 SBE6)     [SBE3 and SBE6 immediately follow]
     CAUSATION (re re)         [the causal connections between SBE1
                                  and its successors]

SBE5:  [patron realizes he doesn't have a token]
     VALUE ((CON ((ACTOR &TOKEN IS
                             (*POSS* VAL &PATGRP))
                             MODE (*NEG*))
                      TOWARD (*MLOC* VAL (*CP* PART &PATGRP))))

     LABEL $SUBWAYENTER3       [SBE5's episode]
     TOP   $SUBWAY             [the Script it belongs to]
     PV INT                    [this event is an interference to
                                 $SUBWAY]
     PT IMMBAKINF              [it is an auxiliary inference of
                                 the backward variety, to be
                                 instantiated whenever its
     NEXTEVENT (SBE6)            causal successor, SBE6, is]
     CAUSATION (IR)

SBE6:  [patron goes to cashier]
     VALUE ((ACTOR &PATGRP <=> (*PTRANS*)
                             OBJECT &PATGRP
                             TO (*PROX* PART &CASHIER)))

     LABEL $SUBWAYENTER1
     TOP   $SUBWAY
     PV NOM                    [going to the cashier is a nominal,
     PT MAIN                      mainpath event in $SUBWAY]
     LASTEVENT (SBE2 SBE5)     [preceded by the mainpath event,
                                 "enters subway" and interfering
                                 event "no token"]
     NEXTEVENT (SBE7)          [succeeded by "ask for token"]
     CAUSATION (IR)
```

The cluster of events defining this turning point contains a default mainpath event (SBE2), a nominal mainpath event (SBE6), and an interference event of the immediate-backward-inference (IMMBAKINF) variety (SBE5). If a story contains a reference to a token-buying transaction, the Script Applier, based on the properties given above, will make the inference that the patron didn't have a token. This interference event is resolved by the event in which the cashier hands the patron a token (SBE7). Pattern SBE7 would have pathvalue RES (for "resolution").

2.5 Scenes and Tracks

Episodes describing the different ways an important Script activity can happen are grouped into <u>scenes</u>. Scenes are chunks of action that follow one another in definite temporal order. In the restaurant Script, for example, there are scenes for "entering the restaurant,"

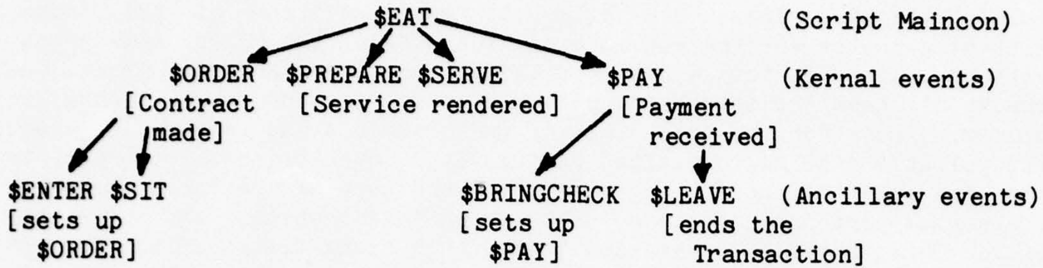
"ordering," "preparing and serving the food," "eating" and "paying/leaving." How do we decide what goes in a scene? We said that a scene comprises all the alternative ways that an "important" activity can be accomplished. But how do we measure "importance"? The division of SAM's Scripts into their scenes is based on three heuristics: one based on clues from the language, one based on unity of setting, and, most important, one based on the structure of the Script itself.

Language itself often suggests the natural order in which Script activities should be arranged. All languages possess special words and phrases which refer, not to discrete actions, but bundles of action. In English, for example, it is possible to talk about the "order" or "service" in a restaurant, a "subway ride," a "drive" in a car, "booking a room" in a hotel, or a "curtain call" at a theatre. When we lay out a Script for SAM, the first thing we do is to accumulate a Script-specific vocabulary, and use it to tentatively block out the scenes.

A second heuristic is just the old Aristotelian notion of the unity of time- and place-setting. Events are likely to belong to the same scene if they go on in the same place at the same time. In a theater, for example, one class of activities goes on in the vestibule, another inside the theater, yet a third in the lounge. In Scripts with vaguely defined settings, the language often provides us with special phrases: for example, "the scene of the crime/accident," "the night of the fire," etc. Note that the place-setting need not have a fixed location. In Scripts whose main event, or Maincon, is a PTRANS of people or goods, the scene shifts to the inside of the conveyance: "on the bus," "in the dining car," etc.

The most important heuristic, however, comes from the type of the Script. Consider, for example, the class of Scripts called Transactions. A Transaction describes an interaction between a member of the public and an organization in which a service is provided in return for payment. (See Chapter 3 for more details.) This kind of Script always has a "kernel" of important events which we may label with the phrases Making the Contract, Providing the Service, and Fulfilling the Contract. Additionally, there may be "ancillary" events associated with getting the actors in a Script into position for one of the kernal events. Figure 2.5 (overleaf) gives the structure of $RESTAURANT and $SUBWAY, two typical Transactions. (The rough temporal order of the Script is given from left to right.)

A scene also contains any well-understood interference events to the scene Maincon, and their resolutions, if any. For example, the ordering scene in $RESTAURANT would contain the episodes shown in Figure 2.6 (overleaf).



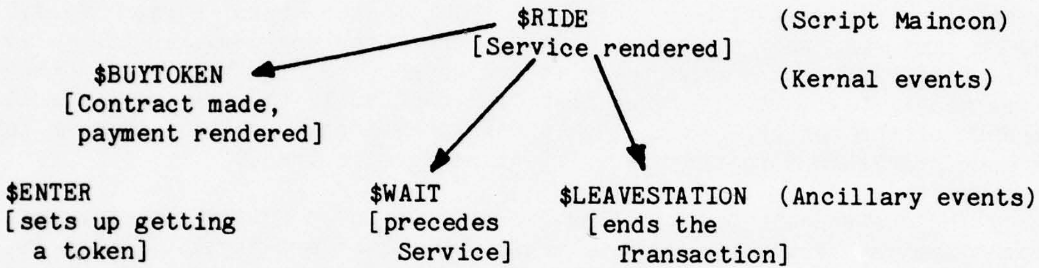


Figure 2.5
The Structure of Transactions

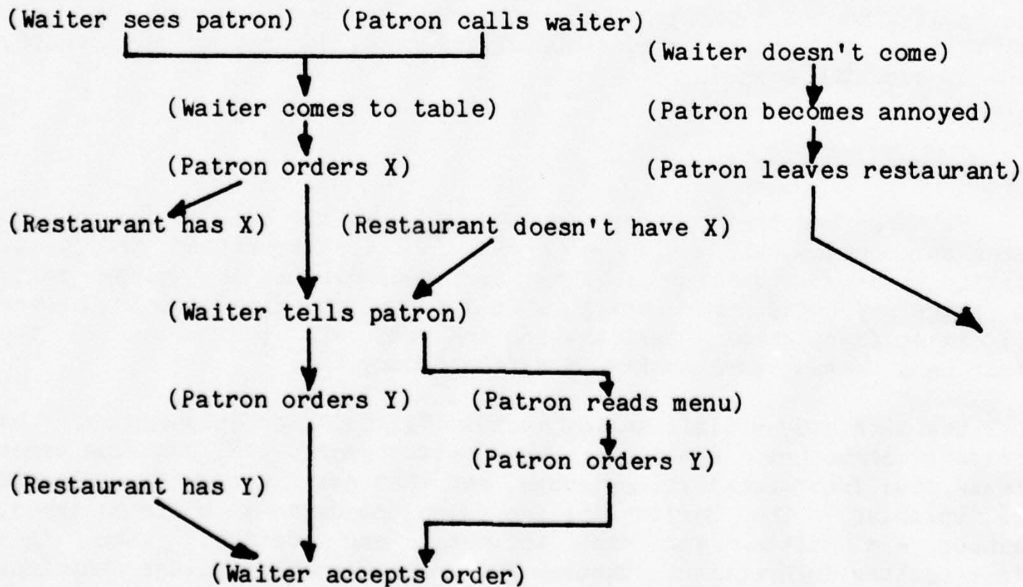


Figure 2.6
Ordering Scene in $RESTAURANT

Script scenes always occur in one or more definite sequences, called tracks. These are alternate manifestations of the Script, differing in the setting and in minor features of the roles and props. Although a story about a Script will pick one track and stay in it, the tracks all have the same Maincon, and share one or more scenes in common. So, for example, eating in McDonald's and Le Pavillon share recognizable seating, ordering, paying, etc., events, but contrast in the fanciness of the setting, the price and type of the food, number of restaurant personnel, sequence of ordering and eating, etc. In the train Script $TRAIN, there are "commuter," "long-distance" and "cargo" tracks.

An important decision that the Script Applier has to make is which one of the tracks to select. Sometimes it can do this on the basis of setting or features of a role filler: "John went into Burger Chef;" "John got on the sleeper car." Other times the decision has to be delayed: "John got on the train in New Haven. He travelled to (New York/Miami)." Sometimes the best it can do is to pick the "default track" of the Script, describing "things in general" in the situation in a manner analogous to the "default" episodes of a track.

The scenes of a track follow one another in a definite sequence. Some scenes, however, contain episodes which are "cyclic," that is, reusable. In $SUBWAY, for example, several trains may arrive and depart while the rider is waiting. Similarly, once the rider gets on a train, arriving at intermediate stops is a cyclic episode of the $SUBWAYRIDE scene. Each arrival event predicts another until the destination is reached. (These episodes were labelled (2) and (3) in Figure 2.1.) Another example of a scene containing cyclic episodes is the arrival of guests at a birthday party. The guests come in their own time, and all go through the routine of greeting the host, depositing their coats, handing over the present, etc.

2.6 Permanent Memory Structures

Patterns and their interconnections make up the bulk of the Script database. Since there is so much of this information, the Script Applier loads only as much of it as seems necessary at any given point in story understanding. (How this is done will be discussed in Chapter 4.) Static facts about a Script which are used all the time, on the other hand, reside permanently in active memory.

One such body of information is the "global" description of the Script's structure, specifying what its tracks are, how each track breaks down into scenes and episodes, and what roles and props appear in the episodes. The Script Applier also needs to know the Entrycon, Maincon and Exitcon for each episode, and default time- and place-setting information. Another permanent memory structure comprises definitions of the Script variables in terms of conceptual class and function markers, and tells how to make a token for a variable which has only been implicitly referenced by a story. All this information is contained in a hierarchical structure which is accessed through the name of the Script.

2.6.1 Setting and Point of View

Let's consider first the static information about the Script's settings. Stories are greatly concerned with making clear where things happen, inserting "in" or "at" phrases when the setting of an event isn't obvious. Sometimes these locales are only referred to in "functional" terms, as in the phrase "the scene of the (crime/accident)." A Script-based story understander, accordingly, will have to have a means for recognizing and using setting information.

Every event in a Script happens in a prescribed setting. Settings are important in Scripts for several reasons. First of all, the existence of a well-defined locale for a bundle of activity is one heuristic for defining the scenes of a Script. For example, consider a building, such as a bank or a subway station, where a Script, or part of a Script, takes place. In a bank, the room with the tellers' counter (a piece of "furniture") is the place where most of the transactions characteristic of the bank Script happen. There is also a room where loan officers do their "thing" (that is, their Script), and a room where the vault and safe deposit boxes are. Each of these places is the setting for a different scene of $BANK. Some of them may physically be part of the same room. For example, the loan officer often sits in the room where the tellers' cages are. However, these two settings are conceptually quite distinct because of the different pieces of the Script associated with them.

In the subway Script, there are three important settings, each associated with a scene of the Script. The $SUBWAYENTER and $SUBWAYEXIT scenes go on in the originating and destination stations; and the $SUBWAYRIDE scene is acted out inside a subway car. The episodes of a scene, in turn, have their own settings. The "acquiring a token" episode, for example, is set in the concourse of the originating station. (This episode was labelled $SUBWAYENTER3 in Figure 2.2.) This information is accessed through the Script name as follows:

```
$SUBWAY:
   PARTS ($SUBWAY1)                [Default track of $SUBWAY]

$SUBWAY1:
   PARTS ($SUBWAYENTER $SUBWAYRIDE $SUBWAYEXIT)
                                   [Scenes of the default track]

$SUBWAYENTER:
   PARTS ($SUBWAYENTER1 $SUBWAYENTER2 $SUBWAYENTER3)
   SETTING (*INSIDE* PART &STATION1)
                     [Episodes and setting of entering scene]

$SUBWAYENTER3:          [Get-token episode]
   ENTRYCON (SBE6)      [First event]
   MAINCON (SBE8)       [Main event]
   EXITCON (SBE9)       [Last event]
   SETTING (*INSIDE* PART &CONCOURSE1)  [Setting]
```

For each episode and scene, the static place-setting information provides a quick answer to the question "Where did this event happen?"

For example, the question "Where did the patron get a token?" would be answered from $SUBWAY, as either "in a subway station" or "in a subway-station concourse."

Another reason for the emphasis on "place" has to do with the roles of a Script. Because roles are so important, SAM keeps track of the places where they have been by updating a Locale-list attached to the token for a person or group after every PTRANS which is instantiated. Additionally, every person has two distinguished locales, "home" and "job," which point to the two most important Scripts the person engages in. A person's possession's, the things that move around with him, are implicitly located by the Locale-list. Here is an example of the information in PP-Memory that would be associated with "Dr Dana Blauchard, of 593 Foxon Rd, New Haven, who works at Milford Hospital" after the sentence "Dr Blauchard took a suitcase to Bridgeport" is processed:

```
HUMO:
     CLASS (#PERSON)
     SURNAME (BLAUCHARD)
     PERSNAME (DANA)
     TITLE (DOCTOR)
     ADDRESS (#LOCALE STREETNAME (FOXON ROAD)
                      STREETNUMBER (593)
                      PARTOF (#POLITY POLTYPE (*MUNIC*)
                                      POLNAME (NEW HAVEN)))
     EMPLOYER (#ORGANIZATION ORGNAME (MILFORD)
                             ORGOCC ($HOSPITAL))
     POSSESSES (#PHYSOBJ TYPE (*SUITCASE*))
     LOCLIST ((TIME1 "BRIDGEPORT")))
```

(Actually, the PPs for "New Haven," etc., would be assigned their own tokens.)

What about organizations? Since organizations don't move, the Script knows about one distinguished place where its "residence" is. If the residence is a building, there are also locales for the various "rooms," with their associated Scripts (or Script scenes).

Another kind of setting information is time-setting. Every track and scene of a Script is assigned a default time-span which defines how long the associated activities should take. This information could be used to answer questions such as "How long does it take to buy a newspaper at a newsstand?" or "How long does it take to get to Miami by train?" in stories which didn't include this information explicitly.

SAM's Scripts use a crude time-scale for the default spans: "short" (order-minutes), "daypart" (order-hours), "day" and "weekpart" (order-days). As an example of how this information is structured, consider the time-span that the "Acquire a token" episode in $SUBWAY would be expected to take. As with the place-setting, this is derived via the Script name:

```
$SUBWAY:
   PARTS ($SUBWAY1)             [Default track of $SUBWAY]
   DEFTIME (*ORDERHOURS*)       [Default time span]

$SUBWAY1:
   PARTS ($SUBWAYENTER $SUBWAYRIDE $SUBWAYEXIT)
                                [Scenes of the default track]
   DEFTIME (*ORDERHOURS*)       [Default time span]

$SUBWAYENTER:
   PARTS ($SUBWAYENTER1 $SUBWAYENTER2 $SUBWAYENTER3)
   DEFTIME (*ORDERMINUTES*)
                    [Episodes and time span of entering scene]

$SUBWAYENTER3:            [Get-token episode]
   ENTRYCON (SBE6)        [First event]
   MAINCON (SBE8)         [Main event]
   EXITCON (SBE9)         [Last event]
   DEFTIME (*ORDERMINUTES*)  [Time span of episode]
```

Another important static datum about a Script is its "point of view." That is, whose Script is it, or, equivalently, whose standpoint is a given story being understood from? Each of the possible participants has a different version of the Script. An attempt to model all these viewpoints runs into several problems, however. From one point of view, the activities of other actors may not be visible. In the subway Script, for example, the trainman who runs the train never really sees the passengers get on and sit down. A patron in a restaurant may never see the person who prepares the food. In both cases, however, these people are aware intellectually of the other actors and what they do, so there is a sense in which the Script is shared. A more serious problem is that certain roles in a Script have specialist knowledge which the others do not share. The pilot of an airplane knows an enormous amount about the mechanics of flying and the details of the route, which his passengers and the rest of the crew don't share.

Every Script in SAM, like $SUBWAY, which describes an interaction between an organization and a member of the public has distinguished "main character," the patron, whose actions the Script concentrates on. (Scripts of this kind are called Transactions.) In the subway Script, for example, this would be:

```
$SUBWAY:
   PARTS ($SUBWAY1)             [Default track of $SUBWAY]
   DEFTIME (*ORDERHOURS*)       [Default time span]
   MCHAR &PATGRP                [Main character]
```

Since stories referencing the Script describe interactions between the main character and the other roles, the point of view the Script takes is that of an outside (but not omniscient!) observer, who knows something about what each of the roles in the Script is expected to do. For example, this observer, using the plane Script, would know that the stewardess is supposed to serve a meal at some point while the plane is

in the air; but not too much about how the food is taken aboard the plane and prepared.

2.6.2 Script Preconditions

Another type of static information contained in Script Transactions comprises the preconditions for the Script. Preconditions are important conditions about the main character (i. e., the patron) which SAM assumes are true when the Script is activated, provided the story hasn't said something to the contrary. For example, if an ambulance is sent somewhere, SAM assumes that it is going to pick up someone who has been injured or is sick.

The most important precondition stored for a Transaction assumes that the main actor is initially in a state which execution of the Script Maincon will "improve." When people enter a drug store, for example, we assume in the absence of other information that they want to possess what the store has to sell. At Script activation time, this object may be bound to a previously mentioned PP, as in "John needed some cough drops;" or only instantiated by "function," here, "a drug." (The inference that $DRUGSTORE will be relevant given the expression of John's want is one that a more general Planning mechanism would have to make, rather than a Script Applier. Such a planner would consult its list of common methods for achieving universal human goals such as preservation of health.) Another important precondition is that the main character has some money to pay for any purchases or services the story may mention.

The existence of a Precondition linked to the Script Maincon in this way provides a means by which a Script Applier, working under a rudimentary Plan-type story understander, could deal with a statement such as:

I went to three drug stores this morning.

This is an example (due to Abelson [1]) of a person's communicating not only the actual events of visiting three drug stores, but also a failure of $DRUGSTORE in at least first two cases.

Let's assume we had previously read "I needed some cough drops." Our Planner would presumably recognize this as a manifestation of the "preserve-health" drive that people share, and trigger predictions about what might be done to get cough drops. A possible plan would be to execute $DRUGSTORE. If the next sentence were:

I went to a drugstore this morning.

the Script Applier would take control. Activation of the Script would cause the Applier to assume that: (1) the patron needed cough drops (the Precondition); and (2) the store was able to provide it (the Maincon); as a result of which (3) the cough drops would now be among the patron's possessions. The "three-drugstores" example, on the other hand, would be analyzed as:

I went to a drugstore this morning
then
I went to another drugstore this morning
then
I went to another drugstore this morning

Processing of the Conceptualization corresponding to the first visit to a drug store would result in the same three assumptions as above. Reading the second sentence would result in reactivation of $DRUGSTORE, but this time processing of the Precondition would be stymied by the fact that the patron already had acquired the needed drug. Although the current version of SAM would boggle at this point -- it has no arrangements for back-up --, one essential ingredient for understanding what must have happened is already available. A vital assumption about what must be true when people go into drug stores has been contradicted.

The best way of viewing Script preconditions is that they are useful "traps" for extra-Scriptal "demons" set up by unusual happenings earlier in a story. In the "three-drugstores" example, the demon is only implicitly present, since we don't know something unusual happened in the first drug store until the second one is mentioned. In more usual circumstances, demons are activated by the explicit negation of the Precondition in the text, or by a Conceptualization which is an inference from such a negation. In entering a birthday-party Script, for example, we usually assume that a partygoer has a present. If some external event, say losing the present or forgetting to buy one, has happened, then this Precondition is directly negated by the former event, negated by inference in the latter case. "Losing" a present has as its immediate result that the present is no longer POSSessed, "forgetting to buy" means that the intention to ATRANS a present to oneself never was carried out, so the POSSession, by inference, never occurred, either.

As an example of this use of preconditions by the Script Applier, consider the following story, part of a longer text which SAM has read (Note 3):

> John got on the subway. On the subway, his pocket was picked. He left the subway and entered a restaurant. He had some lasagna. When the check came, he discovered he couldn't pay.

The pickpocketing event imbedded in $SUBWAY is inferred by the Script Applier (using its pickpocket Script) to have removed money from the main character. This inference hovers around, waiting for the activation of a Script in which the possession of money is assumed. $RESTAURANT is such a Script, and the prediction of trouble -- interference events in the Script -- when the patron goes to pay for the meal is made, through the violation of the Script Precondition, at Script activation time.

----

3. Output produced by SAM for this story is shown in Appendix 2. Details on the actual processing of a Precondition by the Script Applier are given in Chapter 4.

$SUBWAY, then, as a typical Script, has the following preconditions, accessed, as usual, through the Script name:

```
$SUBWAY:
   PARTS ($SUBWAY1)
   DEFTIME (*ORDERHOURS*)
   MCHAR &PATGRP
   PRECONS (SBPC1 SBPC2)

SBPC1:                         [Patron has amount of fare,
   ((ACTOR &FARETOKEN              or a token]
          IS (*POSS* VAL &PATGRP)))

SBPC2:                         [Patron wants to be at destination]
   ((CON ((ACTOR &PATGRP
             IS (*LOC* VAL (*PROX PART &DEST))))
          IS (*GOAL* PART &PATGRP)))
```

The first precondition asserts that the patron will possess either money or a token initially. &FARETOKEN is a special Script variable which is designed to accept either money in the amount of the fare or a token:

```
&FARETOKEN:
   CLASS (#MONEY)
   MONEYTYPE (*CURRENCY* *TOKEN*)
   DUMMY T
```

(Special checks can also be made, via a pattern-invoked function, that the amount of money possessed by the patron is enough to pay for the fare, that his money isn't in the form of a large bill, which cashiers won't change, etc. This process is described in Chapter 4.) The second precondition asserts that the patron wants the result of the Script Maincon, to be at the destination location, to come true.

2.6.3 Script Headers

A final type of permanent memory structure associated with a Script is the set of patterns for events which "invoke" or "initiate" a Script, that is, make available its predictions about what should happen next in the context. (Deciding when a Script has been "invoked," i. e., that additional Conceptualizations will occur which refer to the Script, as opposed to "instantiated," i. e., definitely known to have occurred, is a tricky business. We will return to this in Chapter 4.) These patterns, which are called Script Headers, are the only patterns from the Script that are present in active memory if the Script has not been accessed by the Conceptualizations read so far. It's clear that the entire Script should not be present. This makes neither psychological nor computational sense. The Headers are the tip of the Script iceberg.

What goes into a Header? The basic rule is that a complete event is needed to bring the Script into play. That is, a Conceptualization should be recognized rather than just a PP. For example, $RESTAURANT should not be invoked just because "a restaurant" is mentioned. This is not to say that Script-related information should be completely

suppressed, because it may be useful in later stages of understanding. For example, in "I met a truck driver in a diner," remembering the role the person had in $TRUCK may be crucial to understanding what he might say or do later. (Note, however, that the necessary inferences could not, in most cases, be provided by a Script Applier. For example, understanding a sentence such as "the driver said that he hated Commies and homosexuals" would require knowledge about the personal belief system a truck driver is likely to have, rather than knowledge about what the driver does as part of $TRUCK.)

Conceptualizations are produced, not only by surface clauses, but also by certain kinds of prepositional phrases. Such phrases can act as complete thoughts by modifying the Time- or Place-Setting of the main event. Consider the following sentences:

(2.5) Mary was killed in an accident
(2.6) At the hot dog stand John asked for a Coke.

Example 2.5 can be paraphrased roughly "When there was an accident, Mary was killed." The top-level event is placed into some temporal relation to the Script $ACCIDENT. Similarly, the time of the "asking" Conceptualization in Example 2.6 is modified by the Locale-specification "When someone was at the hot dog stand."

Time/Place modifiers like these, though the sentence may not explicitly say so, are nearly always connected causally to the main events. Making the causal connection is an inference that episodic memory, that is, a Script Applier, has to be prepared to make. Notice that the actual temporal order of the main and modifying events may be ambiguous. In Example 2.5 we know that that the death occurred after the crash. In the following, however, (an example discussed by Riesbeck in [36]):

(2.7) While leaving the bus, Mary thanked the driver.

it seems clear that the physical event of leaving occurs after the thanking. However, there is a sense in which "leaving" in this example refers to the whole complex of activities associated with getting off a bus, or, as we would say, with the "leaving" scene in $BUS. In SAM, phrases such as those occurring in Examples (2.5-2.7) are analyzed into Conceptualizations of the form "When X, Y." If a Script is active, words such as "accident" and "leaving" are analyzed into the associated Scripts: $VEHACCIDENT and $BUSLEAVE, respectively. The pattern for X then calls the appropriate episodes into active memory, and the event Y is found.

Script Headers come in four varieties, which are ranked on the basis of how strongly they predict that the associated context will be instantiated. The first type is called a Precondition Header (PH) because it triggers the Script on the basis of a main Script Precondition being mentioned in the text. (A Precondition is an important global condition which SAM assumes to be true when a Script is activated, unless the text says otherwise.) As an example, the sentence "John was hungry" is a PH for $RESTAURANT because it is an enabling condition for the main Conceptualization (INGEST food) of the Script. A

story understander having access to both Scripts and Plans would make the prediction (a relatively weak one, to be sure) that $RESTAURANT would come up because this is known to be a common means (in the parlance of Schank and Abelson [34], a Named Plan) for getting fed. A related PH would be an actual statement of the goal the Script is normally assumed to achieve, or one from which the goal could easily be inferred. In "John wanted to eat a hamburger" or "John wanted some Italian food," the inference chain to the Script Precondition is relatively straightforward. Interestingly, world knowledge about the existence of restaurants specializing in Italian food would make the PH prediction about this version of $RESTAURANT more forceful than usual. Another kind of PH occurs in Story 1.3 (Chapter 1, p. 19). The phrase "at the invitation of Communist China" triggers the prediction that the Premier of Albania will indeed want to instantiate the state-visit Script, since invitations are often accepted. Patterns for PHs are explicitly stored in the Script, since SAM does not have the ability to use Plans.

We said that PHs are only weakly predictive of their Scripts. There are two reasons why this is so. The first, of course, is that many Scripts have similar PHs. If we read "John was hungry" in a null context, we can't be sure that he intends to instantiate $RESTAURANT. If he's at home, he may just go to his refrigerator to get something to eat, or go to a supermarket to buy some food. The second reason is that, even if the predicted Script does occur, there may be a substantial delay between the announcement of the PH and the instantiation. This is especially true of socially ritualized situations such as parties, marriages, and vacations which require a lot of preparation. If we have "John and Mary wanted to get married," it is possible that $MARRIAGECEREMONY will occur immediately. Perhaps they'll go to the courthouse or to a marriage parlor (the $CIVILCEREMONY and $QUICKIECEREMONY tracks of the Script, respectively). But the church-ceremony version of the Script is the one most often used, and this entails elaborate preparations which take some time. All the things which happen, arranging for the ceremony and reception, getting an organist and photographer, issuing invitations, etc., do point toward the culminating Script, however, so it must be available.

A second type of Header making stronger predictions than a PH about the associated context is called Instrumental (IH). An IH commonly comes up in inputs which refer to two or more Scripts, of which one at least can be interpreted as an "instrument" for the others. For example, in "John took the subway to the restaurant," both $SUBWAY and $RESTAURANT would be predicted, since subsequent inputs about either make perfectly good sense. Here, the reference to $RESTAURANT is anticipatory, and $SUBWAY is a recognized instrumental means of reaching locales in which more important Script goals can be expected to be accomplished. The ways in which Scripts are associated with one another in Instrumental patterns in Script Situations will be discussed in Chapter 3.

The notion of a time-place Setting for a Script leads to the third and most strongly predictive type of Header, the Locale Header (LH). We know that many organizations have a "residence" or "place of business" in which they characteristically carry on their activities. They may

have distinctively designed ornaments or buildings (e.g, a pawn shop's sign, a barber's pole, or McDonald's Golden Arches) which signal their Script to the world. When an understander reads that an actor is in the proximity of the residence, or, better yet, inside the residence, its expectations about the occurrence of the Script are correspondingly reinforced. Examples of LH's are "John went to the soccer field" and "John went into the Museum of Modern Art."

The final type of Header is a flat assertion that the Script occurred. Examples include:

> There was a car accident.
> An earthquake struck.
> John went on vacation.
> Mary went sailing.

Such Direct Headers (DHs) are the top-level patterns in Scripts. DHs are always the first patterns to be checked in a context, since they have the maximum predictive power. They are matched against even in an active Script, since sentences (especially from newspaper stories) may use them to refer to a role or other attribute of the Script. Consider, for example, phrases such as "a two-car crash," "a violent hurricane," "a three-day state visit." A special case of the DH occurs for a Script which can be initiated by "remote control," that is, by a letter or over a phone. If we read, for example: "John called the police" or "John ordered a lawnmover from the Sear's catalog", we infer, if nothing else is said, that the organizations in question will execute the appropriate track of their Script.

Schank and Abelson [34] have introduced another kind of Header, the Internal Conceptualization Header. In the example:

> John went to visit his friend Mary who was a waitress. While he was waiting for her, he ordered a hamburger.

the fact that a role name in $RESTAURANT has been mentioned is given as a reason for predicting that the Script will be invoked. This is a very subtle question, but one can argue that the prediction is made at least as much on the basis of "where she is" as "who she is," that is, the ICH is really an LH.

People characteristically execute Scripts in two important settings, one corresponding to their role in a family, and one for their job. Note in the above that the use of the word "visit" suggests that the person being visited is in the usual place. An additional specifier is needed if this is not the case, as in "I visited Mary in the hospital." Because of these default settings, it is perfectly possible to replace the second sentence with "While he was waiting for her, her mother served him a hamburger." Here, the $RESTAURANT place-setting has been overruled by the $HOME setting because the role "mother" has only one attached Script locale. Even in these liberated times, home and job are the same for many mothers. Time-Setting can have a similar effect: "John went to visit his friend the mailman one evening." Since we know that mailmen work only during the day, it seems clear that the home setting is the one being visited.

SAM does not use ICHs, as such. The mention of a role in a Conceptualization results in a processing suggestion that the associated context be tried. The actual initiation of the Script depends on the Conceptualization's matching either a PH, an IH, or an LH. So, in the sentence "A mailman went into a restaurant," both $RESTAURANT and $DELIVERMAIL would be activated via LH. These would be held in "abeyance" (more about this in Chapter 4) until a further input clarified the issue. If the sentence were instead "A mailman went into a restaurant for lunch," the fact that a PH and an LH for $RESTAURANT have been matched would cause $RESTAURANT to be invoked. $DELIVERMAIL would not be seen.

2.6.3.1 Example Headers for $SUBWAY

Here are the Headers of the subway Script:

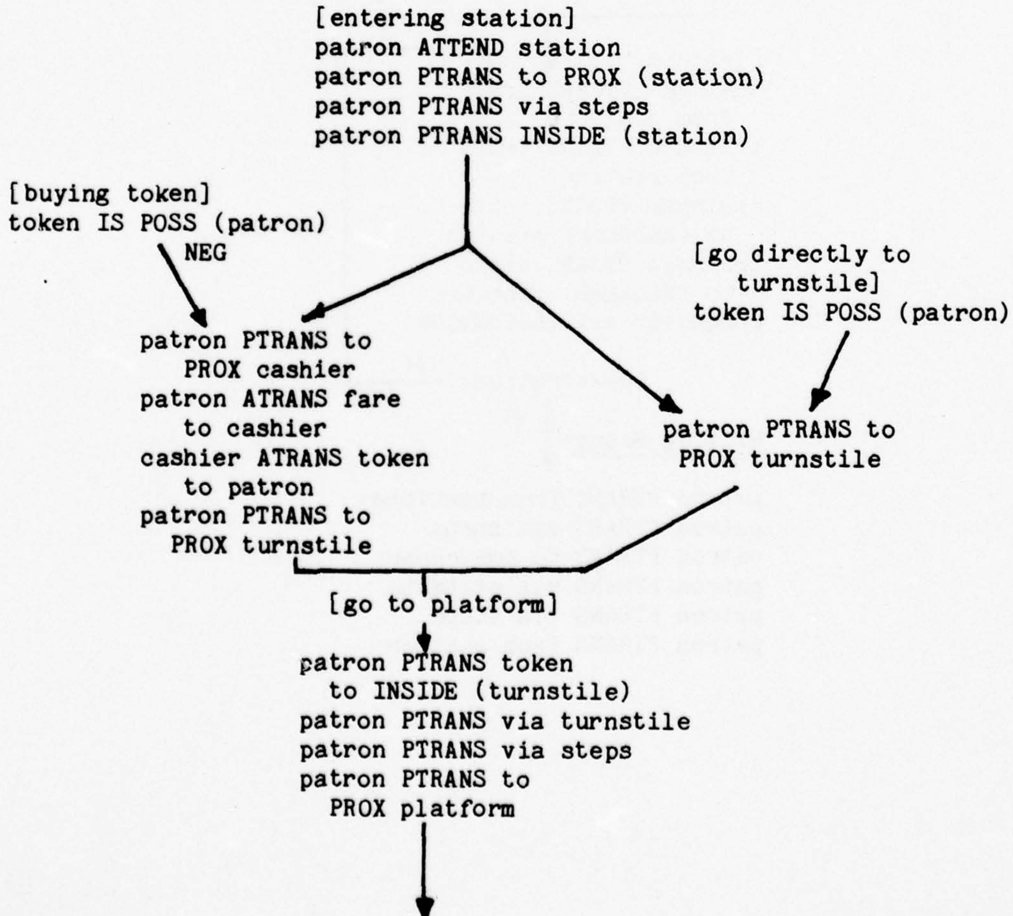
```
$SUBWAY:
   PARTS ($SUBWAY1)
   DEFTIME (*ORDERHOURS*)
   MCHAR &PATGRP
   PRECONS (SBPC1 SBPC2)
   INITQ (SBIN1 SBIN2 SBIN3 SBIN4)

SBIN1:                      [Direct Header]
   ((<=> ($SUBWAY MAIN &PATGRP PTRORG &SUBORG
                ORIG &ORIG DEST &DEST)))

SBIN2:                      [Locale Header]
   ((ACTOR &PATGRP <=> (*PTRANS*)
                 OBJECT &PATGRP
                 TO (*INSIDE* PART &STATION1)))

SBIN3:                      [Instrumental Header]
   ((ACTOR &SUBORG <=> (*PTRANS*)
                 OBJECT &PATGRP
                 TO (*PROX* PART &DEST)))

SBIN4:                      [Precondition Header]
   ((CON ((ACTOR &PATGRP <=> (*PTRANS*)
                 OBJECT &PATGRP
                 TO (*PROX* PART &DEST)))
         IS (*GOAL* PART &PATGRP)))
```

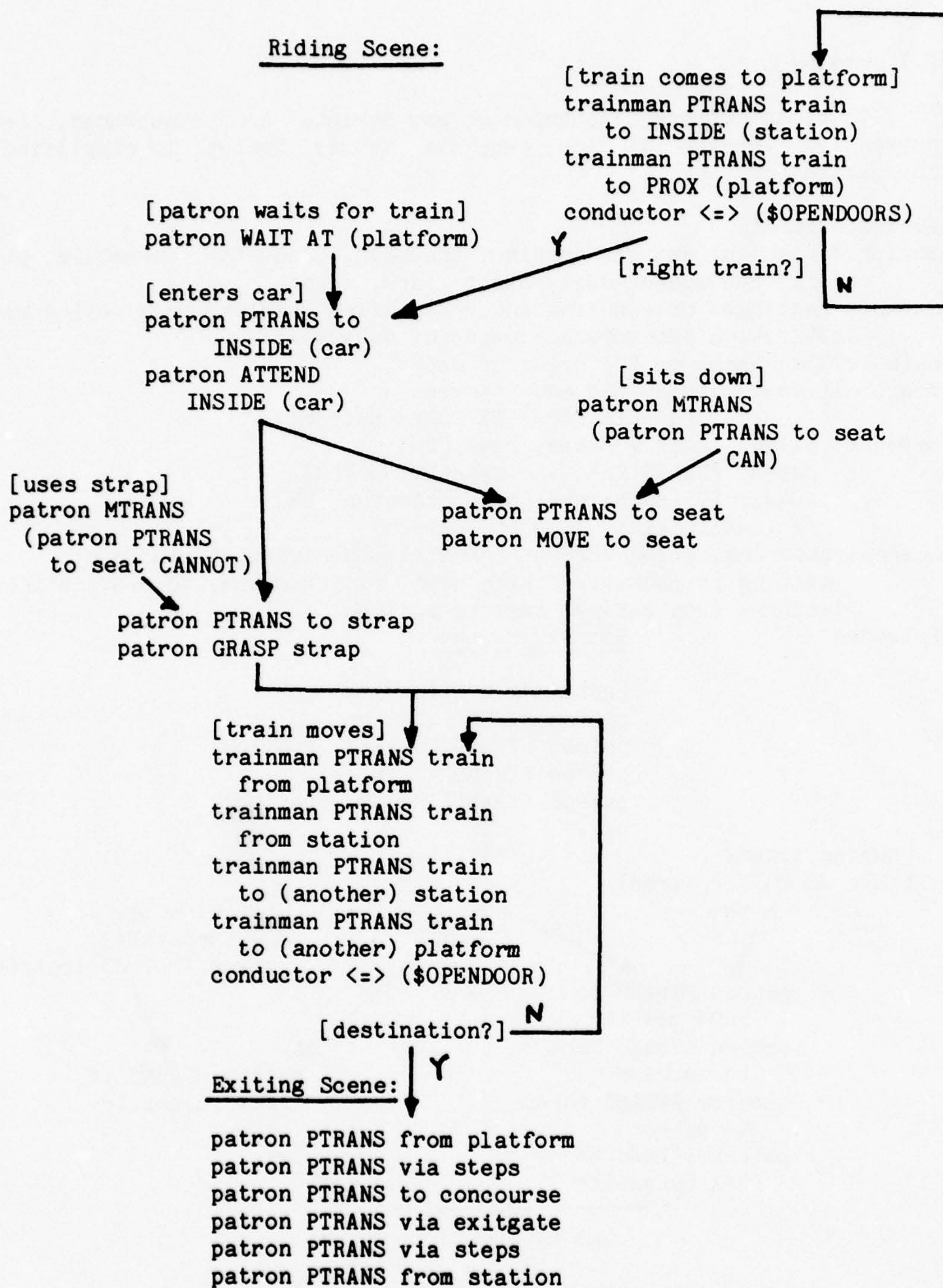
The DH is intended to handle Conceptualizations corresponding to inputs such as "John took a subway ride to Coney Island." The LH takes care of sentences such as "John walked into the Boro Hall subway station." The IH will handle Conceptualizations such as "The IRT took John to Shea Stadium." Finally, the PH would match Conceptualizations for sentences such as "John wanted to go downtown".

2.7 Summing Up

To close out our discussion on how Scripts are structured, let's present a version of the complete subway Script in simplified CD representation:

Script: $SUBWAY
Script Variables: patron, cashier, trainman, conductor, turnstile, platform, train, subwaycar, seat, strap, fare, token.
Default Settings: originating and destination station, originating and destination concourses, inside of subwaycar.
Default Time-Span: on the order of hours
Preconditions: patron POSS money/token
AT (destination) IS GOAL (patron)
Headers: patron takes a subway ride [DH]
patron PTRANS into subway station [LH]
subway PTRANS patron to destination [PH]
AT (destination) IS GOAL (patron)
Scenes: entering, going through turnstile, arriving at platform; getting on subway, sitting down, riding subway to destination; exiting from subway, leaving station.
Episodes: Entering Scene:




Riding Scene:
[train comes to platform]
trainman PTRANS train
to INSIDE (station)
trainman PTRANS train
to PROX (platform)
conductor <=> ($OPENDOORS)
[right train?]
Y
N
[patron waits for train]
patron WAIT AT (platform)
[enters car]
patron PTRANS to
INSIDE (car)
patron ATTEND
INSIDE (car)
[sits down]
patron MTRANS
(patron PTRANS to seat
CAN)
[uses strap]
patron MTRANS
(patron PTRANS
to seat CANNOT)
patron PTRANS to seat
patron MOVE to seat
patron PTRANS to strap
patron GRASP strap
[train moves]
trainman PTRANS train
from platform
trainman PTRANS train
from station
trainman PTRANS train
to (another) station
trainman PTRANS train
to (another) platform
conductor <=> ($OPENDOOR)
[destination?]
N
Y
Exiting Scene:
patron PTRANS from platform
patron PTRANS via steps
patron PTRANS to concourse
patron PTRANS via exitgate
patron PTRANS via steps
patron PTRANS from station

Chapter 3
Managing Scripts

3.1 Why This is a Problem

How do you know when you're in a Script? Suppose things have been going smoothly along inside one Script. What tells you when to terminate it and start up a new one? Is it possible to "suspend" a Script, bring in a new one to take care of things for a while, then restart the old one later? All these questions, considered from the standpoint of designing a computer program for story understanding, are facets of what we call the Script Management Problem.

We wanted to give SAM a sufficient _number_ of Scripts so that it could handle texts from a variety of knowledge domains. We also wanted to put enough _detail_ into the Scripts so that SAM could achieve a reasonable depth of comprehension in each domain. In doing these things, however, we found that the benefits of having Scripts were counterbalanced to some extent by the problems they cause.

We have argued that the most important advantage in having a Script is that it defines a _context_ inside which understanding can proceed. A lot of what we do is governed by rules. There are natural forces which act in invariable ways. A glass released will accelerate toward the ground, every time. Hurricanes are always accompanied by high winds and rain. Institutionalized social situations have a similar character of rigidity. If a person enters a restaurant or a subway, he knows, from long practice, the kinds of things he must do to fit in properly. If working in a restaurant or a subway is his job, his actions are even more circumscribed. In each of these cases, we see a Script in action.

We are not claiming that _all_ of what we do is governed by Scripts. The point is that possession of the appropriate Script can keep us from drowning in irrelevant detail, because it helps us to focus on what is important. Suppose, for example, someone is watching a football game, and one team scores on a long pass. The whole flurry of activity, which would seem disorganized and confusing to someone who has never seen a game before, is reduced in the mind of the knowledgable viewer to the concept "touchdown." Extraneous details are eliminated, in all likelihood they are never seen, because the understander knows the situation, what its rules are, what can happen and when. Someone without the knowledge to guide his attention sees, in William James' phrase, only "a bloomin', buzzin' confusion." The application of Scripts in SAM, we believe, gives it the attention-focussing device it needs to follow what it reads.

We said, however, that Script-based understanding runs into certain problems. One problem in having a number of Scripts around is obvious. The more Scripts available, the harder it is to decide which one to look at next. This is a problem initially, since we must decide which Script to bring in first. It is a more serious problem later, when a story input no longer seems to fit in the Script which has been active.

Another problem with managing Scripts is that they can codify knowledge from very different domains. Consider, for example, the different kinds of knowledge needed to understand the following two sentences:

(3.1) Hurricane Esther lashed the North Carolina coast with high winds and rain last night.

(3.2) Premier Enver Hoxha and Mrs Hoxha arrived in Peking Sunday at the invitation of Communist China.

Example (3.1) is about a natural phenomenon, a hurricane, and its characteristic agents. Understanding (3.1) presupposes some knowledge of physical causality, as manifested in weather patterns. For example, we know that the hurricane did not literally "lash" the land. What is being described are ordinary natural events, wind and rain, but of an unusual severity.

Example (3.2) is a whole world away from this kind of "naive" physical knowledge. It refers to a highly ritualized social occasion, a state visit, and summons up all kinds of information about who the leading actors are likely to be and what they are probably going to do. We know, for example, that the Premier, as leader of his nation, was the one invited, although both he and his wife (and assorted lackeys) travelled to Peking. Because this is a diplomatic situation governed by protocol, we also know that Hoxha has come to meet someone of similar stature, probably the Chinese head of state. Finally, we know that the meeting between these two will be surrounded by other social affairs. It's likely that the Chinese government dispatched an official delegation to meet the Albanians when they arrived, that banquets and other recreations have been scheduled, and that some indication, perhaps an "official communique," will be given of the results of the meeting.

In designing an understander for knowledge domains of this complexity, we are faced with the problem: how much detail should we put in, and which of the details will be useful in deciding whether the Script is the one we want?

People learn a variety of Scripts simply by living. They can also acquire knowledge about situations they haven't actually experienced, such as state visits, by reading and watching television, and comparing the new information with things they know about from daily life. In watching an official party's arrival on the evening news, for example, people can use what they know about airplanes to understand who's coming down the steps from the plane, and who's waiting for them. SAM, however, does not learn the Scripts it applies. To solve the problems created by giving SAM a large number of Scripts of varying complexity, we needed to know what <u>kinds</u> of Scripts there are in the world, and what are the possible <u>interconnections</u> among Scripts.

3.2 Organizing Expectations about Stories

Our discussion of Scripts and their interrelations is based on the ideas of Schank and Abelson [34]. Their theory recognizes three kinds of Scripts which people appear to have. First are Situational Scripts, which are "characteristic of institutionalized public situations (in which) the social interactions are stylized [34, p. 120]." Examples include eating in restaurants and riding on subways. Stories about hurricanes and visiting dignitaries refer to a kind of generalized Situational Script, as well. Next, there are Personal Scripts, which embody ways of achieving a person's idiosyncratic goals independently of the role the person may have in a Situational Script. For example, a man may have a Personal Script for getting a date with a pretty girl, which he can apply to the girl behind the counter in a department store. Finally, there are Instrumental Scripts, which describe rigid action chains, involving a single main actor, appearing in people's everyday activities. Sample Instrumental Scripts are starting a car, stroking a tennis ball, and preparing _coq au vin_.

Since we wanted SAM to read stories which adults would have no trouble understanding, we were concerned with modelling world knowledge which adult readers share, rather than private knowledge. Therefore, SAM's Scripts are Situational rather than Personal Scripts.

What about Instrumental Scripts? We want to argue that the idea of "instrumentality," that actions can serve to support or carry out more important actions, is a key organizing principle for Scripts of all kinds. We have given the name _Situation_ to the Scripts which embody the most important activities in the knowledge domains that SAM handles. Each of the types of newspaper stories read by SAM is organized by a characteristic Situation. There are, for example, Situations for motor-vehicle accidents, state visits and oil spills. In each case we have a class of stories which readers intuitively feel are different. They are interested in the answers to radically different questions in each case. In $VEHACCIDENT, for example, most readers first want to know if anyone was hurt or killed; then whether the police assigned blame to anyone; only later what was the extent of property damage. Oil spills, though they too start with a kind of accident, have a another set of interesting questions. We want to know whether any oil escaped; if it did, how much; and what beaches, flora and fauna are threatened by the spill.

What do Situations look like, and how do they use the idea of Instrumentality? Let's consider as an example the very common Situation, $TRIP. We know that starting a car is an instrument in driving it, in that getting the engine running is a precondition for moving the car. We also know that people use cars to take _trips_ of various kinds. They may go to the store, on a business trip, on vacation, or (if they happen to be Very Important Persons) on state visits. In each of these trips, driving a car is a potential Instrument for getting to a place where a person needs to be to engage in more significant activities. (We're ignoring here the possiblity of driving around "for pleasure.") What we have is a hierarchy of Scripts, each one an Instrument of the one above it. That is, $STARTCAR is subsidiary to $DRIVECAR, and $DRIVECAR is one means of initiating and completing

$TRIP.

In considering $TRIP and its subsidiaries, we observe that they are quite different in structure. $STARTCAR is extremely rigid. There is only a single action chain, and one main actor. The usual outcome is highly predictable. Cars nearly always start. $DRIVE, on the other hand, is somewhat more flexible. What a driver does depends quite a bit on what the other drivers on the road are doing. The routes a driver can take to get somewhere also vary a fair amount. Interferences to $DRIVE are possible. A car may encounter traffic lights and traffic jams. Sometimes a favored route is blocked, and the driver must find another way around. On arriving, there may not be a parking space available.

$TRIP is more flexible yet. At the highest level, all it prescribes is that someone went somewhere, they did something on arrival, then they returned. The choice of instruments for the going and returning, and the possible goal activities, are highly variable. All $TRIP really demands is that the same main actor appear throughout, and that the Maincons of the Scripts which fill in the going and returning parts, or "segments," contain a PTRANS. $TRIP also prefers that the same means be used in returning as in going, but doesn't insist on it.

Can looking at trips in this way give us a clue to managing other Scripts? As an expectation-based understander, SAM makes predictions about what it will read next on the basis of what it has already seen. The more Scripts it has, the larger the number of potential expectations that can be triggered. What we want, therefore, is a method by which the understander can organize its expectations about a text in appropriate ways.

Let's consider how the structure that $TRIP gives us can help SAM process stories about trips. The crucial point to be observed here is that some of the events in trips are important ones, directly concerned with the "point" of the story. Others are less important. $TRIP itself sets up expectations for answering important questions such as "Where is main actor going?" and "What will he do when he gets there?" $DRIVE, on the other hand, expects to see less important things, answers to "What route did he take?", "Was traffic bad?", etc. At the lowest level, $STARTCAR is concerned with "Did the car start?", whose answer is highly predictable.

The use of a multi-level Script, such as $TRIP, with each level contributing its own predictions, lets SAM look for the most important, therefore most likely, events first. Suppose, for example, a story begins:

(3.3) John got in his car.

The Conceptualization corresponding to (3.3) activates the drive-Script, $DRIVE, because it instantiates a Locale Header for $DRIVE. What kinds of things would we expect to happen next? Clearly, reading (3.3) sets up predictions for further events from $DRIVE. We would not be surprised if the next sentence read:

(3.4) He started it up and drove off.

Real stories seldom go into $DRIVE in the detail that (3.4) gives, however. Driving a car is such a commonplace, boring activity that more than (3.3) will not be said unless something out of the ordinary happens, such as:

(3.5) The car wouldn't start.

This sentence refers to an interference event in $DRIVE, and so would be predicted when (3.3) is read. (Details on how this prediction is made are given in Section 4.7.)

In reading (3.3), we form another class of expectations which don't refer to $DRIVE at all. These expectations are concerned with the point of the story, with the answer to "Why are they telling me this?" Instead of (3.4) or (3.5), we are much more likely to see:

(3.6) He drove to the supermarket.

This sentence refers to the trip-Script, $TRIP, in two different ways. First of all, it instantiates the "going" part of $TRIP, since driving is a good way to get somewhere. More importantly, it refers to the "goal" activity of $TRIP, since buying things in a supermarket is one common reason for taking a trip. This goal activity, however, is only <u>predicted</u> by (3.6), since further references to the "going" part may still occur. For example, we may read:

(3.7) He parked in front of the store.

Again, (3.7) instantiates a recognized activity from $DRIVE, since this Script is always terminated by a parking-Conceptualization. But (3.7), like (3.4), is a highly rote, expected activity in $DRIVE. Therefore, it does not seem so likely as an input referring to the "global" context of $TRIP, such as:

(3.8) He bought some groceries.

Here, the supermarket Script which was predicted by the Conceptualization for (3.6) is instantiated. Now we know why John took the trip: to buy some food.

Let's sketch how $TRIP and its subsidiaries organize the predictions which are needed to handle the possible inputs (3.3) - (3.8). (This process is considered in more detail in Chapter 4.) Suppose, for simplicity, that the only Situation known to the system is $TRIP, but that it also knows about the simpler Scripts $BUS, $TRAIN, $DRIVE, $SUBWAY, $RESTAURANT, $MUSEUM and $SUPERMARKET, eight Scripts in all. Associated with $TRIP are the Scripts which can fit into each of its segments, $GOTRIP, $GOALTRIP and $RETURNTRIP:

```
$TRIP:
  $GOTRIP:          $GOALTRIP:             $RETURNTRIP:
        $BUS                $RESTAURANT              $BUS
        $TRAIN              $MUSEUM                  $TRAIN
        $DRIVE              $SUPERMARKET             $DRIVE
        $SUBWAY                                      $SUBWAY
```

Because it is the highest-ranking Script, $TRIP defines an *a priori* processing order for the Scripts in the system, as follows: $TRIP, $GOALTRIP, $GOTRIP, $RESTAURANT, $MUSEUM, $SUPERMARKET, $BUS, $TRAIN, $DRIVE, $SUBWAY. This processing order, in turn, defines a list of expectations:

Script	Expected Input
$TRIP	Main Actor takes a trip
$GOALTRIP	Main actor does a goal activity
$GOTRIP	Main actor goes somewhere
$RESTAURANT	Main actor goes into restaurant
$MUSEUM	Main actor goes into museum
$SUPERMARKET	Main actor goes into supermarket
$BUS	Main actor goes to bus terminal
$TRAIN	Main actor goes to train station
$DRIVE	Main actor gets into car
$SUBWAY	Main actor goes to subway station

(To simplify things somewhat, we are considering only Locale Headers.) Using this list of expectations, the Script Applier would eventually find (3.3), but only after checking first for things which intuitively seem to be more important, such as "John went sightseeing" or "John went downtown."

Sentence (3.3) activates $DRIVE. However, because $DRIVE fits into the $GOTRIP segment of $TRIP, these Scripts are activated as well, and are placed at the head of the search list for subsequent inputs. Now the search list of expectations is:

Script	Expected Input
$TRIP	Main Actor takes a trip
$GOTRIP	Main actor goes somewhere
$GOALTRIP	Main actor does a goal activity
$DRIVE	Main actor starts car
	Main actor drives car away
	Car won't start
	Main actor drives somewhere
$BUS, etc.	
$RESTAURANT,etc.	

Sentences (3.4) and (3.5) are found because of predictions made by $DRIVE. However, the parts of $TRIP are checked out first, and this is how (3.6) is found.

Because $GOTRIP is active, (3.6) would match the prediction for someone travelling somewhere. At this point the Conceptualization for (3.6) would also be matched in the active Script $DRIVE, subsidiary to $GOTRIP. This matches the Script Maincon, so $DRIVE remains active. Furthermore, since (3.6) refers to a possible goal activity, through "supermarket," the associated Script is predicted. Now the order of expectations is:

Script	Expected Input
$TRIP	Main Actor takes a trip
$GOTRIP	Main actor goes somewhere
$GOALTRIP	Main actor does a goal activity
$DRIVE	Main actor arrives at supermarket
	Main actor parks
$SUPERMARKET	Main actor goes into supermarket
	Main actor buys something in supermarket

Here the crucial advantage the Situation provides for managing Scripts has become apparent. Six of the eight Scripts possessed by the system have disappeared from the high-priority search list. The Script Applier's attention has been progressively narrowed by the inputs it has seen. It very quickly finds (3.7), predicted by $DRIVE, and (3.8), predicted by $SUPERMARKET.

This example shows, in outline, how a hierarchically organized knowledge structure, such as $TRIP, can help an understander to arrange its expectations, as a human reader seems to, so as to search for inputs in decreasing order of importance with respect to what the point of a story seems to be. The most important predicted events, those from the highest level of the structure, are looked for first, then events from lesser, subsidiary Scripts. The lesson is that Scripts can be made up of other Scripts. Larger Scripts can "chunk" the knowledge in smaller Scripts. By controlling access to the smaller Scripts, Scripts such as $TRIP provide the solution embodied in SAM of the Script Management Problem.

3.3 Connections Among Scripts

In SAM, Scripts are organized in hierarchical structures called Situations in which the Scripts at one level act as Instruments for the ones at the next level. Scripts on the same level, in turn, interconnect in various ways. Before discussing in detail what the hierarchical structures look like, therefore, we need to have some idea about how Scripts can interact and interconnect.

Possible connections among Scripts can be both temporal and causal. In a sentence such as:

(3.9) I went to a drug store. Then I went to a museum.

the relation seems purely temporal. Going to a drug store does not ordinarily contribute either to the occurrence or non-occurrence of

$MUSEUM. On the other hand, the relation in:

(3.10) I took a bus to the ball park. Then I watched a game.

seems more "causal." We understand that getting to the ball park was the reason for the bus ride. Generally speaking, we can define two different kinds of causal/temporal relationships which Scripts can enter into: (1) <u>sequential</u>, in which Scripts are activated and closed one after the other; and (2) <u>co-occurring</u>, in which Scripts go on simultaneously, sometimes interacting, sometimes not.

3.3.1 Scripts in Simple Sequential Relation

The simplest relation among Scripts is sequential. Here, Scripts occur one after another, and one Script runs to completion without affecting the next, except, perhaps, to get an actor to the next one's setting. Examples (3.9) and (3.10) show Scripts in sequential relation. In (3.9), $DRUGSTORE and $MUSEUM follow one another in chronological order, with the exit from $DRUGSTORE being inferred to have happened before $MUSEUM begins. The use of "then" in (3.9) emphasizes the sequential relation.

The same temporal ordering occurs in (3.10), but now we understand that $BUS contributes a key precondition for $BALLGAME. It is "instrumental" in the Locale Header for $BALLGAME. The insertion of "then" in (3.10) makes the passage sound redundant, since the second Conceptualization could have been predicted from the first. Whenever a causal connection of the kind illustrated by (3.10) occurs, we suspect that we're in the presence of a more global Situation in which predictions of this kind are explicitly made. Example (3.10) is another instance of the Situation, $TRIP.

In both (3.9) and (3.10), it is important to observe that the setting in which the Scripts take place changes. In fact, the constraint on where Scripts can happen is so useful that we have given it a special name:

> The Locale Principle:
> If successive Conceptualizations from a story refer to Scripts which cannot have the same setting (e. g., organization-Scripts in which the place of business is different), then the relation between the Scripts must be sequential.

The Locale Principle is simply a way of stating an idea of causal continuity. In stories with a single main actor, Scripts which have different settings cannot co-occur, unless we infer that the actor has moved. For example, the story:

(3.11) Irving ate some lasagna in a restaurant. Then he bought a watch.

seems to require this inference. Not enough information is given in (3.11) to make a clear decision about locale possible. Since we aren't

told which restaurant is involved, we can't tell whether the main actor, Irving, has moved. In SAM, "indefinite" references to organizations (e. g., "a restaurant" vs. "Mamma Leone's") having different places of business will cause the Script associated with the first organization to run to completion before the second one is started. This is accomplished using static information stored in the first Script prescribing which other Scripts can be in "simple-sequential" relationship with it.

Sometimes the simple-sequential relation can be stated for entire classes of Scripts. An important collection of Scripts which are clearly "orthogonal" to each other, that is, which must occur in simple-sequential relation, are those whose Maincons are a PTRANS-Conceptualization. These Scripts are associated with organizations, called PTRANS-Organizations, which move people or cargo from one place to another. Examples of PTRANS-Organizational Scripts are $PLANE, $TRAIN, $BUS, $SUBWAY and $AMBULANCE. It simply isn't possible to be moved simultaneously by more than one of these. The Script Applier, therefore, closes any active Script of this kind when another one is initiated. For example, reading the passage:

(3.12) John went to New York by bus. He took the subway to Shea Stadium.

would result in an instantiation of the default path through $BUS, connected sequentially to an instantiation of the default path of $SUBWAY. (Schank and Abelson [34] call references to Scripts consisting of a single Conceptualization, such as in (3.12), "fleeting references.")

Another way of describing the orthogonality of PTRANS-Scripts, using the Locale Principle, is that the settings of these Scripts (the inside of the associated vehicles) are distinct. The analogous collection of Scripts involving personal vehicles which people use to move themselves around (e. g., $BIKE and $DRIVE, but <u>not</u> $WALK) share the same property. These must occur in sequential relation with each other, and with PTRANS-Organizational Scripts.

We also know that PTRANS-Scripts must be sequential with Scripts carried out by organizations whose "place of business" is fixed. Examples of organizations with a fixed "residence" are museums, churches and theaters. It is characteristic of many of these Scripts that their Maincons are complex, partly characterized by Scripts rather than simple actions. What, for example, is the main activity a playgoer engages in? It's an extended MTRANS, of course, but it is much more than that. "Applause" and "enjoyment" are also involved. Scripts of this kind are legitimate "goal" activities of persons who take trips. SAM, therefore, shifts from the going-segment of $TRIP, where the Maincons are a simple PTRANS, to the goal-segment of this Script, when a reference to one of these activities is read. The representation of:

(3.13) John went to New York by bus. He took the subway to Shea Stadium. He watched a doubleheader.

would consist of the sequential instantiation of the PTRANS-Scripts $BUS

and $SUBWAY (as in (3.12)), followed, in sequence, by the instantiation of the goal-activity Script $BALLGAME.

3.3.2 Scripts Occurring in the Same Place

Clearly, for Scripts to interact they must share actors and go on, at least part of the time, in the same place. We call Scripts which can happen in the same place "locale-nested," or simply "nested." This relation among Scripts is a very common and complicated one. In SAM, nested Scripts can enter into three different relationships: (1) sequential, (2) parallel-unaffecting, and (3) parallel-affecting. Certain kinds of information about these relations are stored permanently in active memory. However, in many cases specific world knowledge can modify this "a priori" information.

Specific world knowledge about a role from a Script can modify the Script at the time it is activated. (The process of invokin(" Script, and the other processes that go on during Script Application, are considered at length in Chapter 4.) Suppose we read one of the following:

(3.14) Mary walked into Woolworth's and asked for some bobby pins.
(3.15) Mary walked into Woolworth's and asked for some ice cream.
(3.16) Mary walked into J. C. Penney's and asked for some ice cream.

The first clause in each of these examples activates the $VARIETYSTORE context, since the corresponding Conceptualizations match a Locale Header for this Script. In each case, "Mary" assumes the role of main actor, and "Woolworth's" and "J. C. Penney's" take on the role of the organization executing the Script. When the second part of (3.14) is read, we infer that "asked for" is a preliminary to the standard ATRANS of money for one of the goods the store typically sells. In (3.15), however, the inference is that this event is an order at a lunch counter, since Woolworth's doesn't ordinarily stock ice cream but does have a kind of restaurant. Note that both (3.14) and (3.15) potentially refer to Scripts in sequential relation. A story beginning with them may contain further references to Scripts, nested in the store setting, which will be interpreted as happening after the eating/buying Script has been completed. In (3.16), however, our detailed knowledge about Penney's tells us that Mary will be disappointed, since this store doesn't ordinarily sell food.

SAM contains machinery for handling inputs such as (3.14) and (3.15), and for other important classes of locale-nested Scripts. (SAM currently cannot do subtle examples such as (3.16).) The key idea is provided by the Locale Principle. In Chapter 2, we discussed Script Headers, the set of patterns which activate a Script, bringing in its predictions about things which may be mentioned next. One important kind of Header is the Locale Header, a pattern looking for movement of the main actor to a place (such as an organization's place of business) which is strongly associated with a Script. When such a setting is

reached, the associated Script is activated. The point to be made here is that Locale Headers are used in SAM to activate <u>all</u> the Scripts which can go on in that place.

The simplest kind of nesting occurs in a setting in which several different Scripts clearly belonging to one organization can happen. This class of examples really refers to different tracks of the same Script, which go on sequentially if more than one occurs. The Woolworth's lunch-counter example (3.14) illustrates this relationship. The Script $VARIETYSTORE which (3.14) instantiates has a subsidiary Script $RESTAURANT which is activated at the time Woolworth's is entered. In this case, the ordering of expectations, as defined by $VARIETYSTORE, is:

Script	Expectation
$VARIETYSTORE	Patron goes to sales counter
	Patron buys "hard" goods
$RESTAURANT	Patron goes to lunch counter
	Patron orders food

Another example is a bank, the Script for which has tracks $WALKUPTELLER, $DRIVEUPTELLER, $OPENACCOUNT, $MAKELOAN, $SAFEDEPOSIT, etc. A person entering a bank will ordinarily engage in only one of these Scripts, but more than one is possible. If more than one occurs, these will be interpreted by SAM as happening sequentially in the bank-locale.

Sequential nesting may occur in a given place even though there isn't a primary Script present. There are places where many different activities can go on, no one of which would be more strongly predicted than the others. Suppose, for example, a story starts:

(3.17) Bill walked into Central Park.

The setting "Central Park" does not house one dominating Script, as "a restaurant" or "a subway" does. Many different kinds of things can happen. For example, we may next see references to $PICNIC, $PLAYGAME, $ZOO, and (in certain parks) $THEATER. Because the setting "Central Park" has these Scripts associated with it, but not, for example, $SUBWAY, SAM would have no trouble recognizing that:

(3.18) Bill walked into Central Park. He looked at the animals.

is a reasonable story. Reading a Conceptualization for entering Central Park would cause SAM to predict that events from the Scripts which have Central Park as a possible setting might occur. The second sentence of (3.18), which matches a Maincon from $ZOO, would be found on this basis.

Consider, on the other hand, the story:

(3.19) Bill walked into Central Park. He bought a token.

The first sentence of (3.19) activates $PICNIC, $ZOO, $PLAYGAME and

$THEATER, as before. The second sentence, which seems to fit into $SUBWAY, would not be found among the events predicted by these Scripts, and SAM would boggle. Some other mode of processing would be needed to make sense out of (3.19). Script Applying could not handle this, because the activation of Scripts depends on setting, and the subway Script cannot happen in a park.

Examples (3.18) and (3.19) illustrate, in its most extreme form, the mechanism of narrowing of attention among a collection of Scripts that SAM uses to screen out contexts which probably will not be applicable to a given story. The presence of setting is constantly exploited to avoid examining Scripts which cannot be useful because they cannot be instantiated in an established setting. Only when the main actor moves to another location will the Scripts that were previously ruled out be checked again for applicability. If SAM reads:

(3.20) Bill walked into a restaurant.

Scripts which cannot occur in a restaurant -- most of the Scripts, in fact, possessed by the understander -- will never be seen.

On the other hand, if the setting contains both a main Script and some subsidiary Scripts, matching of a Locale Header, such as (3.20) corresponds to, will result in the activation of all these Scripts. The list of active Scripts is dominated by the primary Script, but the others are available to the processor. Suppose, for example, (3.20) were followed by:

(3.21) He used the phone.

Reading (3.20) would have resulted not only in $RESTAURANT's being invoked, but also ancillary Scripts such as $BATHROOM and $PHONE which can occur in a restaurant. Note that these Scripts are not part of $RESTAURANT, as, say, $SERVE is, but are auxiliary services which the place provides.

Subsidiary Scripts such as $PHONE, if they are instantiated, are defined to be in "parallel-unaffecting" relationship with the main Script. They are assumed to occur as a unit and not to have any direct effect on the main Script, if this is activated as well. In (3.21), the activation of $PHONE only places $RESTAURANT in the "pending" category. If a reference to $RESTAURANT occurs, $PHONE will be completed, and placed in parallel-unaffecting relation to $RESTAURANT, which now assumes control of processing. "Pending" here means that $RESTAURANT-specific predictions about seating, ordering, etc., will be made, as though this Script were going on. (See Section 4.4 for details on the prediction process.) So, for example, if (3.21) were followed by:

(3.22) He ordered a hamburger.

the Script Applier would use these expectations to conclude that $PHONE should be closed. The default path in $RESTAURANT involving looking for and sitting down at a table, etc., would then be instantiated and attached at the end of the story representation.

Even if a reference to $RESTAURANT, such as (3.22), does not occur, this Script will not be deactivated until the actor leaves the restaurant locale. The primacy of $RESTAURANT with respect to its subsidiaries in examples such as (3.21) can be seen in the alternative way that (3.21) can be expressed:

(3.23) He used their phone.

Here, the reference to "their phone" clearly indicates the restaurant organization.

An important special case of parallel-unaffecting Scripts occurs when a conversation goes on in a Script's setting. (We're assuming here that conversations have some of the properties of Scripts.) Such conversation does not belong to the Script, _per se_, but the Script may provide a facility for it to take place. If we read:

(3.24) On the bus John talked to an old lady.

we infer that John and the old lady were probably sitting next to each other, and that their proximity was what triggered the conversation. The bus Script would not contain an explicit prediction that the conversation would take place, but provides a place for it to occur. The heuristic that SAM uses to handle "spurious" conversations of this kind is: whenever actors form a group, even if only momentarily, make a low-level prediction that casual dual-MTRANS events may occur. This is used, for example, in the processing of:

(3.25) As John left the bus, he thanked the driver.

where the Script contains the causal-chain information that passengers must pass the place where the driver sits as they exit from the bus. If the driver is there, then a chance for a conversation exists. However, since these conversational predictions are added to the end of the search list of patterns, they will not interfere with the recognition of MTRANS events which are a true part of the Script:

(3.26) John went into a restaurant and sat down. The waiter came over and asked him for his order.

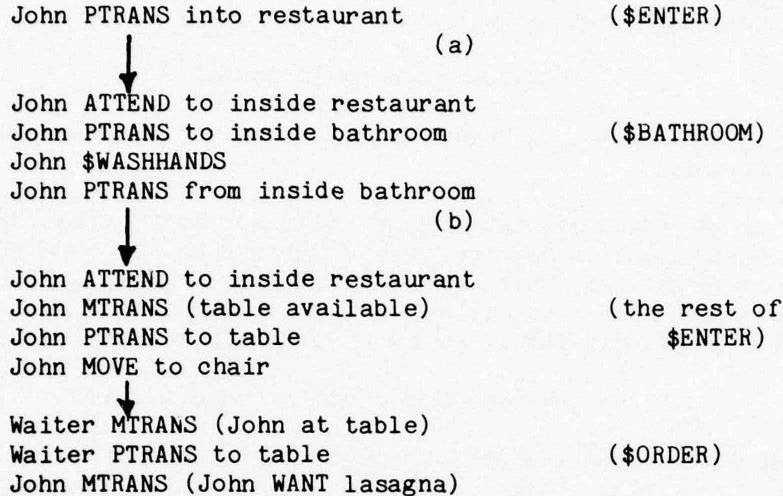
In (3.26), the waiter and John form a group when "asked for his order" is read. This, however, would be identified as a legitimate part of the ordering scene of $RESTAURANT.

SAM records the temporal relation between a main Script and a subsidiary, if both occur, but normally assumes that there is no causal connection between them. The subsidiary Script is treated as a "side effect" of the primary Script. Suppose, for example, we have:

(3.27) John went into a restaurant. He washed his hands in the bathroom. He ordered some lasagna.

The story representation for (3.27) would contain three sequential segments, one from the "entering" scene of $RESTAURANT, one for a "nominal" path through $BATHROOM, and one for the "ordering" scene of

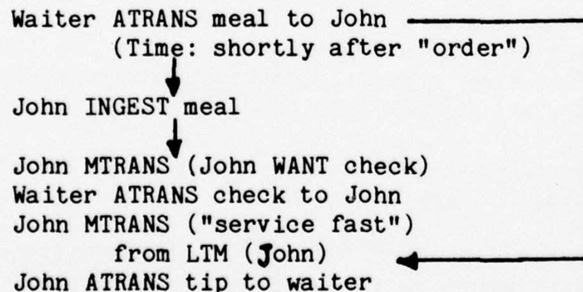
$RESTAURANT. The instantiated causal chain would look roughly like this:



The point to note here is that there are no causal connections recorded between the pieces of $RESTAURANT and $BATHROOM, except at the boundaries (points (a) and (b), above). This is to be contrasted to a case where an event that happened earlier has an effect later in the story, as would be the case if (3.27) continued:

(3.28) His lasagna came quickly. He left a large tip.

In this case the causal chain would look like:



Here, the episode about the fast arrival of the lasagna, remembered at the time John got the check, is the reason for the large tip.

So far we have considered cases in which Scripts referred to by a story did not directly affect one another, even though they might have happened in the same place. Let's consider some cases where active Scripts can have real interactions.

One way two Scripts can interact is if they are going on in the same place and <u>at same time</u>. (We call this a "parallel-affecting" relationship among Scripts.) Then an event from one Script can interfere with what's going on in the other. When this happens, an <u>inference</u> is needed to establish the connection between the Scripts. Suppose, for

example, we have:

> (3.29)
> John got on a train in Penn Station. He went to the dining car and ordered some soup. As the soup came, the train pulled into Newark. The train lurched, and the soup landed in his lap.

The first sentence of this story initiates the passenger-train track of $TRAIN, and its subsidiary Script, $RESTAURANT. The first part of the second sentence corresponds to a Locale Header for $RESTAURANT, and the Scripts run in "parallel-unaffecting" relationship, as usual. (The incarnation of $RESTAURANT brought in with $TRAIN prefers "dining car" or "snack counter" for the restaurant organization. How these preferences are set up is discussed in Chapter 4.)

In the third sentence of (3.29), we have references to both of the active Scripts, and an assertion of a temporal relation between them. The problem here is: are they causally connected, as well? To a human reader, the connection seems accidental. It seems absurd to assert either possible causal relation:

> (3.30) Because the train pulled into Newark, the soup came.
> Because the soup came, the train pulled into Newark.

Why do the sentences (3.30) sound silly? First, we need to observe that the two parts of these sentences refer to boring, commonplace events in their respective contexts. No one reading about a train ride would be surprised at a reference to the train's arriving at a station somewhere. Similarly, reading "the soup came" in a story about $RESTAURANT is a typical way of expressing the highly expected "serving" event. $TRAIN and $RESTAURANT explain "why" each of these events occurred, so it seems pointless to look for a connection between the Scripts. In the next sentence, however, "the train lurched" seems perfectly acceptable as a reason for "the soup spilled." A human reader has very little trouble establishing a causal relation between the sudden movement of a conveyance and movement of things inside.

How can a Script-based understander distinguish between cases such as these? When more than one Script is active, and successive story Conceptualizations refer to different Scripts, the rule of thumb "no damage, no connection" is followed. If one event is marked as being temporally after the other, the Script Applier examines it to see if it instantiates a pre-defined interference event in the Script. If it does not, no causal connection is sought.

The events "train arrived" and "soup came" in (3.29) are both marked with pathvalue "default" in their respective Scripts, rather than with pathvalue "interference." Therefore, SAM doesn't look for a link between the Scripts. Even in the case where the first event is an interference in its Script, the second event must be an interference also before any further processing is done. This would handle the case:

(3.31) The train was delayed in Newark. John's
soup came.

in which, to a human reader, the first event, though an interference, is irrelevent to the second.

In the next sentence of (3.29), however, we read "the train lurched," then "the soup spilled." A reference to the movement of the train, while perfectly possible in a story about a train ride, is only very weakly predicted by $TRAIN. Let's assume, however, that it is eventually found. The Conceptualization for "the soup spilled" is a weakly predicted interference with the normal course of happenings either in the "serving" scene (the waiter could have spilled the soup), or in the "eating" scene (the patron could spill the soup himself). In either case, "spilling the soup" is a pre-defined interference in $RESTAURANT, and we have a case where a connection between two active Scripts should be sought. Successive Conceptualizations refer to alternative Scripts, and the last event is an interference event.

How could a story understander go about determining whether a causal connection exists in a case like this? What we need here is a program for calculating a causal chain between arbitrary events. A Script Applier alone could not compute such connections. Strictly speaking, it doesn't really know anything about causality. The causal chains it possesses as part of its Scripts are pre-defined, given to it as data.

A generalized causal-chain calculator, however, is something nobody yet knows how to build. (For a discussion of the problems associated with computing causal chains, see [28].) We do know that any such program, if it runs without direction from other knowledge sources, runs the risk of an explosion of inferencing such as we discussed with regard to Rieger's Conceptual Memory program in Chapter 1. (As a class, automatic theorem-provers suffer from the same problem.) Every causal-chain connection calculated is itself a candidate for further inferencing. To a causal-chain builder operating by "formal," i. e., syntactic, methods, any inference is as good as any other. Such a process soon becomes clogged up, because it has no way to recognize either "promising" or "absurd" paths.

How could world knowledge, specifically the episodic knowledge of Scripts, aid a causal-chain builder in connecting events such as "train lurched" and "soup spilled." One thing is true about these events: if they were recognized in a Script, they have causal connections in that Script. The Script can propose reasons why they occurred, and the causal chain of which they are a part includes what their expected results, in context, should be. A person's knocking dishes over and spilling things on himself is an accident which can occur in the "eating" scene of $RESTAURANT. If it does happen, it has predictable results. The waiter will bring napkins or cloths to help clean up, and will ask whether the patron wants something else. As for what happens before the event "the soup spilled," we might have the little causal chain:

AD-A056 080 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 6/4
SCRIPT APPLICATION: COMPUTER UNDERSTANDING OF NEWSPAPER STORIES--ETC(U)
JAN 78 R E CULLINGFORD N00014-75-C-1111
UNCLASSIFIED RR-116 NL

(a)	patron PROPEL meal from table	(patron knocks his meal off the table)
	gravity PTRANS meal to patron	(meal falls on the patron)

The event marked (a) is the reason recorded in $RESTAURANT why meals land in people's laps: out of clumsiness, they knock them off the table. This causal connection could be made available to a causal-chain builder by the Script Applier as part of the immediate context in which the event "the soup spilled in his lap" was located.

It is not as clear that "the train lurched" is part of $TRAIN. While this event happens often enough, it ordinarily has no effect on events. Suppose, however, we include a small causal chain describing its usual results:

(b)	train PROPEL obstruction	(train hits something on the track)
(c)	train PROPEL objects	(objects inside the train experience a force)

Event (b) is the realization in the Script of the event "train lurched," and (c) is its expected result: that movable objects inside the train are likely to move. (Let's ignore the problems with marking PPs as movable.) The process of causal-chaining for this example, then, would involve pattern-matching on Conceptualizations (a) and (c). If "train" could be substituted for "patron" and "soup" were known to be a movable object inside the train, we would have the connection we need. (Actually, the system would have to know that "soup" comes in a rigid "container" at meals, and shares its motion -- up to a point.)

3.4 Fitting Scripts Together

Section 3.3 discussed some of the important ways Scripts can interconnect. Now we need to describe how SAM manages these connections, both to understand a story and to build a representation for it afterwards. The basic idea is that Script-based story understanding relies on a <u>hierarchy</u> of Scripts. The Script at the top level is called a Situation. Situations encode the system's expectations about what the main point will be of a story about a given knowledge domain. Scripts at lower levels serve as Instruments of higher ranking Scripts, and contribute expectations which are of a more "local" nature.

To illustrate this, let's consider again the trip Situation, $TRIP. This Script organizes our knowledge of how and why people travel back and forth to certain places. $TRIP has three segments, or scenes: (1) $GOTRIP, which describes how people get to places; (2) $GOALTRIP, which defines the kinds of activities people typically engage in when they arrive at where they want to be; and (3) $RETURNTRIP, which describes how they return to where they started.

$GOTRIP and $RETURNTRIP require PTRANS-Scripts for their instantiation, and will accept several of these Scripts occurring in sequence. $GOALTRIP requires for its realization Scripts which can be goal activities of trips. (Many things that people do on trips, of course, are not explained by Scripts at all. SAM is not intended to handle stories of this kind.) $TRIP, therefore, defines a hierarchy which looks like this:

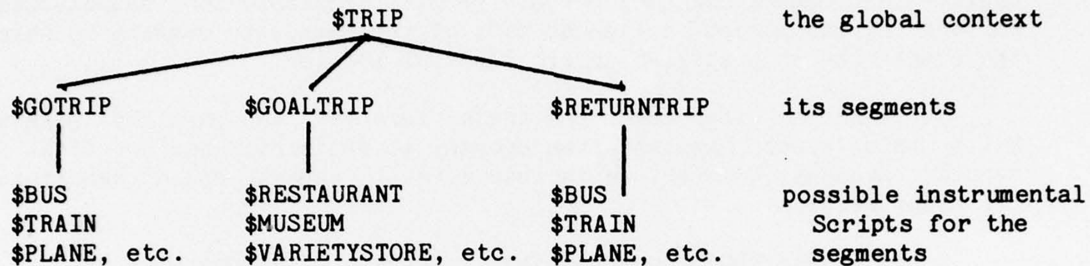


Figure 3.1
The Hierarchy Defined by $TRIP

Figure 3.1 represents a <u>processing</u> hierarchy, since it prescribes the order in which Scripts are examined. For example, the Script Applier, if it thinks $TRIP may be applicable, will look first for Conceptualizations corresponding to inputs such as "John took a trip." Next, it will search for inputs which are characteristic of $GOTRIP, such as "John went downtown," or one of its Instruments, such as "John took the subway downtown." Finally, it will look for Conceptualizations which refer to $GOALTRIP, such as "John went into a restaurant." In this case, it will infer that $GOTRIP has already occurred.

Figure 3.1 also describes the form of the global <u>structure of the story representation</u> for a text about a trip. For example, the representation for the story:

> (3.32)
> John took the bus to New York. He rode the subway to Leone's. He entered the restaurant and had some lasagna. Later he returned to New Haven.

would look like this:

```
!STORY:
  VALUE (SEQ SCLAB1)

SCLAB1:
  VALUE (SEQ SCLAB2 SCLAB3 SCLAB4)
  TOP $TRIP
  MAINCON  (<=>($TRIP MAIN "John"
                      ORIG "New Haven"
                      GOAL "$RESTAURANT")))
```

```
SCLAB2:
  VALUE (SEQ "$BUS" "$SUBWAY")
  TOP $GOTRIP
  MAINCON  (<=>($GOTRIP MAIN "John"
                    ORIG "New Haven"
                    GOAL "$RESTAURANT")))

SCLAB3:
  VALUE (SEQ "$RESTAURANT")
  TOP $GOALTRIP
  MAINCON  (<=>($GOALTRIP MAIN "John"
                    ORIG "New Haven"
                    GOAL "$RESTAURANT")))

SCLAB4:
  VALUE (SEQ "$SUBWAY" "$BUS")
  TOP $RETURNTRIP
  MAINCON  (<=>($RETURNTRIP MAIN "John"
                    ORIG "New Haven"
                    GOAL "$RESTAURANT")))
```

The representation derived from the hierarchical structure defined in Figure 3.1 says that this story is about a trip, by a main actor "John," whose goal is executing the restaurant-Script. The trip has a "going" part, consisting of the simple-sequential (SEQ) instantiation of the PTRANS-Scripts $BUS and $SUBWAY. It has a "goal" part, consisting, in this case, of an instantiation of $RESTAURANT alone. Finally, it has a "returning" part, which, because (3.31) doesn't explicitly say, is inferred to have the same components as the "going" part, executed in the reverse order.

3.4.1 Scripts Involving Organizations and Forces

A Situation defines a hierarchical structure of Scripts. Are there any systematic differences among Scripts at different levels in the hierarchy? Answering this involves a consideration of several factors, including:

1. How rigid are the action-chains that make up its episodes? Are alternative ways of accomplishing the "goal" of the behavior provided?

2. What are the characteristic roles and props in the episodes? Is there a "main actor" in the episode?

3. How much detail does the event chain possessed by an average adult contain? Can the understander fill in a large amount of information, or is what he can say relatively sketchy?

4. How does language make reference to the knowledge structure? Are there words which "clump" activities into large chunks? Are there special senses of words and phrases which are tied to a particular situation?

If we examine Situations such as $TRIP and $VEHACCIDENT, we find two broad classes of Scripts. First are Scripts which describe a stereotyped interaction between an organization and a member of the public. These Scripts, which are called Transaction Scripts, include PTRANS-Scripts such as $BUS and $SUBWAY, other commercial Scripts such as $RESTAURANT, $VARIETYSTORE, $BANK and $SUPERMARKET, and "service" Scripts such as $AMBULANCE and $FIREDEPT. The other major class of Scripts describes the characteristic events associated with natural forces. The forces involved may be simple forces, e. g., gravity, mechanical forces such as are generated by the engines in cars and trains, or complex forces such as hurricanes and earthquakes.

3.4.2 Transactions

Let's consider first the characteristics of Transactions. We use the term Transaction because some idea of a contract is always involved in these Scripts. A member of the public goes to an organization providing a service of some kind and contracts for the service. In a restaurant, for example, the patron "orders" a meal, that is, arranges for the restaurant to prepare and serve the meal in exchange for later payment. In a subway, the rider pays the cashier and gets a token in return. The token symbolizes the subway organization's resposibility for providing a ride to the patron on demand. Bus and airplane tickets serve a similar purpose.

A Transaction is built up out of the routine actions of the organization's agents as they deal with a member of the public. $RESTAURANT, for example, organizes the sub-Scripts of the roles of waiter, cook, cashier, etc., in the course of providing a meal to a patron. The Maincon of a Transaction is always the service event. Contracting for the service always precedes the Maincon. The organization needs to be told what exactly it is that the patron wants done. The order in which the Maincon and the "paying" event occur depends on the details of the situation. In a fast-food place such as McDonald's, one pays before getting one's food. The reverse ordering is typical in ordinary restaurants.

We call Scripts such as $BUS, $RESTAURANT and $THEATER commercial Transactions because providing these services for a profit is the organization's raison d'etre. Commercial organizations characteristically compete with one another for the public's business. Often they adopt special symbols (e. g., McDonald's Golden Arches) or specially designed buildings (e. g., theaters and banks with drive-up windows) which signal their Script to the public. There is another class of Scripts, belonging to governmental and other service organizations, which share some of the characteristics of Transactions. In these Scripts, called service Transactions, an organization provides a service to the public without being directly repaid. Examples are police, fire and ambulance services. These kinds of organizations do not compete with each other, as commercial organizations do, but are services provided by the community in return for the tax money the citizens pay.

How do these observations help in the job of story understanding? First of all, they provide a blueprint for putting together a multitude of different Scripts. If the Script is about an organization and a person interacting with one another, the "kernal" of the Script is the three events of contracting, the service and paying. (In service Transactions, the paying event is implicit.) The other activities of the Script are arranged around the kernal events according to whether they contribute a precondition for the kernal event, or are a result of the kernal event. In $RESTAURANT, for example, the kernal events are ordering, eating and paying. Entering and sitting down are preconditions for ordering. Preparing and serving the food are preparatory for the Maincon. Getting the check is necessary for paying it.

The structure of Transactions also assists in applying the Script. The crucial inference that needs to be made in a story referring to a Transaction is whether the Maincon has occurred. The Script Applier will not infer the Maincon unless it is either explicitly instantiated, or a confirming (that is, succeeding) kernal event occurs.

Suppose a story about $RESTAURANT begins:

(3.33) John entered a diner.

Having read (3.33), SAM now expects to hear about a contracting event, viz., the ordering of a meal. Until such an event is instantiated, the system will not set up an expectation for the Maincon. Suppose now we read about a contract:

(3.34) John asked for a coke.

When (3.34) is read, the system is satisfied to infer that the main INGEST event has occurred if it reads something that can be interpreted as a paying event:

He gave the cashier a dollar.

This is because paying succeeds the Maincon in $RESTAURANT, and hence is a confirming kernal event.

This is to be contrasted with the following two cases, in which SAM will <u>not</u> infer the Maincon:

(3.35) John entered a diner. He gave the cashier a dollar.
(3.36) John entered a diner. He asked for a coke. He left.

In (3.35), we can't be sure that the kernal ordering event has occurred. In (3.36), we haven't heard about either eating or paying, so again the Maincon is not inferred. In both these cases, the system constructs an "abort" path out of $RESTAURANT, without inferring a reason for the exit.

3.4.3 Natural Force Scripts

The other broad class of Scripts involved in the Situations SAM handles are those in which natural forces appear. The structure of force Scripts is simpler and much more rigid than that of Transactions. This is simply because forces have no "intentions" and act in invariable ways. The force Scripts used by SAM are very sketchy. Newspaper stories seldom get into the physical details of what the forces are doing. Therefore, we have taken a "naive" physical view of forces. The Script contains a single Maincon, with connections to the most important events resulting from the application of the force.

One kind of force Script which appears in newspaper stories describes a mechanical force which has gotten out of control. Mechanical forces are generated by artificial objects such as engines, and ordinarily operate at the behest of a person executing a Script. For example, the sentence "John drove his car to a restaurant," when amalgamated with the Script structure $DRIVE, has the representation:

```
((ACTOR HUMO <=> (*PTRANS*) OBJECT STRUCTO
      TO (*PROX* PART ORGO)
      INST ((CON
           ((ACTOR HUMO <=> ($DRIVE)))
                LEADTO
           ((ACTOR FORCEO <=> (*PROPEL*)
                OBJECT STRUCTO))))))
HUMO: "John"
STRUCTO: "car"
ORGO: "restaurant"
FORCEO:
  CLASS (#NATFORCE)
  TYPE (*ENGINE*)
  PARTOF (STRUCTO)
```

This Conceptualization says that "John took the car to a restaurant," with an Instrument "John drives", controlling "engine moves car". If now we read "the driver lost control of the car":

```
((ACTOR HUMO <=> ($DRIVE VEHICLE STRUCTO))
      MODE (*CANNOT*))
```

memory makes the inference "the car ran out of control":

```
((ACTOR FORCEO <=> (*PTRANS*) OBJECT STRUCTO))
```

This, of course, is a Header for the Situation $VEHACCIDENT, since cars out of control soon run into something.

Complex natural forces, such as hurricanes and earthquakes, also have Scripts which newspaper stories talk about. Unlike mechanical forces, complex forces cannot be controlled. They "just happen." Complex forces characteristically use simpler forces as their "agents." If we read "Hurricane Darlene brought high winds to Connecticut," we understand that the complex force "Hurricane Darlene" is being described as the ACTOR in a PROPEL Conceptualization, moving itself and its agent,

the wind, around:

"Hurricane Darlene brought high winds to Connecticut"

```
((CON
   ((ACTOR FORCE1 <=> (*PROPEL*) OBJECT PHYS1
          TO (*PROX* PART POLITO)))
                  LEADTO
  ((ACTOR FORCE2 <=> (*PROPEL*) OBJECT POLITO))))

  FORCE1:
    CLASS (#NATFORCE)
    TYPE (*HURRICANE*)
    FORCENAME (DARLENE)

  PHYS1:
    CLASS (#PHYSOBJ)
    TYPE (*AIR*)

  FORCE2:
    CLASS (#NATFORCE)
    TYPE (*WIND*)
    STRENGTH "high"

  POLITO: "Connecticut"
```

Sometimes the complex forces are spoken of as acting directly:

"An earthquake struck Rumania."

```
((ACTOR FORCE3 <=> (*PROPEL*)
           OBJECT POLIT1))

FORCE3:
  CLASS (#NATFORCE)
  TYPE (*EARTHQUAKE*)

POLIT1: "Rumania"
```

In any case, the interest that newspaper readers have in complex forces is not in the details of how they work. The man in the street doesn't know much about that. What he wants to know are, what were the main effects of the phenomenon?, and how did the area involved react? That is, complex forces fit into another kind of Situation, in which questions like these are explicitly asked.

3.4.4 Situations

> Reading the newspaper is completely unnecessary. If you've read about one train wreck or earthquake, you've read about them all.
>
> Henry Thoreau, Walden.

Each distinct knowledge domain SAM deals with is assigned its own global Script, or Situation. We have already described the characteristics of one Situation, $TRIP, in some detail. What do other Situations look like?

First of all, like $TRIP, Situations such as $VEHACCIDENT and $OILSPILL organize the knowledge contained in smaller Scripts. In $VEHACCIDENT, for example, we have an instance of a mechanical force Script, $CRASH, in which a driver loses control of a motor vehicle and is involved in a collision. After the crash, various service organizations, such as ambulance, police and emergency squad, appear at the scene and perform their Scripts. We may also hear about the activities of emergency rooms and operating rooms at a hospital. Also, like $TRIP, these Situations are relatively sketchy. The main reason for this is that ordinary people simply haven't been involved in these kinds of events personally. Very few of us have experienced $EARTHQUAKE or $FORESTFIRE, for example. We may never have seen an oil spill. All we know about them is what we have read in the newspapers that makes sense in terms of our own personal experience.

A related characteristic of these Situations is that readers are interested in the answers to rather general questions, instead of the details of what happened. (Probably they aren't equipped to understand the details, anyway.) In stories about accidents and natural disasters, we all want to know if anyone was hurt or injured, and what the authorities are doing about it. If there's blame to be assigned, we want to find out who was blamed. As in $TRIP, where the answers to "Where is he going?" and "Why is he going there?" are explicitly encoded in the knowledge structure, the answers to questions about injuries and blame appear as expectations in the Situation. For each Situation, there is a _scorecard_ of questions whose answers the processor wants to find.

Actors and objects in stories handled by Situations may have several roles. They may, for example, fit both in their local Script (a Transaction or force Script) and in the Situation. They may also have roles in more than one component Script in the Situation. To see this, consider a simple car-accident story:

> (3.37)
> Sunday evening an 18-ton tractor'trailer ran off the Connecticut Turnpike and struck a bridge abutment. Frank Smith, 37, the driver, was critically injured. He was taken to Yale-New Haven Hospital by Flanagan Ambulance.

In this story, "Frank Smith" belongs to the component force Script $DRIVE in the role of driver. He simultaneously fills the roles of

"hurt person" in the Situation, $VEHACCIDENT, and of "person taken for medical treatment" in the service Transaction $AMBULANCE. Even objects in newspaper stories may have multiple roles. For example, the tractor trailer has the role "vehicle" in $DRIVE. If a follow-up story to (3.37) contained the sentence "The trailer involved in the accident was declared totally lost," the trailer would be filling the role of "damaged vehicle" in an insurance company Transaction.

A specialized role in every Situational domain is that of "authority figure." This is a person or group of persons representing the organizations involved in the Situation who make pronouncements about what happened. In $VIPVISIT, for example, a typical authority figure is "a White House spokesman." In $OILSPILL, an authority is "the Coast Guard." In accidents and disasters, an "eyewitness" is a person who happened to be at the scene of the event (and, in the case of a disaster, lucky enough to have lived through it). Since, in SAM, we deal with "overt" or "physical" circumstances that no one would want to lie about, the statements of authority figures are assumed to be true. This is the basis of the Authority-Announcement inference, described in Chapter 5.

SAM deals with three major kinds of Situations. First are specializations of $TRIP. One special case of $TRIP that often appears in newspapers is $VIPVISIT. A visiting dignitary is indeed engaged in a trip, but, because he is a special person, it's likely that special modes of transportation will be used to get him where he's going. The President of the United States doesn't ride a bus to the airport to fly Pan Am, but is taken by helicopter to an Air Force base to fly US 1. More importantly, his goal activities are extremely specialized. (Not all of them, of course, are accounted for by Scripts.) There is nearly always an official welcome when the visiting party arrives. There may be a parade in their honor. There are banquets, visits to new buildings, and other recreations. Finally, there are "official meetings" and, perhaps, an "official communique" describing the outcome of the meeting. Eventually, the Very Important Person returns to where he came from.

Since $VIPVISIT has the same global structure as $TRIP, it could be summarized in the same way. Suppose we have the following lead sentence:

> (3.38)
> Foreign Minister Andrei Gromyko arrived in Vancouver this morning on the Russian cruiser Suvarov for the International Conference on The Law of the Sea.

Story (3.38) defines a state-visit having a top-level representation like this:

```
!STORY: (SEQ SCLAB1)

SCLAB1: (SEQ SCLAB2 SCLAB3 SCLAB4)
TOP: $VIPVISIT
MAINCON: ((<=> ($VIPVISIT VIP HUMO
                          INVITINGSTATE POLITO
                          REASON EVNT1)))

SCLAB2: (SEQ SCLAB5)
TOP: $GOVIP
MAINCON: ((ACTOR HUMO <=> (*PTRANS*) OBJECT HUMO
             TO (*PROX* PART POLITO)
             INST ((ACTOR ORGO <=> (*PTRANS*)
                      OBJECT HUMO
                      TO (*PROX* PART POLITO)))))

SCLAB5:
TOP: $SAIL

SCLAB3:
TOP: $GOALVIP
MAINCON: EVNT1

SCLAB4: UNINSTANTIATED

HUMO: "Gromyko"
ORGO: "Russian Navy"
POLITO: "Canada"
EVNT1: ((<=> ($CONFERENCE MEMBER HUMO)))
```

The summarizer would use this structure to generate the following summary of (3.38): "Gromyko sailed to Canada for a conference." Questions such as "Who went to Canada?" and "How did Gromyko get to Canada?" refer to the "going" segment of $VIPVISIT. These would be answered by reference to the Maincon of $GOVIP: "Gromyko" and "The Russian Navy took him there," respectively. Similarly, the question "Why did he go to Canada?" refers to $GOALVIP, and its Maincon "for a conference."

A second class of Situations that SAM deals with is accident stories. The system currently can do examples of motor-vehicle accidents ($VEHACCIDENT), train wrecks ($TRAINWRECK), plane crashes ($PLANECRASH) and oil spills ($OILSPILL). The global structure of these Situations is the same in each case. Each is a mechanical force Script, describing a mechanical force out of control, the resulting accident, and reactions by appropriate authorities. In each case, the Script starts with an instance of a PTRANS-Script in normal operation. For $OILSPILL we would have:

```
((ACTOR ORGO <=> (*PTRANS*) OBJECT STRUCTO
    FROM "origin"
    TO "destination"
    VIA "route"
    INST ((<=> ($SAIL SHIP STRUCTO)))))
```

ORGO: "shipping line"
STRUCTO: "tanker"

This Conceptualization corresponds to inputs such as "The Liberian freighter, Argo Merchant, bound for Boston, was sailing in Nantucket Straits." Suddenly, the Script is interrupted, the captain loses control of his ship, and "the tanker ran aground:"

((ACTOR STRUCTO <=> (*PROPEL*) OBJECT GEOFEATO))

GEOFEATO: "shoal"

The result of the PROPEL event which is typical of $OILSPILL is:

[The ship broke up]
((ACTOR STRUCTO TOWARD (*UNIT*)) MODE (*NEG*))

Since this is a tanker, we have oil in the water. The Coast Guard comes to the scene, and clean-up squads are dispatched. Perhaps the spill is contained, perhaps it threatens beaches and ocean life. The analogies between $OILSPILL and the other accident Situations should be clear.

The analogy also extends to the third class of Situations possessed by SAM: natural disasters. Again we have a force Script, this time describing a complex force and its characteristic results. Again the appropriate organizations react in their Scripted ways. For example, we might have:

```
[An earthquake struck Rumania]
((ACTOR FORCE0 <=> (*PROPEL*) OBJECT POLIT1))

POLIT1: "Rumania"
FORCE1:
  CLASS (#NATFORCE)
  TYPE (*EARTHQUAKE*)
```

The story might describe a typical result of a large natural phenomenon like this:

```
[1000 people were killed]
((ACTOR GROUP0 TOWARD (*HEALTH* VAL (-10))))

GROUP0:
  CLASS (#GROUP)
  NUMBER (1000)
```

It might also describe the severity of the quake:

```
[The quake measured 7 on the Richter scale]
((<=> ($EARTHQUAKE SEVERITY (7))))
```

At any rate, medical and rescue forces will be sent to the scene:

```
[The Rumanian goverment sent troops to the scene]
((ACTOR ORG0 <=> (*PTRANS*) OBJECT GROUP1
     TO (*PROX* PART LOC0)))

ORG0: "Rumanian government"
GROUP1: "troops"
LOC0: "the scene"
```

For both accidents and disasters, the Situation provides a processing hierarchy, a global story structure, and a scorecard. For $VEHACCIDENT, the Situation first prescribes an instance of the mechanical force Script. Initially, we have expectations for inputs such as:

```
[There was a car accident]
((<=> ($VEHACCIDENT VEHICLE STRUCT0)))

STRUCT0: "a car"

[car out of control]
((<=> ($DRIVE DRIVER HUM0 VEHICLE STRUCT0))
         MODE (*CANNOT*))

HUM0: "someone"

[car hits something]
((ACTOR STRUCT0 <=> (*PROPEL*) OBJECT PHYS0))

PHYS0: "something"
```

Once a crash has been confirmed, the Situation sets up expectations for the activities of appropriate organizations, that is, for service Transactions such as $AMBULANCE and $HOSPITAL. The Situation also defines the connections among its components, depending on which of its Transactions is instantiated. For example, if we read:

(3.39)
John was in a car accident. He was treated at Milford hospital.

the Situation would define the global story structure:

That is, $VEHACCIDENT contains the information that people involved in crashes don't suddenly appear in hospitals, but that an ambulance probably took them there.

Finally, the Situation maintains a scorecard of the activities which are "interesting" in that domain. For $VEHACCIDENT, the Script wants the answers to the questions:

Was anyone killed?
Was anyone hurt, and how badly?
What did the police do?

Patterns corresponding to these answers are part of the Script, and are connected in its causal chains as always. Because they are on the scorecard, however, the Script Applier is always looking for events which have a bearing on them. In (3.39), for example, a statement that John was injured is not explicitly made. The event-patterns corresponding to "went to hospital" and "was treated", however, when instantiated, will cause this inference to be made (through a connecting, default episode in $AMBULANCE).

This completes our discussion of the Situations SAM applies. How the Script Applier actually uses these Situations to organize its processing of stories referring complex knowledge domains such as $VIPVISIT and $VEHACCIDENT is described in Chapter 4. A detailed example of SAM in action in one of these domains is given in Chapter 6.

3.5 Summing Up

Let's go over the main points of the Script management methods we discussed in preceding sections.

The first idea is that our expectations regarding stories that we read come in several different varieties. Having heard about some well-known physical activity, such as going to a bus stop, we form predictions, local in nature, about what we may hear next. Because of this we have no trouble understanding a sentence about a bus

arriving or someone getting on. Hovering above all stories, however, is the global question "Why am I being told this?" We want to know what the "point" of the story is. For a text beginning with a bus ride, we want to know why the ride is being taken, where the rider is going and why. That is, we want to know what the reason for the trip is. In SAM, the highest level expectations about a given knowledge domain, such as trips, are organized into global knowledge structures called Situations. Situations are extensions of simpler Scripts, such as $RESTAURANT and $BUS, which define how these Scripts can fit together. A major reason why SAM is able to process newspaper stories of various kinds is that a Situation can be set up for each domain. Each Situation prescribes a processing order in which the component Scripts in the Situation are tested for applicability, a story representation giving the global structure of the permanent memory representation retained for the story, and a scorecard of events the Situation is particularly interested in finding out about.

Below the Situations, simpler Scripts interconnect in various ways. The chief distinction to be made here is on the basis of setting. Scripts can happen in the same place, or in different places. The class of personal and organizational PTRANS-Scripts define the circumstances under which ACTORs move around. PTRANS-Scripts, in turn, combine with Scripts having a fixed setting, such as $THEATER, to define the global Situation $TRIP.

Scripts which can happen in the same place may interact, or they may not. One case of non-interacting Scripts comprises the different tracks of a commercial Transaction. In a bank, for example, a person may withdraw something from a checking account, then arrange for a car loan. These different manifestations of $BANK, though they happen in the same place, happen in sequence, without interacting. Another class of non-affecting Scripts are ancillary Scripts which the setting of a primary Script can contain. An example is using a phone in a restaurant. Here the activation of $RESTAURANT, say by a Locale Header, activates $PHONE, as well. If $PHONE is instantiated, it goes into the story representation as a unit. If $RESTAURANT is also instantiated, SAM assumes that the two Scripts occur in sequence.

The principle of setting is used in processing stories about events in a place, such as a park or a beach, which does not have one dominant Script, but several, equally likely ones. A reference to the setting brings all these Scripts to the forefront of attention together, so that a reference to any of them can be recognized. A reference to a Script which cannot go on in that setting will not be identified immediately. Like people, SAM's processing is dominated by things which are familiar in a known setting.

Chapter 4
Script Application: the Basic Cycle

4.1 Introduction

Story understanding is a process. Script application models an important part of this process: establishing and using contexts, Scripts, containing stereotyped sequences of events in the world.

The Script Applier attempts to understand a story by introducing the "largest," most inclusive Script it possesses which is initiated by the first Conceptualization in the story. As each input Conceptualization is recognized in the Script, predictions are aroused which the Script Applier uses to find further inputs. This cycle continues until the system receives an input which does not refer to a predicted event. At this point, it again brings in the largest Script which the input initiates, matches roles and props across the Script interfaces, checks the preconditions, if any, for the new Script, and starts matching inputs in the new context. As a knowledge-based, top-down understander, therefore, SAM conforms well to the idea [27] that "the process of understanding a passage consists in finding a schema which will account for it."

In this chapter, we turn from the consideration of the Script as a static knowledge structure, to the application of Scripts in the process of story comprehension. The discussion will focus on the "memory" modules, viz., PP-Memory and the Script Applier itself, which, as the "kernal" of SAM, cooperatively process the Conceptualizations produced by the Analyzer and build a story representation. Script-based text comprehension runs in a cycle which can be divided, roughly, into the following phases. First, the input Conceptualization has to be Internalized. This process includes: replacing conceptual references to PPs with memory tokens of the appropriate class; identifying new tokens as instances of ones which the story has already mentioned, or ones ("permanent" tokens) known to memory from prior experience; marking references to Scripts in the Conceptualization; and decomposing the Conceptualization into its "atomic" components.

Next comes a process of Pattern-Matching of the unit Conceptualizations in the "high-priority" queue of Scripts formed from the active Script contexts and those referred to, either explicitly or implicitly, by the Conceptualization. If a new Script has been referenced, this will be activated, i. e., its predictions about events to follow will be made accessible to the understander. If the Script forms a possible start for a Situation, as discussed in Chapter 3, this structure will be activated also. If a Script is already active when a new one comes in, the old one will be closed or continued depending on the instructions provided by an active Situation, if one is available; or by the "local" constraints defined by the old and new active Scripts.

Once a match has been accepted, the understander must build up an inference chain consisting of the episodes or parts of episodes which can be assumed to have happened between the new Conceptualization and the last one which was read. Finally, on the basis of the new pattern, it must update the search list of predicted events: i. e., load those which are expected to be seen next, and clear those which are temporally to the past of the new pattern.

4.2 Story-Telling Conventions

Before describing these activities, we must recognize that there are several different conventions for telling stories. The simplest type is simply "narrative" mode, in which the events described in the story follow one another in the world in the same order as in the text. At the other extreme of the stories that SAM can handle is the newspaper article, which employs the device of the "lead" sentence to quickly put down what the most interesting events in the story were. The points raised in the lead are then brought up again in subsequent paragraphs, with more detail being introduced each time. An intermediate mode of story-telling uses devices such as "flashbacks" to suddenly shift the setting of a story being told, and connectives such as "but" to signal to the understander that something out of the ordinary is about to happen.

We believe that narrative mode is basic to the others we will discuss. By this we mean that, although writers use other devices to flag the events they want the reader to pay attention to, these are just superimposed on a narrative. Consider, for example, the following lead sentence from the New Haven Register:

> (4.1)
> A New Jersey man was killed Friday evening when the car in which he was riding swerved off Route 69 and struck a tree.

This complicated sentence has, at the top level, the most important fact which the writer wanted to convey: that someone died. The "when"-clause describes the accident that caused the death. In this clause, the sub-Conceptualizations are presented in narrative order: someone loses control of a car, then the car strikes a tree.

Because narrative order occurs so often, the Script Applier's basic processing cycle is designed for Conceptualizations that are presented in this way. This cycle first decomposes a complex Conceptualizion into simple sub-Conceptualizations. These are then input to the recognition process in narrative order. The complex Conceptualization underlying (4.1), for example, would be presented to the Script Applier in the following pieces: "a man was riding in a car," "the car left the road," "it struck a tree," and "a New Jersey man was killed." Modifications to this scheme for handling stories with "but"-Conceptualizations and newspaper articles will be discussed in Section 4.9.

Narrative processing order has important advantages for a story understander. As each event is identified, the system need only predict event-patterns in the near future of the present one. Patterns for past

events can be cleared from memory. Thus, an active Script has a "window" defining the events that are expected to be seen, anchored on one end by the last event from the Script that was read, and on the other by the "reach" of the predictions that the Script has made. As a story progresses, this window moves through the Script from the initial Scenes to the later ones: and only a part of the data base is immediately accessible at any given time. For example, if a story begins "John went into a restaurant and ordered a hamburger," predictions for inputs about his looking for a table, going to one and sitting down, etc., will be cleared from active memory, and replaced by predictions about the "service" part of the Script: the preparation and serving of the food, the patron's eating it, etc. In effect, the window on $RESTAURANT has shifted from the Scenes containing the Script's Headers forward to Scenes in the middle of the Script.

A similar windowing effect is built into more complicated Scripts, such as $VEHACCIDENT. The Headers for the vehicle-accident Script are looking for inputs which either suggest that an accident is about to occur (as in the Conceptualization for "Mary fell asleep at the wheel") or has happened (as in "there was a car crash"). Once one of the Headers has been instantiated, its predictions reach forward to events which are closely associated with a crash. For example, having read "Mary fell asleep at the wheel," the understander is prepared for Conceptualizations for sentences such as "her car left the road," "her car went into oncoming traffic," "her car smashed into something," etc. The system will not predict anything which happens *after* a crash until the crash event is instantiated. This is because the crash is the Maincon of the scene, and we want to avoid inferring Maincons. For example, the story "Mary fell asleep at the wheel. An ambulance took her to the hospital" is peculiar because it leaves out the Maincon.

4.3 Internalizing Conceptualizations

As an illustration of the Script Applier's cycle of pattern-matching, instantiation and prediction, consider SAM's processing of a simple story:

> (4.2)
> John Smith decided to go to a museum. The subway took Smith to Manhattan. He strolled up Fifth Avenue and entered the Metropolitan Museum. He gave the cashier fifty cents. He looked at some sculpture. Then he looked at some paintings. Later he went home.

The first step in understanding a story is internalizing a Conceptualization that the Analyzer has produced, that is, transforming it for use by memory and amalgamating it with existing memory structures. After analysis, the first sentence of (4.2) is:

"John Smith decided to go to a museum"

```
GNO:
   ((ACTOR GN1 <=> (*MBUILD*) TO (*CP* PART GN1)
               MOBJECT GN2))
```

```
GN2:
   ((ACTOR GN1 <=> (*PTRANS*) OBJECT GN1
               TO (*PROX* PART GN3))

GN1:
   (#PERSON PERSNAME (JOHN) SURNAME (SMITH))

GN3:
   (#ORGANIZATION ORGOCC ($MUSEUM) REF (INDEF))
```

(The CD representation used here is discussed in Appendix 1. "GN"-atoms represent "gap nodes," i. e., slots with requirements on what can fill them, in the CD structures built by the Analyzer.) PP-Memory replaces the list structures (#PERSON...) and (#ORGANIZATION...) with tokens having the appropriate properties:

```
"John Smith"   HUMO:
                 CLASS (#PERSON)
                 PERSNAME (JOHN)
                 SURNAME (SMITH)

"a museum"     ORGO:
                 CLASS (#ORGANIZATION)
                 ORGOCC (MUSEUM)
                 REF (INDEF)
```

Then, PP-Memory attempts to identify the tokens just created with tokens already present in its memory: "permanent" tokens for well-known PPs which are always around; and tokens created in the course of reading the story thus far. Assuming that "John Smith" is not a special person known to SAM, PP-Memory can't identify either PP at this point. The REF (INDEF) marker is left by the Analyzer to tell PP-Memory not to look for a referent for the PP among existing entities. (For a discussion of this and other kinds of notes left by ELI, see [25].)

Next, PP-Memory marks the reference to the occupation Script of the museum as a suggested Script to be tested by the Script Applier for applicability.

The tokenized Conceptualization and the various processing structures that PP-Memory has built (basically, the list of new tokens and suggested Scripts) are passed to the Script Applier, and the main phase of Script-based comprehension begins. The Script Applier decomposes the Conceptualization into sub-Conceptualizations containing only a single CD ACT or STATE. The result of this process is a list of simple Conceptualizations preserving the temporal or causal ordering between events .

Simple Conceptualization-patterns are needed to calculate causal-chain results of connections. For example, the causal result of a PTRANS is a change in the location of the OBJECT PTRANSed. A sentence in a natural language can clump simple Conceptualizations together in arbitrarily complex ways. Consider, for example, the CD structure built

by the Analyzer for the following sentence:

(4.3) Mary Jones died Tuesday of head injuries received in a car accident on Sunday.

```
     ((CON GN1 LEADTO GN2))

GN1: ((ACTOR GN5 TOWARD (*PSTATE* VAL (*NEGVAL*)))
           REL GN3))

GN2: ((ACTOR GN6 TOWARD (*HEALTH* VAL (-10))) TIME (TIM2))
TIM2:  ((WEEKDAY TUESDAY))

GN3: ((CON GN4 LEADTO GN1))

GN4: ((<=> ($VEHACCIDENT VEHICLE GN7)) TIME (TIM4))
TIM4:  ((WEEKDAY SUNDAY))

GN5: (#BODYPART TYPE (*HEAD*))
GN6: (#PERSON PERSNAME (MARY) SURNAME (JONES) GENDER (*FEM*))
GN7: (#PHYSOBJ TYPE (*CAR*))
```

The Conceptualization underlying (4.3) says, roughly, that a negative change in the physical state of a bodypart (belonging, by inference, to Mary Jones) caused a terminal change in her state of health; and that the physical change was caused, somehow, by the occurrence of a Script, $VEHACCIDENT. The process of decomposition for (4.3) would produce the ordered list of simple Conceptualizations: "there was a car accident," "a head injury occurred," and "Mary Jones died." In a story containing (4.3), SAM would try to locate each unit event in the indicated order. (In (4.3), there is also "global" Time-Setting data: "accident on Sunday" and "died on Tuesday." This information would be used later to infer that Mary Jones spent two days in the hospital before dying, rather than being pronounced dead at the scene, or on arrival at the hospital.) Putting the simple Conceptualizations into "narrative order" takes advantage of the natural causal/temporal order of the Script. Each new Conceptualization is expected to be found on the basis of predictions set up by earlier inputs. After decomposition, the process of internalization is complete.

4.4 Choosing a Context

In the course of understanding a story, the Script Applier manipulates several processing structures, some of them specific to a particular Script, others related to the story representation as a whole. Information about the state of each Script possessed by the system forms a *Script context* which, if the Script is active, is updated whenever a new Conceptualization is found to fit within the context. Each Script context is defined by: the list of patterns from that Script which are currently in memory; an association-list of tokens bound to Script variables; the name of last pattern matched in the Script; the list of Script episodes currently in memory; the header

for this incarnation of the Script; and a Script-global inference-strength indicator which the Applier uses to flag how probable its inferences appear to be.

The Script Applier also maintains several important global variables which are updated in all the active modules of the system whenever they change, and so are available to affect the process of analysis and internalization. These variables are: the list of all the currently active Scripts: the name of the Script accessed by the last story input; and a pointer to the the most "global" Script currently available. Other Script Applier variables include: (1) the list of Scripts suggested by PP-Memory; (2) the high-priority search list of Scripts where the current Conceptualization is expected to fit, formed from the list of suggested Scripts and currently active Scripts: and (3) the header for the story representation being constructed by the Script Applier for the current text. The story header provides access to the final record of the story, from the most general information about the story to the most specific.

Initially, the record of the Script contexts contains only the Headers for the Scripts in the system, and the variables keeping track of the active contexts are null. The Script Applier receives the tokenized Conceptualization for the first sentence of Example (4.2), and looks at the museum Script, $MUSEUM, because of PP-Memory's suggestion. The "local" context variables for $MUSEUM are loaded into active memory. Note that this means that the contexts which are active do not "see" each other directly, but overwrite each other when a context-shift takes place. Because this is the first input, the high-priority search queue of Scripts is simply the list of suggested Scripts, ($MUSEUM).

At this point the only parts of $MUSEUM which are present in active memory aside from its permanent memory structures (described in Section 2.6) are the Header episodes. The input is matched successively against the Headers. The input matches a Precondition Header of $MUSEUM ("main actor decides/wants to go to a museum"), and $MUSEUM is activated. More predictions from the Script are loaded, and $MUSEUM is added to the list of active Scripts and the Script search list. It is also marked as being the most recent Script accessed by an input.

In addition, the setting of the museum-Transaction has the property that other Scripts can take place there. These Scripts (e. g., $BATHROOM and $RESTAURANT) are added to the search list of Scripts. Next, the Analyzer's handling of lexical and phrasal information is changed. For example, ELI is given definitions of words such as "exhibit" and "painting" which are appropriate for $MUSEUM.

Finally, the information that $MUSEUM is always imbedded in a trip-Situation is used. That is, $MUSEUM is an appropriate part of $GOALTRIP in $TRIP. $TRIP is a Script Situation, ranking above $MUSEUM in the hierarchy of Scripts described in Chapter 3, so it takes control of processing.

This means that $TRIP is marked as the most global Script currently active, and added to the active Script list. Next, the global story representation variable is initialized with the $TRIP story-template,

discussed in Chapter 3. The Script label corresponding to the goal-segment of $TRIP is initialized with the Scriptname $MUSEUM. Since $MUSEUM was initiated via a Precondition Header, we may hear about the going-segment of $TRIP, $GOTRIP. Therefore, $TRIP is marked as having progressed into the $GOTRIP segment, and this segment is initialized with a list of Scripts which are appropriate for moving people around, viz., the personal and Organizational PTRANS Scripts. "John Smith" is assigned the role of main actor in both $MUSEUM and $TRIP.

At this point, a number of partially built processing structures have come into existence. At the top level, the story is a simple-sequential $TRIP. $TRIP is currently in the go-segment. $TRIP in turn consists of a simple-sequential arrangement of $GOTRIP, $GOALTRIP and $RETURNTRIP. The go-segment is as yet unspecified, but is expected to be instantiated by a simple-sequential connection among PTRANS-Transactions. The goal-segment is expected to consist of a sequential instantiation of $MUSEUM and, perhaps, some other Scripts which are appropriate "goal" activities of a trip. The return-segment, if explicitly instantiated, will be a sequential arrangement chosen from among the PTRANS-Scripts. Finally, the instantiation of $MUSEUM will be a Locale-nested arrangement among $MUSEUM, $BATHROOM and $RESTAURANT. The current story representation (stored in the global !STORY) has the following property-list structure:

```
!STORY:   (SEQ SCLAB1)

SCLAB1:   (SEQ SCLAB2 SCLAB3 SCLAB4)
    Scriptname: $TRIP
    Scriptseg: $GOTRIP

SCLAB2:   (SEQ)
    Scriptname: $GOTRIP
    Scriptq: ($BUS $SUBWAY $TRAIN $DRIVE $WALK)

SCLAB3: (SEQ SCLAB5)
    Scriptname: $GOALTRIP
    Scriptq: ($MUSEUM $RESTAURANT $MOVIE $THEATER $VARIETYSTORE)

SCLAB4: (SEQ)
    Scriptname: $RETURNTRIP
    Scriptq: ($BUS $SUBWAY $TRAIN $DRIVE $WALK)

SCLAB5: (NST)
    Scriptname: $MUSEUM
    Scriptq: ($BATHROOM $RESTAURANT)
```

The introduction of $TRIP in this way partly "hides" the activities appropriate to $MUSEUM. The active Script list is currently ($TRIP $GOTRIP $MUSEUM). Since the global search list which guides the selection of contexts is built up from the active-Script global, the patterns appropriate for $TRIP will be looked at first, then those for $GOTRIP, finally those for $MUSEUM. (This assumes that no new Scripts are suggested by PP-Memory.) The Script search list thus has this

(implicit) structure:

```
$TRIP
  $BUS $SUBWAY $TRAIN $DRIVE
  $MUSEUM
         $BATHROOM $RESTAURANT
```

4.5 Pattern-Matching

We said that the first Conceptualization from (4.2) matched a Precondition Header for $MUSEUM. Strictly speaking, the real inference from this sentence is that the decider intends be at a museum at some time. The pattern for this Precondition Header is:

```
(4.4)  ((ACTOR &MGRP <=> (*MBUILD*) TO (*CP* PART &MGRP)
                MOBJECT
                  ((ACTOR &MGRP <=> (*PTRANS*)
                                OBJECT &MGRP
                                TO (*PROX* PART &MORG)))))

       EXPLICIT (&MORG)
```

In pattern (4.4), the variable &MGRP is the main actor of $MUSEUM, that is, a group of people (perhaps only one) who don't have a function in $MUSEUM and who will perform as the "public" in this Script. &MORG is a variable standing for the museum Organization, the "actor" providing this service to the public. The property EXPLICIT on the pattern indicates that the museum-Organization must explicitly appear in the input Conceptualization. This avoids a spurious match on an input such as "John decided to go."

Information about these two Script variables is stored on the property lists of the variables as follows:

```
&MGRP: CLASS (#PERSON #GROUP)
       DUMMY T
       SFUNCTION (*NONE*)

&MORG: CLASS (#ORGANIZATION)
       DUMMY T
       SFUNCTION ($MUSEUM)
```

The "dummy" property is a flag which tells the matcher that this atom is to be bound to the corresponding conceptual cluster in the input. The CLASS marker gives the conceptual PP-classes the input PPs are expected to fall into. The SFUNCTION property indicates the most global function the PP can have in the Script. For &MORG, this is the Script itself. For the "main character," the marker (*NONE*) indicates that the input PP should not be strongly identified with any Script context. If, in fact, the input PP has a strong connection, by function, with some other context, the Script Applier delays activation of a Script, as discussed in Section 4.9.

4.5.1 The Backbone Match

The first phase of matching consists of a comparison of the "constant" parts of the Conceptualization for the sentence with the constant parts of the pattern. The full form of the Conceptualization for the first sentence of (4.2), after PP-Memory has finished with it, is:

(4.5) "John Smith decided to go to a museum"

```
MEM1:
   ((ACTOR HUM1 <=> (*MBUILD*)
               TO (*CP* PART HUM1)
               FROM (NIL)
               INST (NIL)
               MOBJECT MEM2)
            TIME (TIM1) MODE (NIL) MANNER (NIL))

TIM1: ((BEFORE *NOW* X))

MEM2:
   ((ACTOR HUM1 <=> (*PTRANS*) OBJECT HUM1
               TO (*PROX* PART ORG1)
               FROM (NIL)
               INST (NIL))
            TIME (TIM2) MODE (NIL) MANNER (NIL))

TIM2: ((VAL GNO))

HUM1: "John Smith"

ORG1: "a museum"
```

Note, in (4.5), that many "gaps," such as the INSTrument of the PTRANS (how John went), have been left unfilled (NIL), because the sentence did not explicitly refer to them.

The basic rules in the backbone match are: (1) "literal" roles and fillers specified by the pattern must appear in the input; (2) extra roles and fillers in the input are ignored; (3) a dummy must be matched against the same conceptual cluster each time it appears in the pattern; and (4) an empty filler slot in the input matches anything, unless the pattern, using the EXPLICIT property, demands that the filler be explicitly present.

Since the pattern does not contain instrumentals, and doesn't care where John is deciding to leave FROM, these roles are not examined. Since the OBJECT to be PTRANSed is the same as the ACTOR of both the PTRANS and the MBUILD, we avoid a spurious match on a sentence such as "John decided to throw a ball at the museum." Finally, since (4.4) has an EXPLICIT marker on &MORG, the backbone match will fail if the corresponding slot is (NIL). This would abort the match if the sentence were "John decided to go."

4.5.2 Rolefit

The result of a successful backbone match is a list of bindings of candidate PPs to Script variables. The next step in the matching process is checking that the candidates can in fact be instances of the variables. The general process of fitting variables to PPs is called Rolefit. When a variable has been previously bound to a token by Rolefit, the fitting process must be augmented by checks to be sure that the new PP can be an instance of both the variable <u>and</u> the old token. This process is the manifestation in SAM of Reference Specification, and is called Rolemerge.

Rolefit on the results of the match of (4.4) to (4.5) involves an intersection of the conceptual class markers of PP and token, and a check that the <u>function</u> specified by the variable can be performed by the PP. &MGRP can either be a person or group, since we want the Script to handle cases such as "John and Mary went to a museum." "Museum" matches the class specified for &MORG exactly. The point of this initial check on PP-class is to provide a fast failure if the candidate PP obviously cannot fill the specified role. This is possible because, as we argue in Chapter 7, the PP-classes form a contrast set which is intended to model the distinctive ACTORs who do things in the world, and the objects associated with their activities. Note that the class distinctions used are to a certain extent antihierarchical: human and group actors are physical objects, for example, but it usually isn't very helpful to think of them this way in story understanding.

The checking of function is also facilitated by the existence of the PP-classes. One feature of each class is that the indicators of function it may contain are to some extent unique to the class. For people, a title or occupation marker strongly suggests the function the person will have in a context. For organizations, the associated Script (and the sub- and super-Scripts it points to) is the main indicator of function. Physical objects, however, often have a function in more than one Script. A car, for example, can figure as a "vehicle" in a driving situation, or as the "object of sale" in a car-showroom situation. A reference to "a car" in text maps into a conceptual structure of the form (#STRUCTURE TYPE (*CAR*)), where the reference to *CAR* is a shorthand for the cluster of functions and other information that PP-Memory possesses about cars in general:

```
*CAR*: CLASS (#STRUCTURE)
       SCRIPTROLES ((&VEHICLE1 . $DRIVE)
                    (&SALE-OBJ . $AUTOSHOWROOM))
```

Picture Producers fit into Scripts on the basis of function. Because of this, a simple comparison of features may not be sufficient to determine whether a PP can be an instance of a variable. This is the case, for example, for the "obstruction" role from $VEHACCIDENT, and the role "group of visiting dignitaries" in $VIPVISIT. When faced with complex functions such as these, the Script Applier resorts to a form of pattern-directed function invocation [15], to examine the input and the association-list of tentative variable bindings for applicability. In a state-visit context, for example, there is a Header pattern looking for

the arrival of a Very Important Person. If a group of people actually arrived, this pattern would call a function to search among the members of the group to see if one of them qualifies as a VIP. This is the only way to distinguish $VIPVISIT from other manifestations of the trip Situation, such as $VACATION or $BUSINESSTRIP. A characteristic of functions invoked at match time is that they have no real side-effects. If the match fails, all is as before.

4.5.3 Rolemerge

The process of fitting new PPs to Script variables bound to PPs from previous Conceptualizations is called Rolemerge. It is important to note that Rolemerge can, in fact, be implemented in two different places in SAM: in PP-Memory or in the Script Applier itself. The primary reference processor, however, is the Script Applier, since this module may have to act as a "backstop" for PP-Memory in cases where too little information is available in the PPs alone to make the decision. The reason the Applier can figure out the needed reference in these latter cases is that it has the additional information that a predicted pattern involving the variables has been matched (Note 1).

The most important signal in text that Rolemerge will be needed is an instance of a definite determiner (REF (DEF)) attached to a PP by the Analyzer. For example, "he" is mapped into (#PERSON GENDER (*MASC*) REF (INDEF)).

This processing note is first seen by PP-memory, which looks for the referent among the tokens already in existence, those created during the processing of the current Conceptualization, or in its collection of permanent tokens. If the PP is a permanent token, PP-Memory makes the connection and turns off the processing suggestion. The Script Applier then searches for the PP as though the reference were indefinite. Suppose, for example, the second sentence of (4.2) referred to "the BMT" rather than to "the subway." The corresponding PP would be treated as though it were simply "a subway," which, from the point of view of <u>function</u>, it is.

The first example of a definitely determined PP in (4.2) is "the subway" in the second sentence. This is an interesting case, because "the subway" can either refer to the class of PTRANS-Organizations that move people around in certain cities, or the unique one which Smith would use to get where he wants to go. If our memory representation for Smith were more complete, it might be possible to decide which subway is in fact being referenced by knowing where Smith lives and what his destination is. In this case, Smith would be a permanent token. We are

----

1. Versions of the Rolemerge processor are implemented in both modules of SAM. In the "newspaper-processing" arrangement of the system, however, the Applier is the "active" module during the understanding phase. This is why, in the sample story given in Chapter 1.6, Rolemerge is only done by the Applier: PP-Memory's actions are confined to identifying PPs as instances of permanent tokens.

assuming, however, that PP-Memory doesn't know about Smith initially, so the decision about "the subway" has to be left for the Script Applier. As was previously discussed, the possibility of a subway ride was introduced when the $TRIP Situation was activated, so the second sentence of (4.2) would match a prediction from $GOTRIP of the form:

```
((ACTOR &PTRORG <=> (*PTRANS*)
                OBJECT &TRPGRP
                TO (*PROX* PART &PTRDEST)))
```

This match would result in a second call to the Applier Rolemerge procedures, to see if "Smith" can be an instance of "John Smith," previously bound to the global main actor &TRPGRP by the activation of $TRIP. The basic heuristic used by Rolemerge is called "Reference-by-Abstraction." It is assumed that each new reference to a previously bound token will be defined by features which are a subset of the previous ones. In this case, the new PP is defined simply by a SURNAME feature: a subclass of the (CLASS SURNAME PERSNAME GENDER) features that came in with "John Smith." (Rolemerge does not demand that the new features form a proper subset. It would accept an occurrence of "John Smith" in the second sentence, even though this sounds strange.) The Reference-by-Abstraction heuristic would handle cases such as "John" and "he," and also a definite reference to Smith's occupation through an OCCUPATION or TITLE: e. g., "Patrolman John Smith," followed by "the policeman." If the PP were simply "a man," however, Rolemerge would boggle on the REF (INDEF) processing note.

4.6 Making and Unmaking Predictions

Each time a pattern is matched, the "window" on the associated Script must be moved to conform with the Script's expectations about what will happen next. Since, as was discussed in Section 4.2, the Script Applier assumes that stories will be told in narrative format unless something signals otherwise, the prediction process has two distinct phases: (1) clearing patterns from active memory which lie in the past of the presently matched pattern; and (2) bringing in episodes containing the patterns predicted by the present one. The removal of unneeded patterns is accomplished by consulting the permanent memory structures for the Script to determine which scenes currently in memory precede the scene of the currently active pattern, and by nullifying the property lists of patterns belonging to these scenes.

The stories SAM reads refer in general to Transactions and Situations. How are predictions made in these cases? Predictions made in Transactions use the structure of the Script: that is, that it is about a request for some kind of service which an organization provides in return for some sort of payment. Since the Script Applier is interested in determining whether the Transaction can be inferred to have taken place, the first line of predictions after the Headers are about requests for the service the organization provides. Once a request can definitely be assumed, predictions are made about the occurrence of the service and payment. The order in which these things are predicted depends on the details of the Transaction. In each case, if the predicted scene contains turning points, places where Scriptal

interferences can happen, the prediction list for patterns preceding a turning point contain templates looking for the possible outcomes.

Let's consider how the prediction process works in a simple story about a restaurant-Transaction:

> (4.6)
> John walked into a restaurant and ordered a hamburger. The waiter said they didn't have any, so he asked for a hot dog. When the hot dog came, it was burnt. John left the restaurant.

Internalization of the first sentence of (4.6) would result in a suggestion by PP-Memory that $RESTAURANT be tried. The first clause satisfies a Locale Header for this Script, so $RESTAURANT is activated. (Actually, $TRIP would be activated, too, but it is not needed for this story.) The predictions attached to the Header look for events related to a request for service. In the restaurant context, this is the "ordering" scene. The prediction for an ordering event is matched by the second clause. Since the request has been instantiated, the Script Applier now predicts an instantiation of the service (this precedes payment in the "default" track of $RESTAURANT), and scenes which prepare for it. Also, since the act of ordering is a turning point, predictions about possible interference events (the restaurant can't fill the order, or the preparation will take a long time) are added to the search list.

The next sentence from (4.6) instantiates an interference prediction, so patterns for resolving events are predicted. For example, the waiter might ask the patron if he'd like something else. A resolution for the interference appears in the next clause. Since "reordering" is a case of "ordering" (it's an alternative Maincon in this scene), the standard predictions about service and payment are restored to the search list. The clause "When the hot dog came..." is interpreted as a realization of the predicted service event, and so predictions about events to follow (eating and paying) are made. As before, the serving event precedes a turning point in the Script, namely the things that can happen when the customer reacts to the meal. The "default" course of events is that the patron will accept the order and begin to eat. An "interference" reaction is that he will find the food unsatisfactory in some way. This interfering event in fact occurs. Possible resolutions of the interference, e. g., sending the meal back to the kitchen, are added to the prediction list. The most extreme resolution for this interfering event, namely an abort path out of the Script, is realized by the last sentence. The Script Applier instantiates a path out of $RESTAURANT which includes a negative change in the customer's *ANGER* STATE, but not an instance of a paying or tipping event. Since "leaving" is an Exitcon in $RESTAURANT, its instantiation deactivates the Script.

The process of making and clearing predictions in stories processed under Script Situations is very similar to what has been described here. In Story (4.2), the Situation $TRIP is active after the first sentence is read. $TRIP is a linear sequence of going-, goal- and returning-segments, with appropriate Transactions filling the segments. The structure of Story (4.2), for example, is roughly as follows:

The processing of this story will involve, at various points, predictions from its component Scripts: $SUBWAY and $MUSEUM. But there is a predictive component in Situations, over and above what the component Scripts provide. The $TRIP Situation, for example, predicts that only certain Scripts will be used in each of its segments. For example, it initially prescribes that only personal and organizational PTRANS Scripts will be referenced. Yu have to get to the place where something "important" is happening before you can participate in it. Furthermore, a text can refer to a trip in general without filling in too many details. Consider, for example, what it means to "take a train trip," "go on vacation," or "return from Miami." These phrases refer, respectively, to the three segments of $TRIP in general terms.

In SAM, when a story is read with the aid of a Script Situation, the first predictions to be looked at are those provided by the Situation. After the first sentence of (4.2) is processed, the $GOTRIP segment of $TRIP is active, making available the toplevel patterns:

[A trip was taken]
((<=> ($TRIP MAIN &TRPGRP ORIGIN &ORIG DESTINATION &DEST
INSTRUMENTALITY &PTRORG)))

[Someone went somewhere]
((<=> ($GOTRIP MAIN &TRPGRP ORIGIN &ORIG DESTINATION &DEST
INSTRUMENTALITY &PTRORG)))

Recall that "John Smith" is bound to &TRPGRP and "museum" to &DEST. These patterns would handle inputs such as: "The trip/outing took three hours;" and "John journeyed downtown." When the second sentence is read, the variable &PTRORG would be bound to "subway." When Smith enters the museum, $TRIP would move into the predicted $GOALTRIP phase. At this point, a further prediction about a return journey would be made, corresponding to the toplevel pattern:

[Someone returned from somewhere]
((<=> ($RETURNTRIP MAIN &TRPGRP ORIGIN &ORIG DESTINATION &DEST
INSTRUMENTALITY &PTRORG)))

This pattern would be instantiated by the last sentence of (4.2): "Later Smith returned home."

The introduction of $TRIP in this way partly "hides" the activities appropriate to $MUSEUM. The active Script list is currently ($TRIP $GOTRIP $MUSEUM). Since the global search list which guides the selection of contexts is built up from the active-Script global, the patterns appropriate for $TRIP will be looked at first, then those for $GOTRIP, finally those for $MUSEUM. (This assumes that no new Scripts are suggested by PP-Memory.) The Script search list thus has this (implicit) structure:

```
$TRIP
  $BUS $SUBWAY $TRAIN $DRIVE
  $MUSEUM
         $BATHROOM $RESTAURANT
```

4.7 Instantiating Episodes

When a predicted pattern has accepted the current input, the input has been "recognized" within the Script, and a causal chain is constructed which connects the last input to the new one. This chain contains events which can be plausibly inferred to have happened given the inputs that were read.

The Script Applier builds up causal chains by examining the episode structure stored in permanent memory for the active Script. Suppose, for example, we read:

(4.7) Smith went into the BMT. He took the train downtown.

The first sentence of (4.7) instantiates an event from the entering scene, $SUBWAYENTER, of $SUBWAY, and activates the Script. The second sentence realizes the Maincon of $SUBWAY, which is in $SUBWAYRIDE.

To compute a causal connection between events such as these, the Script Applier uses the following rules. First, if the episodes are in the same scene, the Applier traces the causal successors of the earlier event, remaining on the main paths of the episodes, until it encounters the later event. While looking at any given main path event, the Applier will also check on events which are immediate inferences (forward or backward) from it. If the events are in different scenes, the Applier will construct a causal chain having three segments. First is a causal connection between the earlier event and an Exitcon for its scene which has pathvalue "default." Next is a segment consisting of the default paths through scenes which lie between the scenes of the earlier and later events. Finally, there is a connection between a default Entrycon of the later event's scene and the later event. In (4.7), $SUBWAYENTER and $SUBWAYRIDE are adjacent scenes, so the connecting path contains only the first and last segments.

The result of this process is a list of uninstantiated patterns representing a mainpath connection between the events. The Applier takes this list and "realizes" the patterns, one after the other, by replacing occurrences of Script variables in the patterns with the PP-tokens bound to the variables. For example, in the pattern for "patron goes to cashier:"

```
((ACTOR &PATGRP <=> (*PTRANS*) OBJECT &PATGRP
                TO (*PROX* PART &CASHIER)))
```

the variable &PATGRP would be replaced by the token for Smith, say, HUM0.

The above pattern contains a variable &CASHIER, which is not bound to a PP, because the role was not mentioned in (4.7). When this happens, we have a need for a Role-Instantiation inference, in which the Applier asks PP-Memory for a token having properties which are appropriate, in default, for this role. At the time $SUBWAY was activated, PP-Memory was informed about the properties of all the variables having a place in the Script. It uses this information to supply the Instantiator with a token for "cashier" in which the PP's place in the Script is recorded under the SROLES property:

```
HUM1:
 CLASS (#PERSON)
 SROLES (($SUBWAY . &CASHIER))
```

With this token supplied, the realized form of the pattern is:

```
((ACTOR HUMO <=> (*PTRANS*) OBJECT HUMO
         TO (*PROX* PART HUM1)) TIME (TIME5))
```

where TIME5 defines an appropriate temporal relation between this Conceptualization and the other events in the causal chain.

Inference events which are immediate results or enabling conditions of mainpath events are realized at the same time as the main events.

4.8 Changing Contexts

When the second sentence of Story 4.2 is read, $SUBWAY is activated under control of the $GOTRIP segment of $TRIP. As we said in Chapter 3, this means that the bindings for "person(s) taking trip," "conveyance" and "destination" are copied from $GOTRIP to $SUBWAY, and various parts of $SUBWAY are realized.

The instantiation of $SUBWAY up to the point referred to by the Conceptualization for the second sentence has several parts. First, the Preconditions of $SUBWAY are realized. For example, the Script Applier, not having read anything to the contrary, will assume that Smith possesses a token to get through the turnstile, and wants to go somewhere on the subway. The "main character" is known to be "Smith" and the "destination" is known to be "a museum," so these tokens are used directly. PP-Memory uses the defaults supplied by the Script Applier to create a "token." Next, the Script Applier constructs a causal chain from the default Entrycon to the Maincon (which the second sentence of (4.2) instantiates), and realizes this chain as described in the previous section. In this process it obtains tokens from PP-Memory for "cashier," "turnstile," "platform," etc., as it encounters the associated Script variables. The first part of the third sentence instantiates an Exitcon from $SUBWAY, so a path consisting of the default episodes of $SUBWAYRIDE and $SUBWAYEXIT is constructed, and $SUBWAY is closed.

The Conceptualization for the second part of this sentence ("...he entered the Metropolitan museum.") activates $MUSEUM using the Locale Header. At this point the Situation moves into the goal-segment,

$GOALTRIP, because this is the goal activity that was predicted at the beginning of the story. The specification of the reference for "he" in this clause is made on the basis that the main character in a Transaction which is part of a trip is required to be the same as the global main character, &TRPGRP. Activation of $MUSEUM results in the prediction of a possible "admission" event, in which a member of the public pays to get into a museum. This pattern matches the next sentence from Story 4.2: "He gave the cashier fifty cents." Now that the main character is firmly inside the museum Locale, the Script predicts a number of episodes which are appropriate for museum-going. One of the most important of these is the "cyclic" episode in which the patron goes to an exhibit of some sort, and studies the things on display there. The Maincon of this episode matches the Conceptualizations for the next two sentences, where each instance of matching predicts another possible instantiation of the episode.

The last sentence of Story 4.2 fails to match the outstanding predictions in either $GOALTRIP or $MUSEUM. It does, however, fit the prediction associated with the returning-segment of $TRIP. This pattern has as one of its roles the conveyance that the main character used to get back to where he started from. The conveyance is not mentioned in the last sentence, so $RETURNTRIP assumes that the conveyance that was instantiated in $GOTRIP is the one that Smith used to get home.

4.9 Processing Newspaper Stories

Story 4.2 illustrates the processing of stories using Situations to organize the understander's expectations about a large knowledge domain. Understanding of certain kinds of newspaper stories, as we indicated in Chapter 3, is also facilitated by the presence of an appropriate Situation. SAM, for example, has applied Situations for vehicle accidents, state visits, train wrecks and oil spills.

Newspaper stories, however, differ from stories such as (4.2) in that they are typically not told in the same way, even though the knowledge domain they refer to may be describable in terms of a Script. The most important difference is that they do not use narrative mode, that is, events may not follow one another in the story in the same way as they do in time. In this section we describe some of the effects of the way in which newspaper articles are told on SAM's processing. The general conclusion is that the machinery set up for narratives is adequate, with minor changes and extensions, for newspaper articles as well.

The most striking thing about stories from newspapers is the phenonemon of the <u>lead sentence</u>. This, the first sentence of a story, is usually a very complicated one giving the major event that happened, together with time- and place-setting, and other supporting information. A typical lead sentence from $VEHACCIDENT, for example, might be:

(4.8) A Pennsylvania man and wife returning from vacation were killed today in a violent car crash on the Connecticut Turnpike.

Sentence (4.8) announces the most important thing that happened in this instantiation of the Script, the thing that readers want to know first, namely, that a certain couple died. The death is placed in an appropriate setting in time ("today") and in relation to the Script itself ("in a violent car crash"). The Conceptualization concerning $VEHACCIDENT, in turn, is located on a typical road-link, "the Connecticut Turnpike."

Newspaper writers use the lead sentence as a means of quickly communicating the main point of a story. Readers can scan the first sentence and quickly extract the gist of each article. The format of lead sentences is roughly, "When X, Y," where X is a Conceptualization about the Script which was realized, as in (4.8), or a Script Maincon. Having selected the appropriate knowledge domain, the writer describes the most important thing that happened. If someone was killed or hurt, Conceptualization Y will say so. If people are both dead and injured, the details on the dead people are usually given first.

This format for writing lead sentences fits rather well into the decomposition strategy which SAM uses to find the unit Conceptualizations in a complex input. (This process was described in Section 4.3.) In the case of (4.8), for example, the right Script would be found very quickly because the phrase "in a violent car accident," which would be searched for first, corresponds to a Direct Header for $VEHACCIDENT. The match against the Header would also yield information about the vehicle involved ("car accident") and the place ("on the Connecticut Turnpike") of the accident. Having activated this Script, the Script Applier would go on to find the top-level event: "two people died," since this realizes one of the standard predictions that the Script Applier makes when it reads about an accident.

Example 4.8 also illustrates a typical feature of newspaper stories, the use of paraphrases to refer to Picture Producers. Here, the PPs serving in the role of "people killed in the crash" are introduced by residence, rather than by name. This process, which is called "paraphrastic reference," is handled by a simple modification to the Script Applier Rolemerge heuristics. The standard method used by Rolemerge is called "reference by abstraction." It is assumed that each new description of a PP will use a subset of the features named when the PP was first introduced. Having heard about a certain "John Smith," for example, Rolemerge now expects to see PP-tokens containing some or all of the features (FIRSTNAME LASTNAME GENDER), as in "John," "Smith" or "he." (Note 2.) In the case of newspaper stories, Rolemerge will not try to do reference by abstraction until it knows the "tag," i. e., the unique name of the PP. For persons, for example, the tag is an instance from the set (FIRSTNAME LASTNAME). Until it gets a tag, Rolemerge will be satisfied if there are no contradictions between features of a new PP proposed for a variable and the PP already bound. For example, it would accept "the Pennsylvanians" as a reference to the "Pennsylvania man and wife" introduced by (4.8). If the story which (4.8) leads off

----

2. Reference-by-Abstraction assumes that the noun-group descriptions of PPs will be "simple," that is, no additional appositives or relative clauses will appear in later descriptions of the PP. For example, we will not see something like "Smith, who was a boxer."

continued:

> (4.9) John and Mary Gavin, of Morristown, Pa, were pronounced dead on arrival at Milford Hospital.

the Script Applier would now have a tag for the group of people that died, and would be prepared to accept "Mr and Mrs Gavin," "the Gavins" or "both." It would make the identification between "a Pennsylvania man and wife" and "John and Mary Gavin" even if (4.9) omitted the residence specification "Morristown, Pa." This is because the event of a death in $VEHACCIDENT predicts the announcement of this event by an appropriate medical authority.

A final feature of newspaper articles, related to the lead sentence, is the "multi-pass" nature of descriptions of events. The lead sentence introduces the main event of the story, then the story may talk about other things which happened. However, it is quite possible that the writer will later refer again to an event which has already been instantiated, for the purpose of filling in some details which are not appropriate for a first reading. For example, suppose the full story introduced by (4.8) were:

> (4.10)
> A Pennsylvania man and wife were killed today in a violent car accident on the Connecticut Turnpike. John and Mary Gavin, of Morristown, Pa., were pronounced dead on arrival at Milford Hospital. The Gavin's 17-year-old son, Irving, was taken to the Milford Hospital emergency room, where his condition was described as "critical." The Gavin's car went off Interstate 95 and struck a bridge abutment. Both victims died instantaneously.

The first two sentences of (4.10) talk about the people who were killed. The third sentence refers to a third participant in the crash, who was still alive at the time of writing. Suddenly the story jumps back to the Maincon of the Script, the crash itself, and an event which has already been explicitly mentioned is brought up again, with a new detail added.

A mode of processing which established connections between events at the time when each new input is matched would be forced to "backtrack" in the presence of inputs such as the above. For example, the obstruction role in $VEHACCIDENT, which is implicitly introduced by the first sentence of (4.10), would be filled in by default when this input is processed. The pattern which is instantiated would be matched again when the sentence containing "a bridge abutment" is read, and new features would have to be added to the "default" PP. Similarly, the pattern for "someone died," instantiated when the first sentence is processed, would have to be updated when "instantaneously" is added by the last sentence.

To avoid backtracking in cases such as these, the Script Applier delays computing causal chains until all the story inputs have been read. At that time, all the facts about PPs and events have been accumulated, and the instantiation process can proceed without the need

to overwrite something that has already been done. For more information on how the pattern-match and instantiation cycle is implemented in newspaper stories, the reader is referred to the detailed example given in Chapter 6.

Chapter 5
Inferencing in SAM

5.1 Introduction

Chapter 4 described the Script Applier's fundamental process of matching an input against a pattern from one of its Scripts. This activity has three major goals: to identify the context to which an input Conceptualization refers; to move around in an an active Script; and to change Scripts when the one which was active no longer seems appropriate. Since the SAM understander is driven by this recognition process, it is crucial that it be "robust," that is, capable of coping with inputs which, although they are "equivalent" to an expectation in some sense, deviate from an exact match in various ways.

A Script's expectations are encoded as patterns which fit into causal chains. As we explained in Chapter 2, the form the pattern takes is determined by its connections to other patterns in the chain. In $RESTAURANT, for example, the pattern looking for an instance of the waiter going to the patron to take his order has the form:

```
((ACTOR &WAITER <=> (*PTRANS*) OBJECT &WAITER
                    TO (*PROX* PART &PATGRP)))
```

The pattern requires the waiter to be "near" the patron because this is the only way, causally speaking, that the next event in the chain (an MTRANS between waiter and patron) can take place. The literal English realization of this pattern: "The waiter went to the patron," while a possible way of expressing this event, sounds a little strange, as though the waiter were about to ask the patron for some help. Another, "equivalent" way to express this event is "The waiter went to the table." The first-stage match of the pattern to the Conceptualization for the latter sentence will fail, however, because of the discrepancy between the slot-filler "proximity of patron" demanded by the causal chain, and the filler "proximity of table" appearing in the input.

Reconciling the difference between what was expected and what is read requires inferencing. We discuss several classes of discrepancies between "equivalent" inputs and predictions, and a set of inferences for ironing them out, some of which have been incorporated in SAM.

A prediction and a closely related Conceptualization may fail to match for one of two reasons: (1) the backbone of the input may be slightly different from what was expected; or (2), as in the above example. some features of a PP bound to a Script Variable may deviate from the norm. In SAM, a failed match is handled by processes attached directly to the patterns. This way, the machinery for dealing with problems is only used where and when a specific pattern doesn't accept an input for a specific reason.

The discrepancy-resolving process proceeds in the following cycle. First, the backbone match takes place. If the match fails. a discrimination net is applied to decide which inference, if any, should

be made. The result of the inference is a modification of the original pattern which is then fed back into the pattern-matcher, proper. If the initial match succeeds, or if the match against a derived pattern succeeds, it may still be the case that a candidate PP does not conform to the requirements the Script has for the slot. In these cases "PP-fitting" inferences are tried. These call either PP-Memory or Script Applier routines which search special data structures attached to the Script.

5.2 Classes of Inferences

Below we discuss a set of problems presented by certain texts which SAM has read. For each problem, we describe the circumstances under which it arises, and the inference needed to solve it. If a pattern modification is required. the test on the result of the failed initial match is given. If an auxiliary memory call is necessary, we describe the routines which perform the call. In cases where SAM cannot handle the needed inference entirely, we indicate the additional procedures that an extended version of SAM, or a more general understander, would require.

The classes of inferences automated wholly or in part in SAM are:

(1) Rolefit
(2) Rolemerge (Reference Specification)
(3) Causal-Chain Instantiation
(4) Role Instantiation
(5) Immediate-Result Inferences
(6) Locational Inferences
  (a) Transitivity of Proximity
  (b) Enclosure
(7) Movement Inferences
  (a) Personal Possessions
  (b) PTRANS Organizational
(8) Mental-Act Inferences
  (a) Perception/Remembering
  (b) Authority Announcement
(9) Agency Inferences

Of these, the first four, Rolefit, Rolemerge, Causal-Chain Instantiation and Role Instantiation, are built into the basic control structure of the Applier. These processes were described in Chapter 4. Rolefit or Rolemerge is performed on every pattern that is matched. Causal-Chain Instantiation (including Role Instantiation, as required) fills in the events of Script episodes which, though not explicitly mentioned by a story, can reasonably be inferred to have happened. The inferred items are then available for later stages of inferencing.

5.3 Immediate-Result Inferences

A simple, but important kind of inference computes Immediate Results of patterns. Immediate results describe the CD STATE arrived at by the ACTOR or OBJECT participating in the event. For example, if we have:

(5.1) John gave Bill a book.

```
((ACTOR HUMO <=> (*ATRANS*)
          OBJECT PHYSO
          TO HUM1))
```

the immediate result inferred is that "Bill has the book:"

```
((ACTOR PHYSO IS (*POSS* VAL HUM1)))
```

Immediate result STATives of this kind are not stored in the Script, since they can be computed from inputs such as (5.1) if the fillers for the ACTOR, OBJECT and TO slots are known. The practical benefit of this is that the physical size of the Script is reduced considerably.

If an input STATive is compared to an ACT-pattern, a simple test can be tried to see if an immediate result inference is appropriate. The tests and derived patterns for various ACTs are as follows:

Act	Test	Derived Pattern
ATRANS	Is input a stative containing IS (*POSS*)?	((ACTOR &OBJ IS (*POSS* VAL &GRP)))
PTRANS	Is input a stative containing IS (*LOC*)?	((ACTOR &OBJ IS (*LOC* VAL &LOC)))
MTRANS	Is input a stative containing IS (*MLOC*)?	((CON &MOBJ IS (*MLOC* VAL &MLOC)))

Table 5.1
Immediate-Result Inferences

Suppose, for example, a pattern of the (*PTRANS* TO) variety is compared against a stative of the (IS *LOC*) type. Immediate Result generates a new stative pattern from the original one and hands it to the matcher. An actual instance of this occurred in the sentence: "The Albanian party was welcomed at Peking Airport by Foreign Minister Huang," from Story 1.3 (Chapter 1, p. 19). Here, the STATive Conceptualization for "at Peking Airport" would be matched against the pattern looking for a PTRANS-organization taking its passengers (the group of visiting dignitaries) to the organization's terminal (e. g., an airport).

5.4 Mental-Act Inferences

5.4.1 Perception and Remembering

Another set of inferences, analogous to Immediate Result but performed for only "mental" ACTs, is Immediate Enablement. The causal chains in Scripts are built up out of events performed by one or more actors, including ACTs of perception, remembering and communication.

English characteristically refers to the thing perceived or remembered, less often to the *MTRANS*/*MBUILD* ACT itself. That is, the causal chain may contain connecting patterns involving mental ACTs on the part of some roles, but the linguistic input will refer to the mental object contained in the pattern, not to the ACT itself.

The inference is called "enabling" because it is assumed in Scripted situations that the things perceived, remembered, decided or communicated are, in fact, true. There isn't any deception, amnesia or hallucination going on. So, if we read "The doctor pronounced John dead," we infer that John is indeed dead, since this is an immediate enablement for the pronouncement.

A failed match against a "mental-act" pattern causes the MOBJECT to be extracted and used as a pattern, provided a simple test is passed. Suppose we have:

> (5.2) John had some lasagna in a restaurant. The service was good so he left a large tip.

The stative "in a restaurant" in (5.2) would activate $RESTAURANT via an Immediate Result inference operating on the Locale Header. When the Conceptualization for "the service was good" is processed, an active expectation (from the pay/tip scene $PAY) is "Patron remember something about service:"

```
((ACTOR &PATRON <=> (*MTRANS*)
             TO (*CP* PART &PATRON)
             FROM (*LTM* PART &PATRON)
             MOBJECT
             ((CON ((<=>($SERVE))) IS
                             (*APPROPRIATE*
                                   VAL &VAL)))))
```

The test used by the Script Applier to decide whether to try a Perception/Remembering inference when the toplevel match fails is simply to check that the input and the MOBJECT of the pattern are of the same conceptual "type:" that is, that both are either ACTs or STATEs. If this test is passed, the MOBJECT is extracted from the pattern, and the pattern matcher is called again. In the example we are considering, the input ("service was good") and MOBJECT are both STATives, so the test is passed. When the matching process is restarted, the value of "appropriateness" bound to &VAL would be used as the basis of a prediction about the size of the tip. For example, if &VAL were low enough, the tipping event would, in fact, be "unpredicted."

5.4.2 Authority Announcements

A second mental-act inference, which comes up repeatedly in newspaper stories, is the Authority-Announcement inference. If a person, group or organization known to be an "authority" in a context makes a pronouncement, the event referred to is inferred to be "true." For example, suppose we have an oilspill story that includes a sentence "The Coast Guard reported that the oil slick was 100 miles long." The knowledge

that "Coast Guard" is an authority in $OILSPILL would be exploited by the Authority-Announcement Inference.

In SAM, each Script has associated authority figures, for example, "police," "an eyewitness" or "a reliable source." Authority-Announcement differs from the other inferences discussed here in that the input is modified, rather than the pattern, before matching begins. This is because the Script is interested in the events themselves, rather than in some authority's talking about them. The needed modification is made at the top-level of the Script Applier, when decomposition of the input Conceptualization is taking place.

5.5 Locational Inferences

Much of the activity in "scripty" stories concerns actors moving from place to place. Because Scripts are associated so strongly with "place," as was discussed in Chapter 2, the Script Applier keeps track of where its various roles are at any given time in the story. If the role is in fact the main actor, we described in Chapter 2 how the Applier updates the Locale-List attached to the token bound to the main actor each time a PTRANS (either explicit or implicit) occurs. Additionally, roles strongly associated with a certain place in the Script have a direct indication of that place stored as one of their properties. Examples of non-moving roles are the tellers in a bank, the cashiers in the token-booths in a subway, and the people who sell newspapers and magazines in kiosks on city streets. ("Non-moving" is not literally meant, only that the activity of these actors is confined to a small place in the setting.)

5.5.1 Transitivity of Proximity

Because of this information, the Script Applier has the capability of making several kinds of inferences about locations of actors. One of these, called Transitivity-of-Proximity, is concerned with spatial relations in groupings of similar sized PPs. As an example of the need for such an inference, consider:

> (5.3) John sat down in a restaurant. A waiter came over
> to the table.

The first sentence of (5.3) gets John into position to initiate a restaurant-Transaction. We understand that he is sitting on a chair at (an implied) table. The patterns from the $ORDER scene in $RESTAURANT which would be predicted after this sentence is understood would include one for an agent of the organization to approach John to take his order. Note that causal-chain continuity demands that the waiter approach the patron because the predicted MTRANS requires that the participants form a small enough group that verbal communication can take place. But the second sentence of (5.3) in fact maps into a PTRANS of a waiter to the proximity of "the table" which was introduced when $RESTAURANT was initiated, and instantiated when John sat down.

Because of this, the initial match would fail. The features of the patron-PP bound to the main-actor role do not match those of "the table." At this point, the test for Transitivity-of-Proximity would be carried

out: is the pattern of the (*PTRANS* TO *PROX*) variety? The answer is "yes," so the inference process examines the main-actor's Locale-List to see where he is. The last thing that was stored there (in the processing of the first sentence of (5.3)) is the assertion that he is "at a table." The tokens for "the table" that the waiter went to, and "the table" John is near merge properly, so the heuristic "things near the same thing are near each other" is used to infer that the waiter is indeed near the patron.

A slightly different form of the inference would be required to process the story fragment:

(5.4) John walked up Main Street to the Bijou Theater. He went over to the ticket counter.

From the causal-chain viewpoint, the relevant pattern is (patron PTRANS TO cashier), which then enables the ticket-buying transaction. Again, the toplevel backbone match would succeed, but the acceptance of the match would hang up on the disparity of the features of the predicted "cashier," and "the counter" actually read. However, the role &cashier is known to be a "non-moving" one in $THEATER. Stored in the Script is the information that "the ticket counter" is where "the cashier" is. and once again the inference goes through.

The Transitivity-of-Proximity inference, though useful, has two provisos. First, note that the PPs involved should be roughly commensurable in size: from the facts "John left his bike near Ft. Lee" and "Ft. Lee is near Manhattan," we would probably not want to infer that "John's bike is near Manhattan." The Script Applier will, in fact, attempt the inference only if people and "small" physical objects are involved. Also, it is clear that the heuristic can't be applied indefinitely: if A is near B, B is near C, and C near D, it isn't necessarily true that A is near D. The Applier cannot chain transitivity inferences together, so this problem is avoided.

5.5.2 Enclosure

A related locational inference, called Enclosure, keeps track of where actors are in a Script-global context. Many Scripts have well-defined locations where their activities go on, often associated with buildings. For these kinds of Scripts, the Script Applier consults a crude "building-frame" stored in the Script to decide whether an actor has left the associated setting. Suppose, for example, a story starts "John was taken to a hospital." If this sentence, which initiates $HOSPITAL, were followed by "he was treated in the emergency room," the Enclosure Inference would conclude that the treatment was taking place in the hospital, since hospitals have emergency rooms, operating rooms, intensive-care wards, etc.

5.6 Movement Inferences

Another class of inferences is associated with the motion of actors. These inferences are connected with a generalized idea of "possessions." An actor's possessions are those things either permanently or temporarily

controlled by the actor, which share the actor's motion. The two most important classes of entities which move are people and vehicles.

For people, the idea of possessions is clear: these are their clothes, things they may be holding, and the contents of their pockets. The animate and inanimate contents of vehicles share their motion, so there is a kind of possession here as well. If the vehicle belongs to an organization (e. g., a PTRANS-Organization), then there is a sense in which the Organization "possesses" the contents of the vehicle for the duration of the ride. These ideas are used to keep track of the movements of things and people through various Script contexts.

5.6.1 Movement of Personal Possessions

When the main actor role in a Script is bound to a token for a real-world person, the Script Applier has access to the list of the person's possessions: the things explicitly stated as belonging to him. If the Script contains an ATRANS event with the main actor as recipient, the OBJECT ATRANSed (if it is a small PP) is added to this list. The new possession then shares in any PTRANS the person is involved in. For example, suppose we have:

(5.5) John went into the movie theater and asked for a ticket.
The usher took his ticket and showed him to his seat.

The event of "asking for a ticket" is followed by an dual ATRANS event (inferred via Causal-Chain Instantiation), money being given to an implicit cashier, and the patron getting the ticket. This is the ticket that "the usher took." In terms of the implementation, the Script Applier would not ask PP-Memory to create a new ticket-token when "the usher took his ticket" is processed, because one would already exist among the patron's possessions.

Sometimes the possession-adding ATRANS can only be inferred indirectly. Suppose, for example, we read:

(5.6) John picked up a magazine from the kitchen table,
walked into the living room, sat down and began to read.

In processing (5.6), the Script Applier would eventually activate the read-Script, $READ. At this point, it should not create a new token for the thing read, but use the one that appeared in the first Conceptualization. This requires an inference that the GRASP event ("picked up a magazine") implemented an ATRANS of the magazine to John. (This inference is called an Immediate-Enablement Inference in Rieger's inference system, and was incorporated in his Conceptual Memory program [22].)

The current implementation of the Applier pattern-matcher will make this connection provided two conditions are met. First, the PP GRASPed must be small and capable of being moved from place to place. This avoids problems with sentences such as "John grabbed the doorknob." Secondly, the PP must have at least one function (that is, a role in some Script) that is intelligible in the existing context. In (5.6), one function attached to "magazine" is that it can be "printed-matter" in the Script $READ.

This Script is expected (though on a low-level) in the Script Situation $HOME which presumably would be controlling the processing of (5.6). Therefore, the ATRANS would be inferred in this case. On the other hand, the second condition, on functions of objects in context, would prevent "John walked into the kitchen and picked up a rock" from adding a rock to his possessions. Note that we are not saying that in fact the rock won't be carried around. The next sentence might very well read "He went into the living room and threw the rock at his wife." However, the possessional inference should not be made by a Script Applier, since in fact there is no Script for the above.

5.6.2 Conveyance Inferences

A second class of movement inferences is concerned with the organizations whose business is PTRANSing people from place to place. A reference to such a PTRANS-Organization occurs, for example, in the sentence:

(5.7) "John Smith took the BMT to Manhattan"

```
((ACTOR HUM1 <=> (*PTRANS*) OBJECT HUM1
                          TO (*PROX* PART POLIT1)
             INST ((ACTOR ORG1 <=> (*PTRANS*) OBJECT HUM1
                             TO (*PROX* PART POLIT1))))

HUM1: (#PERSON LASTNAME (SMITH) FIRSTNAME (JOHN))
POLIT1: (#POLITY POLNAME (MANHATTAN) POLTYPE (*MUNIC*))
ORG1: (#ORGANIZATION ORGNAME (BMT) ORGOCC ($SUBWAY))
```

Suppose this Conceptualization is be matched against a pattern from the Script Situation $TRIP of the form (&ptrans-org PTRANS &grp TO &dest), with the &ptrans-org being specified by $TRIP as a subway or bus. This pattern would match directly an input such as "the BMT took Smith to Manhattan," which mentions the PTRANS-Organization as the ACTOR in the Conceptualization. It would also match on inputs where the ACTOR of the PTRANS is not specifically mentioned, as in "Smith was taken to Manhattan."

The initial backbone match of the Conceptualization for (5.7) would succeed, but there would be a contradiction between &ptrans-org (CLASS #ORGANIZATION) and "Smith" (CLASS #PERSON). This contradiction would trigger the Conveyance processor, which has several means for reconciling the difference.

The first thing Conveyance tries is to check for the occurrence of an Instrumental Conceptualization. The idea here is that natural-language utterances are more likely to focus on the <u>intended</u> ACT ("Smith went to Manhattan") rather than its Instrument ("Smith went to Manhattan <u>by subway</u>"). If there is an Instrument, Conveyance extracts it from the input Conceptualization and rematches it against the original pattern. This strategy would succeed for (5.7). If there is not an Instrument, as in "Smith went to Manhattan," Conveyance would accept the match, leaving the decision as to what &ptrans-org is to the Role-Instantiation processor.

Another type of Conveyance inference would be needed if the sentence were instead:

(5.8) "A bus took Smith to Manhattan"

```
((ACTOR STRUCT1 <=> (*PTRANS*) OBJECT HUM1
                             TO (*PROX* PART POLIT1)))

HUM1: (#PERSON LASTNAME (SMITH))
POLIT1: (#POLITY POLNAME (MANHATTAN) POLTYPE (*MUNIC*))
STRUCT1: (#STRUCTURE TYPE (*BUS*))
```

Here, the contradiction is between &ptrans-org and "a bus." Conveyance would test to see that "a bus" is a vehicle, and that it functions as one in a PTRANS-Organization Script. Both conditions are met, so the inference goes through. When the match is accepted, the Role-Instantiation processor would specify $BUS as the occupation of the Script variable &ptrans-org.

5.7 Agency Inferences

The class of Agency inferences is illustrated in the following story fragment:

> (5.9) John went to the theater to see a movie. The film was so offensive he decided to leave. The theater refused to refund his money.

Reading this, we understand that it was probably the cashier or manager, speaking for the organization, that actually participated in the MTRANS. Here, the Script Applier makes use of a built-in "chain-of-command" in Transactions, in this case (cashier, manager, theater), to replace the ACTOR in patterns, producing a new pattern for the matcher to try. The various patterns generated would match inputs such as: "the cashier/manager refused...," "the management refused...," or even "they refused..." Which patterns are selected is determined by the conceptual class of the PP in the input ACTOR slot (viz., person, group, organization), so once again the match proceeds in a structured manner.

Chapter 6
A Very Detailed Example

6.1 Introduction

This chapter is for the benefit of those who wish to see the innards of SAM as it reads a newspaper story. We've selected a fairly long story which exercises most of the inference capabilities and data structures described in previous sections. We'll follow the processing of the story from start to finish, giving as much detail as possible about what PP-Memory and the Script Applier are doing. From time to time we'll interrupt the processing log to show some of the data structures the system is using. (As in previous examples, our discussion of ELI and the postprocessing modules will be relatively cursory.)

Our story describes a car accident and what happened afterwards. The story is essentially the same as one that appeared in a local newspaper, the New Haven Register:

> (6.1) Friday evening a car swerved off Route 69. The vehicle struck a tree. The passenger, a New Jersey man, was killed. David Hall, 27, was pronounced dead at the scene by Dr Dana Blauchard, medical examiner. Frank Miller, 32, of 593 Foxon Rd, the driver, was taken to Milford Hospital by Flanagan Ambulance. He was treated and released. No charges were made. Patrolman Robert Onofrio investigated the accident.

By "essentially the same," we mean that the real story and our example differed only in that the real story packed the first three sentences of (6.1) into a complicated lead sentence, as follows:

> (6.2) A New Jersey man was killed Friday evening when the car in which he was riding swerved off Route 69 and struck a tree.

The remaining six sentences of (6.1) are exactly as they appeared in the newspaper. We simplified (6.2) because ELI was not up to analyzing the complicated, nested relative clauses this sentence contains. The Script Applier, however, is capable of handling the Conceptualization for (6.2) just as it stands, using the decomposition techniques described in Section 4.3.

Before starting on the details of the story processing, let's give an overview of what SAM is up to as it reads (6.1). The first sentence of the story, "Friday evening a car swerved off Route 69," initiates the car-accident Script, $VEHACCIDENT, because it suggests the loss of control of a motor vehicle which typically precedes a crash. When the second sentence is read, SAM definitely knows that this Script is applicable, and immediately sets up expectations for Conceptualizations about injury to the persons in the car, and what the appropriate authorities will do as a result.

When SAM reads "a New Jersey man was killed," it makes predictions about a police investigation of the accident, as is routine when a serious injury or a death occurs. It also makes short-range predictions about what ambulance and hospital authorities are likely to do. One short-range prediction, being "pronounced dead," is in fact instantiated by the next sentence. (Another possiblity allowed for in the Script is being pronounced dead "on arrival" at a hospital.) Because this is a car-accident Situation, the Script Applier has retained its highest level expectations about further injuries and police action. It uses its injury prediction to understand the fourth sentence, "Bill Miller was taken to the hospital," because this event customarily follows the (in this case, implicit) event of an injury in a crash. Since the ambulance component of the Script has been accessed, SAM now makes short-range predictions about what usually happens next, namely, activities in a hospital. These predictions are fulfilled by the next input, "he was treated and released," which SAM understands as referring to emergency room treatment in an (implicit) hospital. (Alternatives to "released" in the Script include "sent to operating room," "kept for observation," etc.) Because the injured man was allowed to go, SAM concludes that his injuries were minor.

The next sentence, "the passenger was extricated...", represents a sudden jump in the story, back to the events surrounding the deceased man. Such jumps are typical of the "multi-pass" nature of newspaper reports. SAM handles jumps like this by retaining all the predictions it ever made in the course of reading the story. As new inputs appear, the associated predictions go at the head of the search list, so that the older ones gradually lose priority. On this basis, SAM finally recognizes the event as an instance of an "emergency-service" organization assisting at a motor-vehicle accident. (Another service such an organization could render in the Script would be putting out a fire caused by the crash.)

The high-level prediction about police activity, which has been kept active all this time, is instantiated by the eighth sentence, "No charges were made." Because SAM is interested in what the police may do in accidents, it interprets "charges" as the police department's initiating a prosecution, rather than, say, a bill for services from the hospital. The last sentence, "Patrolman Onofrio investigated the accident," is a continuation of the police activity component of $VEHACCIDENT. This event is causally in the <u>past</u> of the sentence about charges previously read. Therefore, finding it would ordinarily be delayed until that event's short-range predictions are processed. Since "no charges" is an Exitcon from $VEHACCIDENT, there aren't any successors, so "investigated" is found because it is a high-level expectation. If, however, the input had been something like "police charged Miller with reckless driving," finding "investigated" would be delayed until the expectations corresponding to "they gave him a ticket" or "they took him to the station house" were processed.

6.2 Understanding the Story

Story (6.1) is processed by SAM with the help of the Script Situation $VEHACCIDENT. Because this is a newspaper story rather than a simple narrative, SAM understands the story in two distinct phases. First, it tries to recognize all the inputs, then it builds a story representation on the basis of the facts that have been accumulated. As we discussed in Section 4.9, newspaper stories tend to go over events in several passes, adding new details each time. As a practical means of dealing with this, the Script Applier delays instantiating any of its causal-chain patterns until all the inputs have been read. We consider the processes of locating the inputs and building up the story representation in turn.

6.2.1 Finding the Inputs

The version of SAM described here is intended for reading newspaper stories. It has Scripts describing four different knowledge domains which newspapers report on: car accidents, train wrecks, state visits and oil spills. Initially, only the headers from these Scripts, together with the permanent data structures described in Chapter 2, are present in active memory.

The log of a SAM run given below was made under a DECsystem-10 utility program called OPSER, which has facilities for starting up and controlling several jobs, and sending any output from the jobs to a single terminal. Lines beginning with a "!" are OPSER messages indicating the module from which succeeding lines of output came. PARSER is the Conceptual Analyzer, TOK is PP-Memory, and APPLY is the Script Applier. The data structures shown are displayed as they existed at the corresponding point in the processing. The log has been edited slightly for readability.

COMPUTER OUTPUT	COMMENTARY
OPSER TRANSACTION LOG YALE SYSTEM 507B-2	
!(PARSER) Friday evening a car swerved off Route 69. The vehicle struck a tree. The passenger, a New Jersey man, was killed. David Hall, 27, was pronounced dead at the scene by Dr Dana Blauchard, medical examiner. Frank Miller, 32, of 593 Foxon Rd, the driver, was taken to Milford Hospital by Flanagan Ambulance. He was treated and released. The passenger was	First PARSER displays the text to be read, a 9-sentence story about a car crash. Then it turns control over to APPLY...

extricated from the vehicle by the
Branford Fire Department. No charges
were made. Patrolman Robert
Onofrio investigated the accident.

```
!(APPLY)
SCRIPT APPLIER MECHANISM...VERSION 4.1
            ...12 JULY 1976
PROCESSING NEWSPAPER TEXT (TEXT . C1)
AVAILABLE SCRIPTS:
($TRAINWRECK $VIPVISIT $VEHACCIDENT
        $OILSPILL)
APPLIER RUNTIME: 560  APPLIER GCTIME: 0
```

APPLY notes that it has four Scripts available to read this story...

```
FREE: 9615 FULL: 1626
GETTING NEW INPUT
```

...then it instructs PARSER start working on the first sentence.

```
!(PARSER)
Friday evening a car swerved
off Route 69.
```

PARSER displays the first sentence...

```
CONCEPT·
GN7 =
((ACTOR   GN30 <=> (*PTRANS*)
  OBJECT GN30 TO (NIL)
  FROM (*TOPOF* PART
   (#LINK LINKTYPE (*ROAD*)
          LINKNUMBER (69)
          DIRECTION (1)
          ROADTYPE (HIGHWAY)))
INST (NIL))
MODE (MOD1)
TIME (TIM2))
```

and parses it into a Conceptualization about a PTRANS from a "link" of type "highway"...

```
GN30 =
(#STRUCTURE TYPE (*CAR*) REF (INDEF))

PARSING TIME: 31431 GCTIME: 3223
```

...by a "structured" object of type "car."

```
!(TOK)
top level PARSER atom is: GN7
processing PP:
(#STRUCTURE TYPE GN17 REF GN18)
creating new token: STRUCTO
```

TOK looks at this concept, and assigns tokens to

the car

```
processing PP:
(#LINK LINKTYPE GN47 LINKNUMBER
GN48 DIRECTION GN49 ROADTYPE GN50)
creating new token: LINKO
```

and the road

```
Suggesting Script $VEHACCIDENT


top level TOK atom for GN7
       is MEMO
```

Because the car has a possible role in one of the Scripts that APPLY has, TOK suggests that it be tried first.

```
!(APPLY)
NEW INPUT: MEMO
```

At this point, APPLY has been passed the tokenized Conceptualization corresponding to the first sentence. Let's look at some of the data structures SAM is using. First, the Conceptualization itself:

```
    MEMO:
      ((ACTOR STRUCTO <=> (*PTRANS*) OBJECT STRUCTO TO (NIL)
        FROM (*TOPOF* PART LINKO) INST (NIL))
          MODE (MOADO) TIME (TYMEO))
```

This asserts that a certain physical object (STRUCTO) moved itself away from a certain kind of place, a road "link." The time specifier (TYMEO) for the Conceptualization says that the event happened on a certain day of the week (FRIDAY), during a certain part of the day (EVENING), and that all of this happened sometime in the past ("swerved") of the time (*NOW*) when the story is being read:

```
    TYMEO:
      ((WEEKDAY FRIDAY)
       (DAYPART EVENING)
       (BEFORE *NOW* X))
```

The Conceptualization mentions two PPs. The first is a structured physical object, a car. The property-list representation for this object indicates its conceptual class, possible lexical realizations for it in English, Spanish and Mandarin, and a note (REF) that it was referenced indefinitely in the surface string using the slots (SURFSLOTS) of "class" and "type." The token also points to the car's function (as a vehicle), and the possible roles it can fill (SROLES) in various Scripts:

```
    STRUCTO:
     TOKEN (T)
     CLASS (#STRUCTURE)
     ELEX (AUTOMOBILE)
     SLEX (AUTO)
     CLEX (CHE/ TZ)
     REF (INDEF)
     SURFSLOTS
        (CLASS TYPE REF)
     SUPERSET (*VEHICLE*)
     SROLES
     (($VEHACCIDENT . &VEHICLE1)
      ($DRIVE . &VEHICLE1)
      ($AUTOSTORE . &SALEOBJ))
```

The other token refers to a "link," a certain specialized kind of "place" which connects various localities together. This particular link is a "highway," it was described in the surface string using the SURFSLOTS of "class," "number" and "type," and has been assigned a

default direction (1) by PP-Memory:

```
LINKO:
 TOKEN (T)
 CLASS (#LINK)
 LINKTYPE (*ROAD*)
 SURFSLOTS (LINKTYPE CLASS LINKNUMBER)
 ROADTYPE (HIGHWAY)
 DIRECTION (1)
```

Important global variables manipulated by TOK are the list of new tokens created for this Conceptualization (!CDTOKENS), the list of all currently active tokens (!TOKENS), and the suggested Script (!SCRIPTSUGG). The most important variable used by APPLY is the Script context (!SCRPTCNTXT), which records the current state of each Script possessed by SAM. The most important features of a Script's "state" are the patterns currently in memory and the association list of Script variables bound to tokens. Initially, only the headers for each Script are in active memory, and the association lists are null. Here's how these variables look:

<table>
<tr><th>TOK</th><th>APPLY</th></tr>
<tr><td><pre>
!CDTOKENS:
  (STRUCTO LINKO)

!TOKENS:
  (STRUCTO LINKO)

!SCRIPTSUGG:
  ($VEHACCIDENT)
</pre></td><td><pre>
!SCRPTCNTXT:
  (($TRAINWRECK
     (TCRLOC TCR1 TCR2 TCR3))
   ($VIPVISIT
     (VIP1 VAR1 VAR2 VAR3)
   ($VEHACCIDENT
     (ACC1 ACC2 CRA50 CRA51 CRA52
      CRA53 CRA11 CRA12 CRA21 CRA22
      CRA13 CRA2 CRA3 CRA4)
   ($OILSPILL
     (SPL1 SPL3 DSC61))))
</pre></td></tr>
</table>

APPLY begins searching for the first input.

<table>
<tr><td><pre>
FINDING TOPLEVEL CDS: (MEMO)
SEARCHING FOR MEMO IN SCRIPT $VEHACCIDENT
</pre></td><td rowspan="2">APPLY searches for the
input in the Script
suggested by TOK.
It finds a match at pattern
CRA3 in the "crash" scene.</td></tr>
<tr><td><pre>
PATTERN BACKBONE MATCHED AT CRA3
LOCATED AT CRA3
</pre></td></tr>
<tr><td><pre>
MAKING TIME-SETTING ASSERTION
((WEEKDAY FRIDAY) (DAYPART EVENING))
IN SUBSCENES:
($CRASH1 $CRASH2 $CRASH3 $CRASH4
$CRASH5 $TREAT1 $TREAT2 $TREAT3 $INVEST1)
</pre></td><td>APPLY stores the global
time-setting "Friday
evening" in the episodes
which can occur at the
same time as the crash.</td></tr>
<tr><td><pre>
BOUND SCRIPT VARIABLE:
&LINK1 TO LINKO
&VEHICLE1 TO STRUCTO
</pre></td><td>APPLY notes that the
Script variables for road
and vehicle are bound.</td></tr>
</table>

Here is the pattern the input matched, and the Script variables it contained:

```
CRA3:
  ((ACTOR &VEHICLE1 <=> (*PTRANS*) OBJECT &VEHICLE1
                     FROM (*TOPOF* PART &LINK1)))
```

&VEHICLE1: CLASS (#STRUCTURE) DUMMY T SFUNCTION (*VEHICLE*) SUBSET (*CAR* *BUS* *TRUCK* *MOTORCYCLE*) ELEX (VEHICLE)	The Script variable "vehicle" A structured object It is a dummy It has the function "vehicle" Vehicle includes these kinds of PPs.
&LINK1: CLASS (#LINK) DUMMY T SFUNCTION (*ROAD*) ELEX (ROAD)	The Script variable "road"
PREDICT: (CRA4 CRA5)	These are the inputs predicted by CRA3: "vehicle hits obstruction" "vehicle is damaged"

Having made a successful match in $VEHACCIDENT, APPLY activates this Script:

TRACK $VEHACC1 OF $VEHACCIDENT ACTIVATED COMPONENT SCRIPTS: ($CRASH $AMBULANCE $HOSPITAL $POLICE)	"Swerve off a road" suggests that a one-car accident is about to occur. APPLY invokes this track of $VEHACCIDENT, noting that it includes these component Scripts.
SETTING PARSER WORD-SENSES FOR $VEHACCIDENT !(PARSER) (DICTIONARY FILES ARE) (TOK: (VHACC . IDX) (28 345) (BASIC . IDX))	Activating the Script means re-ordering word senses used by PARSER. VHACC is an accident-specific dictionary, BASIC is PARSER's ordinary dictionary.
!(APPLY) SETTING TOK SCRIPT VARIABLE DEFINITIONS FOR $VEHACCIDENT	APPLY notifies TOK of PP-definitions which are specific to vehicle accidents.
!(TOK) (LOADING VARIABLE DEFINITIONS FROM) ACCVAR	

```
!(APPLY)
COMPONENT SCRIPT $CRASH
    OF $VEHACCIDENT ACCESSED
```

Finally, APPLY notes that the input event refers to the "crash" scene of the Script.

```
APPLIER RUNTIME: 12654
     APPLIER GCTIME: 0
FREE: 9020 FULL: 1600
GETTING NEW INPUT
```

APPLY goes for the next Conceptualization.

```
!(PARSER)
The vehicle struck a tree.
```

PARSER gets the next sentence

```
CONCEPT:
GN60 =
((ACTOR   (#STRUCTURE
             FUNCTION (*VEHICLE*)
             REF (DEF))
  <=> (*PROPEL*)
  OBJECT (#PHYSOBJ TYPE (*TREE*)
                   REF (INDEF))
  INST (NIL))
TIME (TIM4)
MODE (MOD2))
```

which corresponds to a PROPEL Conceptualization involving a structured object having function "vehicle."

```
PARSING TIME: 22391 GCTIME: 3146
```

```
!(TOK)
top level PARSER atom is: GN60
```

TOK assigns tokens to the PPs for:

```
processing PP:
(#STRUCTURE FUNCTION GN64 REF GN65)
creating new token: STRUCT1
```

the vehicle

```
processing PP:
(#PHYSOBJ TYPE GN88 REF GN89)
creating new token: PHYSO
```

and the tree.

```
top level TOK atom for GN60 is MEM7
```

```
!(APPLY)
NEW INPUT: MEM7
```

Here is the new concept passed to APPLY and the state of the "deep-memory" modules as the second sentence is processed:

```
MEM7:
  ((ACTOR STRUCT1 <=> (*PROPEL*) OBJECT PHYSO INST (NIL))
       TIME (TYME1) MODE (MOAD1))
```

TOK	APPLY

!CDTOKENS:
  (STRUCT1 PHYSO STRUCTO LINKO)
 	
!TOKENS:
  (STRUCT1 PHYSO)
 	
!SCRIPTSUGG
  NIL
	!SCRPTCNTXT:
	 ((\$VEHACCIDENT
	   (CRA4 CRA5 ACC1 ACC2
	    CRA50 CRA51 CRA52
	    CRA53 CRA11 CRA12
	    CRA21 CRA22 CRA13
	    CRA2 CRA3)
	   ((&LINK1 . LINKO)
	    (&VEHICLE1 . STRUCTO))))

Note how the inputs predicted by the pattern previously matched have been added at the front of the search list for $VEHACCIDENT. Also, since "vehicle" is neutral as to Script, TOK has not suggested a Script for this input. Now APPLY searches for the second concept:

FINDING TOPLEVEL CDS: (MEM7)
SEARCHING FOR MEM7 IN SCRIPT
  $VEHACCIDENT
 	
PATTERN BACKBONE MATCHED AT CRA4
CHECKING GLOBAL TIME ASSERTION
    IN MEM7
	APPLY begins looking for new input in the currently active Script. It finds a backbone match at CRA4, checks whether the crash can happen on the same day (Friday) as the "swerve,"

RUNNING PATTERN-SPECIFIC FUNCTION (PFCRA4)	then checks that the PP bound to the obstruction variable can fill that role.
POSSIBLE REFERENCE FOUND: STRUCT1 IS STRUCTO LOCATED AT CRA4	This works, so APPLY notes that "a car" and "the vehicle" can be the same, and accepts the match.
BOUND SCRIPT VARIABLE: &OBSTRUCTION TO PHYSO	It binds up the variable for "obstruction."

Here is the pattern the second input matched, its Script variables, and the pattern-invoked function that checked on the plausibility of the candidate obstruction. Note that the pattern function first checks to see whether the candidate obstruction (appearing in the pattern-matcher's list of tentative role bindings, NEWPAIRS) is a solid, immovable object, such as a pole or abutment. If this check fails, it retrieves the token attached to the vehicle role (from the Script-global binding list ALIST) and passes vehicle and obstruction to an auxiliary function (not needed in this case), to look at the mass and speed of the vehicle, and the solidity of the obstruction.

```
CRA4:
   ((ACTOR &VEHICLE1 <=> (*PROPEL*) OBJECT &OBSTRUCTION))

&OBSTRUCTION
   CLASS (#PHYSOBJ #STRUCTURE)
   DUMMY T

PATFCT: PFCRA4

(DEFPROP PFCRA4
 (LAMBDA NIL
  (PROG (PAIR OBJ VEH TYP)
     (COND
      ((NULL
           (SETQ PAIR (ASSOC (QUOTE &OBSTRUCTION) NEWPAIRS)))
       (RETURN T)))
     (SETQ OBJ (CDR PAIR))
     (SETQ TYP (GET OBJ (QUOTE TYPE)))
     (COND
      ((INTERSECT TYP (QUOTE (*POLE* *TREE* *BRIDGE* *WALL*)))
       (RETURN T)))
     (SETQ VEH (CDR (ASSOC (QUOTE &VEHICLE1) ALIST)))
     (COND ((CHKOBST VEH OBJ) (RETURN T)) (T (RETURN NIL)))))
EXPR)
```

SAM is now firmly in the vehicle-accident domain. It resumes processing of the story.

```
GETTING FILES FOR $VEHACCIDENT
FTRE1
FTRE2
FTRE3
FTRE4

APPLIER RUNTIME: 10534  APPLIER
              GCTIME: 0
FREE: 6328 FULL: 1472
```

APPLY loads predictions about injuries and treatment into active memory...

```
MERGING TOKENS ((STRUCT1 . STRUCTO))

GETTING NEW INPUT
```

instructs TOK to merge the PPs "car" and "vehicle," and goes for next input.

```
!(TOK)
merging STRUCT1 with STRUCTO
```

TOK carries out this command.

```
!(PARSER)
The passenger, a New Jersey man, was
killed.
```

PARSER analyzes the third sentence...

```
CONCEPT:
GN99 =
((CON (NIL TIME (TIM7))
LEADTO
((ACTOR GN155
      TOWARD (*HEALTH* VAL (-10))
      LEAVING (*HEALTH* VAL (NIL)))
      INST (NIL)
TIME (NIL)
MODE (NIL)))
MODE (MOD3))
```

as an unknown event (NIL) causing a certain person's death...

```
GN155 =
(#PERSON
   DREL
     ((<=> ($DRIVE PASSENGER GN155)))
   REF (DEF)
   GENDER (*MASC*)
   RESIDENCE
     (#POLITY POLTYPE (*US-STATE*)
      POLNAME (NEW/ JERSEY)))

PARSING TIME: 45670 GCTIME: 6502
```

...and that this person was filling the "default" role of passenger in the Script $DRIVE, and lived in New Jersey.

```
!(TOK)
top level PARSER atom is: GN99
processing PP:
(#PERSON DREL GN103 REF GN105
         GENDER GN111
         RESIDENCE GN112 REF GN115)
creating new token: HUMO
```

TOK gets the new concept, assigns tokens to:

the passenger

```
processing PP:
(#POLITY POLTYPE GN113 POLNAME GN114)
creating new token: POLITO
```

New Jersey

```
processing script:
      ($DRIVE PASSENGER GN155)
```

TOK sees the reference to $DRIVE in the input,

```
$DRIVE imbedded in $VEHACCIDENT
SCRIPTROLES addition:
      (PASSENGER $DRIVE HUMO)

top level TOK atom for GN99 is MEM12
```

notes that this is imbedded in $VEHACCIDENT, and passes the role information on to APPLY.

```
!(APPLY)
NEW INPUT: MEM12
```

Here is the state of deep-memory at this point in the reading process: (!SCRIPTROLES is a message from TOK telling APPLY about a Script role attached to a PP in the input.)

```
MEM12:
   ((CON (NIL TIME (TYME2))
         LEADTO
   ((ACTOR HUMO TOWARD (*HEALTH* VAL (-10))
                LEAVING (*HEALTH* VAL (NIL)))
            INC (NIL) TIME (TYME3) MODE (MOAD2)))
    MODE (MOAD3))
```

TOK	APPLY

!SCRIPTROLES:
  ($DRIVE PASSENGER HUMO)
 	
!CDTOKENS:
  (HUMO POLITO)
 	
!TOKENS:
  (HUMO POLITO PHYSO STRUCTO LINKO)
	!SCRPTCNTXT:
	  (($VEHACCIDENT
	    (CRA5 CRA9 TRE1 TRE5
	     TRE6 TRE9 TRE50 TRE55
	     TRE78 CRA4 ACC1 ACC2
	     CRA50 CRA51 CRA52 CRA53
	     CRA11 CRA12 CRA21 CRA22
	     CRA13 CRA2 CRA3)
	    ((&OBSTRUCTION . PHYSO)
	     (&LINK1 . LINKO)
	     (&VEHICLE1 . STRUCTO))))

APPLY begins processing the third sentence:

SETTING SCRIPT ROLE
  (PASSENGER $DRIVE HUMO)
  IN $VEHACCIDENT
BINDING SCRIPT ROLE IN $DRIVE
SCRIPT VARIABLE BOUND
    &PASSGRP1 TO HUMO
	APPLY binds up the role of passenger in $VEHACCIDENT and in the imbedded $DRIVE. It updates the association list..

FINDING TOPLEVEL CDS: (MEM15)
SEARCHING FOR MEM15 IN SCRIPT
    $VEHACCIDENT
PATTERN BACKBONE MATCHED AT TRE1
CHECKING GLOBAL TIME
    ASSERTION IN MEM15
	and begins looking for the new input. It gets a match at TRE1. APPLY checks that this event can happen on Friday.

LOCATED AT TRE1 BOUND SCRIPT VARIABLE: &DEADGRP TO HUMO	APPLY accepts the match, with HUMO in the role of "the group of dead people."
GETTING FILES FOR $VEHACCIDENT FINV1 APPLIER RUNTIME: 7953 APPLIER GCTIME: 0 FREE: 5757 FULL: 1421 GETTING NEW INPUT	APPLY predicts a police investigation, then goes for the next input.

```
!(PARSER)
David Hall, 27, was pronounced dead
at the scene by Dr Dana Blauchard,
medical examiner.
```

PARSER analyzes the next sentence...

```
CONCEPT:
GN177 =
((ACTOR   GN222 <=> (*MTRANS*)
MOBJECT
((ACTOR   (#PERSON GENDER (*MASC*)
           FIRSTNAME (DAVID)
           LASTNAME (HALL) AGE (27))
   IS (*HEALTH* VAL (-10)))
   TIME GN238)
FROM (*CP* PART GN222)
TO (*CP* PART (NIL))
INST ((ACTOR GN222 <=> (*SPEAK*))
TIME GN238 MODE GN239))
TIME GN238 MODE GN239)
```

.. as a communication (MTRANS) event in which the state of a person's being dead is announced...

```
GN222 =
(#PERSON OCCUPATION (*DOCTOR*)
   FIRSTNAME (DANA)
   LASTNAME (BLAUCHARD))

PARSING TIME: 61595 GCTIME: 10014
```

.. by a doctor named Blauchard.

```
!(TOK)
top level PARSER atom is: GN177
```

TOK makes tokens for

```
processing PP:
(#PERSON OCCUPATION GN257
FIRSTNAME GN258 LASTNAME GN259)
creating new token: HUM1
```

the doctor

```
processing PP:
(#PERSON GENDER GN181
         FIRSTNAME GN182
         LASTNAME GN183 AGE GN188)
creating new token: HUM2
```

David Hall

```
processing PP:
(#LOCALE REF GN254)
creating new token: LOCO
top level TOK atom for GN177 is MEM26
```

"the scene"

```
!(APPLY)
NEW INPUT: MEM26
```

When the third sentence is received, the state of the system is as shown below. Note the setting-Conceptualization "at the scene" (MEM31) which is imbedded in the "pronounced dead" concept. Note also that HUMO

is serving simultaneously in the roles of "dead person" and "passenger" in this story.

```
MEM26:
   ((ACTOR HUM1 <=> (*MTRANS*)
               MOBJECT
               ((ACTOR HUM2 IS (*HEALTH* VAL (-10)))
                 TIME (TYME4))
               FROM (*CP* PART HUM1)
               TO (*CP* PART (NIL))
               INST
                 ((ACTOR HUM1 <=> (*SPEAK*))
                     TIME (TYME5) MODE (MOAD4)))
             TIME (TYME6) MODE (MOAD5))

TYME6:
   ((BEFORE *NOW* X)
    (WHEN MEM31))

MEM31:
    ((ACTOR HUM1 IS
        (*LOC* VAL (*PROX* PART LOCO))))
```

TOK

```
!CDTOKENS:
  (HUM1 HUM2 LOCO)

!TOKENS:
  (HUM1 HUM2 LOCO HUMO POLITO
      PHYSO STRUCTO LINKO)
```

APPLY

```
!SCRPTCNTXT:
  (($VEHACCIDENT
    (TRE1 TRE5 TRE6 TRE9
     TRE11 TRE50 TRE55
     TRE78 CRA4 ACC1 ACC2
     CRA50  CRA51 CRA52
     CRA53 CRA11 CRA12 CRA21
     CRA22 CRA13 CRA2 CRA3)
    ((&DEADGRP . HUMO)
     (&PASSGRP1 . HUMO)
     (&OBSTRUCTION . PHYSO)
     (&LINK1 . LINKO)
     (&VEHICLE1 . STRUCTO))))
```

APPLY will attempt to recognize the new input by decomposing it into its "units" and finding them in the temporal order indicated.

```
INPUT HAS COMPLEX TIME IMBEDDINGS·
((WHEN MEM31))
FINDING IMBEDDED CDS: (MEM31)
SEARCHING FOR MEM31 IN SCRIPT
     $VEHACCIDENT
```

APPLY notes the concept in the time atom of the top-level concept, and looks for it first.

```
TRYING INFERENCE TYPE IMRES ON TRE5
```

It attempts an immediate result inference on TRE5, which is looking for a doctor to come to the scene of the accident. This works.

```
PATTERN BACKBONE MATCHED
      ON DERIVED PATTERN:
((ACTOR &MEDIC1 IS (*LOC*
       VAL (*PROX* PART &ACCLOC))))
SUCCESSFUL MATCH ON DERIVED PATTERN
LOCATED AT TRE5
BOUND SCRIPT VARIABLE:
&ACCLOC TO LOC0
&MEDIC1 TO HUM1
```

APPLY binds up variables for
"accident-location"
"doctor at the scene"

```
COMPONENT SCRIPT $AMBULANCE OF
     $VEHACCIDENT ACCESSED
```

APPLY notes that the imbedded concept refers to $AMBULANCE...

```
FINDING TOPLEVEL CDS: (MEM26)
SEARCHING FOR MEM26 IN SCRIPT
             $VEHACCIDENT
PATTERN BACKBONE MATCHED AT TRE9
```

APPLY begins looking for the top concept, and locates it at TRE9.

```
CHECKING GLOBAL TIME
             ASSERTION IN MEM26
POSSIBLE REFERENCE FOUND:
HUM2 IS HUM0
LOCATED AT TRE9
```

APPLY concludes that "Hall" must be the "New Jersey man."

```
APPLIER RUNTIME: 12123  APPLIER
             GCTIME: 0
FREE: 5284 FULL: 1388
MERGING TOKENS ((HUM2 . HUM0))
GETTING NEW INPUT
```

APPLY goes for the next input.

```
!(TOK)
merging HUM2 with HUM0
```

TOK combines the properties of these two tokens.

```
!(PARSER)
Frank Miller, 32, of 593 Foxon Rd,
the driver, was taken to Milford
Hospital by Flanagan Ambulance.
```

PARSER analyzes the next sentence as...

```
CONCEPT:
GN272 =
((ACTOR   (#ORGANIZATION
           ORGOCC ($AMBULANCE)
           ORGNAME (FLANAGAN))
<=> (*PTRANS*) OBJECT GN330
TO (*PROX* PART
(#ORGANIZATION ORGOCC ($HOSPITAL)
   ORGNAME (MILFORD)))
FROM (NIL))
```

...an ambulance organization transporting (PTRANS) a person to a hospital organization.

```
TIME (TIM13)
MODE (MOD5))
```

```
GN330 =
(#PERSON GENDER (*MASC*)
  FIRSTNAME (FRANK)
  LASTNAME (MILLER)
  AGE (32)
  RESIDENCE
  (#LOCALE STREETNUMBER (593)
    STREETNAME (FOXON/ ROAD)
    STREETTYPE (ROAD))
  DREL ((<=> ($DRIVE DRIVER GN330)))
  REF (DEF))
```

This person is "Frank Miller."

```
PARSING TIME: 70370 GCTIME: 10216
```

```
!(TOK)
top level PARSER atom is: GN272
```

TOK assigns tokens as usual.

```
processing PP:
(#ORGANIZATION ORGOCC GN351
               ORGNAME GN353)
creating new token: ORG0

processing PP:
(#PERSON GENDER GN276
         FIRSTNAME GN277
         LASTNAME GN278 AGE GN283
         RESIDENCE GN285
         DREL GN291 REF GN293)
creating new token: HUM3

processing PP:
(#LOCALE STREETNUMBER GN286
         STREETNAME GN287
         STREETTYPE GN288)
creating new token: LOC1

processing script:
    ($DRIVE DRIVER GN330)
SCRIPTROLES addition:
    (DRIVER $DRIVE HUM3)
```

TOK notes that Miller is playing the part of "driver."

```
processing PP:
(#ORGANIZATION ORGOCC GN344
               ORGNAME GN346)
creating new token: ORG1

top level TOK atom for
     GN272 is MEM44
```

```
!(APPLY)
NEW INPUT: MEM44

SETTING SCRIPT ROLE
      (DRIVER $DRIVE HUM3)
       IN $VEHACCIDENT
BINDING SCRIPT ROLE IN $DRIVE
SCRIPT VARIABLE BOUND
      &DRIVER1 TO HUM3
```

APPLY uses this information to bind roles in $DRIVE and $VEHACCIDENT.

```
FINDING TOPLEVEL CDS: (MEM44)
SEARCHING FOR MEM44 IN
       SCRIPT $VEHACCIDENT
PATTERN BACKBONE MATCHED AT TRE55
CHECKING GLOBAL TIME ASSERTION
       IN MEM44
LOCATED AT TRE55
```

APPLY finds the new input at TRE55

```
BOUND SCRIPT VARIABLE:
&HOSPORG TO ORG1
&HURTGRP TO HUM3
&AMBORG TO ORGO
```

APPLY binds the roles "hospital". "group of hurt people" and "ambulance."

```
APPLIER RUNTIME: 11895  APPLIER
         GCTIME: 0
FREE: 4961 FULL: 1365
GETTING NEW INPUT
```

APPLY asks for next input.

```
!(PARSER)
He was treated and released.
```

This sentence is analyzed as two separate concepts.

```
CONCEPT:
GN362 =
((CON ((ACTOR GN408
($TREATMENT
   DOCTOR GN408
   PATIENT GN415))
TIME (TIM16) MODE (MOD6))
```

First, the occurrence of the "treatment" Script (a sense of "treated" specific to accidents),

```
LEADTO
((ACTOR GN415
      TOWARD (*PSTATE* VAL (3))
      LEAVING (*PSTATE* VAL (NIL)))
INC (NIL)
TIME (NIL)
MODE (NIL)))
MODE (NIL))
```

causing an improvement in someone's state of physical health.

```
GN415 =
(#PERSON GENDER (*MASC*) REF (DEF))
```

This is "he",

```
GN408 =
(NIL)
```

and the unnamed person who did the treating.

```
CONCEPT:
GN441 =
((ACTOR   GN450 <=> (*MTRANS*)
MOBJECT
((ACTOR   GN462 <=> (*PTRANS*)
   OBJECT GN462 TO (NIL)
   FROM (NIL) INST (NIL))
   MODE (MOD8) TIME (TIM19))
TO (*CP* PART GN462)
FROM (*CP* PART GN450)
INST
((ACTOR GN450 <=> (*SPEAK*))
    TIME (NIL) MODE (NIL)))
MODE (MOD7) TIME (TIM18))
```

The second concept refers to someone's telling someone else that "he" can leave from somewhere.

```
GN450 =
(NIL)
```

This is the communicator

```
GN462 =
(#PERSON GENDER (*MASC*) REF (DEF))
```

This is the "he" who can leave.

```
PARSING TIME: 62726 GCTIME: 7184
```

```
!(TOK)
top level PARSER atom is: GN362
processing PP:
(#PERSON GENDER GN366 REF GN367)
creating new token: HUM4
top level TOK atom for GN362
          is MEM52
```

TOK processes "he was treated"...

```
!(APPLY)
NEW INPUT: MEM52

FINDING TOPLEVEL CDS: (MEM53 MEM59)
SEARCHING FOR MEM53 IN SCRIPT
          $VEHACCIDENT
```

APPLY breaks this up into "there was treatment" and "someone improved in physical state."

```
PATTERN BACKBONE MATCHED AT TRE57
CHECKING GLOBAL TIME ASSERTION
          IN MEM53
POSSIBLE REFERENCE FOUND:
HUM4 IS HUM3
LOCATED AT TRE57
```

It finds "treatment" at TRE57, noting that "he" must be the person that was hurt.

```
COMPONENT SCRIPT $HOSPITAL OF
       $VEHACCIDENT ACCESSED
```

The $HOSPITAL part of the accident Script is now active.

```
SEARCHING FOR MEM59 IN SCRIPT
       $VEHACCIDENT
```

```
PATTERN BACKBONE MATCHED AT TRE58
CHECKING GLOBAL TIME ASSERTION
       IN MEM59
RUNNING PATTERN-SPECIFIC
     FUNCTION (PFTRE58)
POSSIBLE REFERENCE FOUND:
     HUM4 IS HUM3
LOCATED AT TRE58
```

APPLY finds "physical state improved," and checks on the size of the improvement with a pattern function.

APPLY identifies "he," as before,

```
APPLIER RUNTIME: 16119
      APPLIER GCTIME: 3992
FREE: 4673 FULL: 1343
MERGING TOKENS ((HUM4 . HUM3))
GETTING NEW INPUT
```

and sends for another input.

```
!(TOK)
merging HUM4 with HUM3
```

TOK gets rid of "he",

```
top level PARSER atom is: GN441
processing PP:
(#PERSON GENDER GN366 REF GN367)
creating new token: HUM5
top level TOK atom for GN441
            is MEM68
```

and processes "he was released."

```
!(APPLY)
FINDING TOPLEVEL CDS: (MEM68)
SEARCHING FOR MEM68 IN SCRIPT
            $VEHACCIDENT
PATTERN BACKBONE MATCHED AT TRE59
CHECKING GLOBAL TIME ASSERTION
            IN MEM68
```

APPLY searches for new input, and finds it at TRE59.

```
POSSIBLE REFERENCE FOUND:
            HUM5 IS HUM3
LOCATED AT TRE59
```

Again "he" is identified, since people who have been treated may leave the hospital.

```
RUNNING PATTERN FUNCTION (RFTRE59)

BINDING VARIABLE &NVAL2 TO -3
```

Since the patient has been allowed to go after being treated, APPLY can make an inference about how badly he was hurt. This was not mentioned, so a default value of "slightly injured" is assumed.

The inference about how badly the person was hurt is made by a function invoked when the pattern for the patient's leaving the hospital is instantiated. This function essentially asks the following questions. Has a "value" for the hurt person's injury been seen? (That is, does a value on his HEALTH scale appear in the binding list (ALIST)?) If not, assume "slight" injury, since this episode was about emergency room treatment.

```
(DEFPROP RFTRE59
    (LAMBDA NIL
     (COND
      ((ASSOC (QUOTE &NVAL2) ALIST) NIL)
       (T (TERPRI NIL)
          (PRINTSTR (QUOTE "BINDING VARIABLE
                 &NVAL2 TO -3"))
          (SETQ ALIST
          (APPEND (LIST (CONS @&NVAL2) -3)) ALIST))
          (SAVESCRIPTCNTXT))))
   EXPR)
```

```
APPLIER RUNTIME: 10581  APPLIER
         GCTIME: 0
FREE: 4406 FULL: 1319
MERGING TOKENS ((HUM5 . HUM3))
GETTING NEW INPUT
```

APPLY asks for another input.

```
!(TOK)
merging HUM5 with HUM3
```

Again, "he" is discarded.

```
!(PARSER)
The passenger was extricated from
the vehicle by the Branford Fire

CONCEPT:
GN493 =
((ACTOR    (#ORGANIZATION
             ORGOCC ($FIREDEPT)
             ORGNAME (BRANFORD)
             REF (DEF))
     <=> (*PTRANS*) OBJECT GN532
     FROM (*INSIDE* PART
       (#STRUCTURE
           FUNCTION (*VEHICLE*)
           REF (DEF))))
     MODE (MOD9) TIME (TIM22))
```

PARSER interprets this sentence, in the accident context, as an instance of an organization removing a person from inside a car.

```
GN532 =
(#PERSON DREL ((<=>
  ($DRIVE PASSENGER GN532)))
         REF (DEF))

PARSING TIME: 44418 GCTIME: 3439
```

The person is "the passenger."

```
!(TOK)
top level PARSER atom is: GN493
processing PP:
(#ORGANIZATION ORGOCC GN563
               ORGNAME GN565
               REF GN566)
creating new token: ORG2
```

processing PP:
(#PERSON DREL GN497 REF GN499)
creating new token: HUM6
processing script:
    ($DRIVE PASSENGER GN532)
SCRIPTROLES addition:
    (PASSENGER $DRIVE HUM6)
processing PP:
(#STRUCTURE FUNCTION GN554 REF GN555)
creating new token: STRUCT2
top level TOK atom for GN493 is MEM86
 	
!(APPLY)
 	
NEW INPUT: MEM86
	TOK sees the reference to the "passenger."

The new sentence represents a "jump" in the story, since we have been reading about the injured man, and this returns to the man who died. Predictions about an emergency squad assisting police and ambulance after an accident were made when SAM read about the crash. These predictions, however, were not immediately fulfilled. As a result, although they are still in the search list, the immediate predictions associated with events instantiated since then (e. g., ambulance and treatment) will be looked at first. Here's the state of "deep" memory as this Conceptualization is processed. Note how far down the search list the relevant pattern, TRE11, is.

TOK	APPLY
!CDTOKENS: (ORG2 HUM6 STRUCT2) !TOKENS: (ORG2 HUM6 STRUCT2 HUM3 ORGO ORG1 LOC1 HUM1 LOCO HUMO POLITO PHYSO STRUCTO LINKO)	!SCRPTCNTXT: (($VEHACCIDENT (TRE62 TRE59 TRE58 TRE56 TRE57 TRE6 TRE7 TRE9 TRE11 TRE51 TRE52 TRE54 TRE55 TRE2 TRE4 TRE5 INV2 INV3 CRA5 CRA9 TRE1 TRE50 TRE78 CRA4 ACC1 ACC2 CRA50 CRA51 CRA52 CRA53 CRA11 CRA12 CRA21 CRA22 CRA13 CRA2 CRA3)))

APPLY starts on the new Conceptualization.

SETTING SCRIPT ROLE
  (PASSENGER $DRIVE HUM6)
  IN $VEHACCIDENT
BINDING SCRIPT ROLE IN $DRIVE
REFERENCE SPECIFIED: HUM6 IS HUMO
	APPLY picks up on the processing note about "the passenger," but notes that this role is already bound.

FINDING TOPLEVEL CDS: (MEM86) SEARCHING FOR MEM86 IN SCRIPT $VEHACCIDENT	APPLY searches for the new input, finally finding it at TRE11.

```
PATTERN BACKBONE MATCHED AT TRE11
CHECKING GLOBAL TIME ASSERTION
  IN MEM86
POSSIBLE REFERENCE FOUND:
STRUCT2 IS STRUCTO
HUM6 IS HUMO
LOCATED AT TRE11
```

APPLY notes the reoccurrence of "the vehicle" and "the passenger"

```
BOUND SCRIPT VARIABLE:
&ORG1 TO ORG2
```

and binds up the role for "emergency service organization."

```
COMPONENT SCRIPT $AMBULANCE OF
    $VEHACCIDENT ACCESSED
```

APPLY notes it is back in the $AMBULANCE part of $VEHACCIDENT.

```
APPLIER RUNTIME: 15381
    APPLIER GCTIME: 3932
FREE: 4175 FULL: 1299
MERGING TOKENS
   ((HUM6 . HUMO)
    (STRUCT2 . STRUCTO))
GETTING NEW INPUT
```

```
!(TOK)
merging HUM6 with HUMO
merging STRUCT2 with STRUCTO
```

TOK abolishes the redundant tokens.

```
!(PARSER)
No charges were made.
```

Because this is an accident story, PARSER analyzes "charges" as referring to the prosecution Script.

```
CONCEPT:
GN575 =
((ACTOR   GN582 <=>
($PROSECUTION CHARGER GN582
              PROSOBJ (NIL)
              CHARGEE (NIL)))
TIME (TIM23) MODE (MOD10))
```

There is no reason for a charge in this case, and no one to be charged,

```
GN582 =
(NIL)
```

The entity that didn't make a charge, however, is unknown.

```
PARSING TIME: 27284 GCTIME: 3767
```

```
!(TOK)
top level PARSER atom is: GN575
top level TOK atom for GN575 is MEM93
```

TOK processes this input.

```
!(APPLY)
NEW INPUT: MEM93
```

The MODE atom in the Conceptualization received by APPLY specifies that a prosecution was not initiated. The relevant pattern, INV3, was predicted when "a man died" was read. Because inputs referring to the injured man were subsequently seen, this prediction was pushed back in the search list. INV3 has the form "police decide not to start prosecution:"

The Input:

```
MEM93:
((ACTOR (NIL) <=>
      ($PROSECUTION CHARGER (NIL)
                    PROSOBJ (NIL)
                    CHARGEE (NIL)))
TIME (TYME14) MODE (MOAD14))

MOAD14:
(*TS* *NEG*)
```

The "Equivalent" Pattern:

```
INV3:
((ACTOR &POLORG <=> (*MBUILD*)
      MOBJECT INV4
      TO (*CP* PART &POLORG)))
```

This pattern is looking for "police decided not to prosecute".

```
INV4:
((<=>($PROSECUTION
        CHARGER &POLORG
        CHARGEE &CHARGEE
        PROSOBJ &PROSOBJ))
 MODE (MMODE3))
```

Here is "no prosecution."

Recognizing this input will require a mental-ACT inference on APPLY's part:

FINDING TOPLEVEL CDS: (MEM93) SEARCHING FOR MEM93 IN SCRIPT $VEHACCIDENT	APPLY searches for the new Conceptualization.
TRYING INFERENCE TYPE MTACT ON INV3 PATTERN BACKBONE MATCHED AT INV4	Eventually it gets to INV3. The top-level match fails but there is a mental-ACT inference available. APPLY extracts the MOBJECT of INV3 and redoes the match.
CHECKING GLOBAL TIME ASSERTION IN MEM93 SUCCESSFUL MATCH ON DERIVED PATTERN LOCATED AT INV3	This works.

```
COMPONENT SCRIPT $POLICE OF
   $VEHACCIDENT ACCESSED
```

Now we are in the police-investigation part of $VEHACCIDENT.

```
APPLIER RUNTIME: 16704
   APPLIER GCTIME: 4001
FREE: 3987 FULL: 1283
GETTING NEW INPUT


!(PARSER)
Patrolman Robert Onofrio investigated
the accident.
```

PARSER interprets this as a policeman executing his investigation Script.

```
CONCEPT:
GN637 =
((ACTOR   GN654 <=>
($INVESTIGATION   MAIN (NIL)
   INVESTIGATOR GN654
   INVOBJECT
     ((<=> ($VEHACCIDENT))
     TIME (NIL) REF (DEF))))
TIME (TIM27)
MODE (MOD11))
```

PARSER has been told that this is a motor-vehicle accident.

```
GN654 =
(#PERSON OCCUPATION (*POLICEMAN*)
FIRSTNAME (ROBERT)
LASTNAME (ONOFRIO))

PARSING TIME: 27478 GCTIME: 3434

!(TOK)
top level PARSER atom is: GN637
processing PP:
(#PERSON OCCUPATION GN641
         FIRSTNAME GN642
         LASTNAME GN643)
creating new token: HUM7
processing script: ($VEHACCIDENT)
top level TOK atom for GN637 is MEM100


!(APPLY)
NEW INPUT: MEM100
FINDING TOPLEVEL CDS: (MEM100)
```

APPLY searches for the last sentence,

```
SEARCHING FOR MEM100
     IN SCRIPT $VEHACCIDENT
PATTERN BACKBONE MATCHED AT INV2
CHECKING GLOBAL TIME ASSERTION
     IN MEM100
RUNNING PATTERN-SPECIFIC
    FUNCTION (PFINV2)
```

getting an initial match at INV2. It checks to see that the reason for the investigation (here, the accident) makes sense.

```
LOCATED AT INV2
```

BOUND SCRIPT VARIABLE: &INVEVNT TO MEM103 &POLSTAFF TO HUM7	The last input has now been absorbed.
APPLIER RUNTIME: 18080 APPLIER GCTIME: 3972 FREE: 3791 FULL: 1255 GETTING NEW INPUT	

6.2.2 Building the Story Representation

At this point, all the story inputs have been exhausted. For an ordinary story in simple narrative format, APPLY would have been linking each input event into the story representation as it was recognized. Newspaper stories, however, do not use narrative mode, and may refer to the same event several times in the course of a story. In our example, we read about a man who was killed, then some things about a man who was injured, then a reference to an emergency squad extricating the dead man from the wreck. As a practical means of avoiding backtracking in cases like these, the Script Applier delays the instantiation (or "realization") process, but keeps track of the patterns that were matched by a story, and the episodes these are part of.

APPLY, using its record of the episodes of $VEHACCIDENT that were referenced, now proceeds to instantiate these episodes for inclusion in the final representation for the story. This process has several important sub-processes. First, when APPLY attempts to realize a pattern from an episode, it may encounter a Script variable not explicitly mentioned by the story. When this happens, it goes to TOK for a token having the appropriate properties. (TOK was given the properties of all variables at Script activation time.)

Next. when an episode Maincon, interference or resolution is encountered, APPLY attaches an indication of the place where these events occurred. This setting information is used by the summarizer. When an interference is seen, APPLY notes the fact, and searches for a resolution elsewhere in the story representation. Finally, when the mainpath causal chains through the episodes have been instantiated, APPLY calculates the appropriate causal relations among the episodes. The log to follow contains examples of all these processes.

COMPUTER OUTPUT	COMMENTARY
!(APPLY) BUILDING STORY REPRESENTATION FOR (TEXT . C1)	
MAKING STORY SEGMENT FOR SUBSCENE $CRASH1 IN $VEHACCIDENT	APPLY begins by instantiating the "crash" scene from $VEHACCIDENT
!(TOK) creating new token: LOC2	At APPLY's behest, TOK creates tokens for the (unspecified)

origin and destination of the $DRIVE the crash interrupted.

```
!(APPLY)
GOT TOKEN LOC2 FOR &ORIG1

!(TOK)
creating new token: LOC3

!(APPLY)
GOT TOKEN LOC3 FOR &DEST1

DEFINING SETTING FOR EVNT4
```

APPLY creates the setting "near Route 69" for the crash event.

```
!(TOK)
creating new token: LOC4

!(APPLY)
GOT TOKEN LOC4 FOR &ACCLOC

RUNNING PATTERN FUNCTION (RFCRA5)
BINDING VARIABLE &NVAL1 TO -6
```

APPLY assumes that the car in the accident was "badly damaged."

```
MAKING STORY SEGMENT FOR SUBSCENE
           $TREAT1 IN $VEHACCIDENT
```

Now APPLY realizes the episodes in which police, ambulance and medical examiner come to the scene.

```
!(TOK)
creating new token: HUM8
```

This is the unnamed person who saw the accident,

```
!(APPLY)
GOT TOKEN HUM8 FOR &ONEO

!(TOK)
creating new token: ORG2
```

and the police department he called.

```
!(APPLY)
GOT TOKEN ORG2 FOR &POLORG

INTERFERENCE ENCOUNTERED: EVNT14
DEFINING SETTING FOR EVNT14
DEFINING SETTING FOR EVNT17
```

"A man died" is recognized as an "interference" in $VEHACCIDENT. The setting for the interference and the episode Maincon.

```
MAKING STORY SEGMENT FOR SUBSCENE
           $TREAT2 IN $VEHACCIDENT

INTERFERENCE ENCOUNTERED: EVNT20
DEFINING SETTING FOR EVNT20
```

"A man was hurt" is also seen to be an interference.

```
!(TOK)
creating new token: STRUCT3
```

Tokens are created for:

```
!(APPLY)
GOT TOKEN STRUCT3 FOR &AMBULANCE
```

the ambulance

```
!(TOK)
creating new token: LOC5
```

```
!(APPLY)
GOT TOKEN LOC5 FOR &TRELOC
```

the emergency room

```
!(TOK)
creating new token: HUM9
```

```
!(APPLY)
GOT TOKEN HUM9 FOR &DOCTOR1
```

and the emergency-room doctor.

```
INTERFERENCE RESOLVED:
    (EVNT20 EVNT26)
```

The treatment resolves the hurt person's problem.

```
DEFINING SETTING FOR EVNT26
```

```
MAKING STORY SEGMENT FOR SUBSCENE
       $INVEST1 IN $VEHACCIDENT
DEFINING SETTING FOR EVNT33
```

Finally, APPLY realizes the investigation episode.

```
CONNECTING STORY SEGMENTS
EVENT GRAPH:
((EVNT1 EVNT2 EVNT3 EVNT4 EVNT7)
 (EVNT8 EVNT9 EVNT10 EVNT11 EVNT13
  EVNT17)
 (EVNT18 EVNT19 EVNT23 EVNT24 EVNT25
  EVNT26 EVNT28 EVNT29 EVNT30)
 (EVNT31 EVNT32 EVNT33))
DUMPING STORY REPRESENTATION
```

APPLY interconnects the four Script episodes it instantiated, displays the mainpath portions of each episode, and dumps the story representation.

Let's look at some of the data structures in the story representation. First, there is the variable !STORY, which gives global access to the memory representation.

```
!STORY: (SEQ SCLAB3)

SCLAB3:
  SCRIPTNAME $VEHACCIDENT               Situation
  MAINCON EVNT4                         Story Maincon
  SCENECONS (EVNT4 EVNT17 EVNT33)       Episode Maincons
  INTERFERENCE ((EVNT20 EVNT26)         Interference/resolution
                (EVNT14))                 events
  ENTRYCON EVNT1                        First event in the story

EVNT4:                                  The Maincon is the crash
  VALUE ((ACTOR STRUCTO                  itself
             <=> (*PROPEL*)
             OBJECT PHYSO
             TIME (TIME5))
  TOP SCLAB3                            The story label
  LABEL $CRASH1                         The episode
  SCRIPTID CRA4                         The Script pattern
  LOCALE (EVNT5)                        The setting
```

```
  PATHTYPE MAIN                               Pathvalue and
  PATHVALUE DEF                                  pathtype
  LASTEVENT (EVNT3)                           Predescessor
  NEXTEVENT (EVNT20 EVNT14 EVNT7)             Successors
  CAUSATION (RESULT RESULT RESULT)            Causal type

EVNT5:                                        Maincon setting:
  VALUE ((ACTOR LOC4 IS (*LOC*                 "near Route 69"
           VAL (*PROX* PART LINKO))))


EVNT14:                                       One result of the crash
 VALUE ((ACTOR HUMO ISTOWARD                    is that someone died.
             (*HEALTH* VAL (-10)))
            TIME (TIME17))
 TOP SCLAB3
 LABEL $TREAT1
 SCRIPTID TRE1
 LOCALE (EVNT15)
 PATHTYPE IMMBAKINF                           This is a "back" inference
 PATHVALUE INT                                  from "pronounced dead,"
 LASTEVENT (EVNT4)                              and an INTerference.
 NEXTEVENT (EVNT17)
 CAUSATION (INITIATE)

EVNT20:                                       Another result is that
 VALUE ((ACTOR HUM3 ISTOWARD                    somebody was injured.
             (*PSTATE* VAL (-3)))
            TIME (TIME17))
 TOP SCLAB3
 LABEL $TREAT2
 SCRIPTID TRE50
 LOCALE (EVNT21)
 PATHTYPE IMMBAKINF                           This is a "back" inference
 PATHVALUE INT                                  from "examined"
 LASTEVENT (EVNT4)                              and an INTerference.
 NEXTEVENT (EVNT19)
 CAUSATION (INITIATE)

EVNT26:                                       This is the resolution for
 VALUE ((<=> ($TREATMENT                        EVNT20: treatment.
               DOCTOR HUM9
               PATIENT HUM3))
        TIME (TIME31))
 TOP SCLAB3
 LABEL $TREAT2
 SCRIPTID TRE57
 LOCALE (EVNT27)                              This is a mainpath event,
 PATHTYPE MAIN                                 with pathvalue RESolution.
 PATHVALUE RES
 LASTEVENT (EVNT25)
 NEXTEVENT (EVNT28)
 CAUSATION (RESULT)
```

Here is the instantiated crash episode:

```
Episode: (EVNT1 EVNT2 EVNT3 EVNT4 EVNT7)

EVNT1:
 ((<=> ($DRIVE DRIVER HUM3 PASSENGER HUMO
              ORIG LOC2 DEST LOC3
              VEHICLE STRUCTO ROUTE LINKO))
       TIME (TIME2))
[There was an instance of the Script $DRIVE, HUM3
  driving, from LOC2 to LOC3 via LINKO]

EVNT2:
 ((<=> ($DRIVE DRIVER HUM3 PASSENGER HUMO
              ORIG LOC2 DEST LOC3
              VEHICLE STRUCTO ROUTE LINKO))
       TIME (TIME3) MODE (MMODE2))
[The driver lost control of the car: MMODE2 = *CANNOT*]

EVNT3:
 ((ACTOR STRUCTO <=> (*PTRANS*) OBJECT STRUCTO
                FROM (*TOPOF* PART LINKO))
       TIME (TIME4))
[The car left the road]

EVNT4:
 ((ACTOR STRUCTO <=> (*PROPEL*) OBJECT PHYSO)
        TIME (TIME5))
[It ran into an obstruction]

EVNT7:
 ((ACTOR STRUCTO ISTOWARD (*PSTATE* VAL (-6))
        TIME (TIME8))
[and was demolished]
```

Here, as an example of what the tokens look like after SAM has finished with them, are "Miller," "Hall" and "Flanagan Ambulance:"

```
HUMO:
 CLASS (#PERSON)
 SURNAME (HALL)
 PERSNAME (DAVID)
 GENDER (*MASC*)
 AGE (27)
 RESIDENCE (POLITO)
 SROLES (($VEHACCIDENT  . &DEADGRP)
        ($DRIVE . &PASSGRP1))

HUM3:
 CLASS (#PERSON)
 SURNAME (MILLER)
 PERSNAME (FRANK)
 GENDER (*MASC*)
 AGE (32)
```

```
  RESIDENCE (LOC1)
  SROLES (($VEHACCIDENT  . &HURTGRP)
         ($DRIVE . &DRIVER1))

 ORGO:
  CLASS (#ORGANIZATION)
  ORGNAME (FLANAGAN)
  ORGOCC ($AMBULANCE)
  SROLES (($VEHACCIDENT . &AMBORG))
```

Finally, we show another "interesting" event, the Maincon of the investigation episode. This event is marked as being of special importance because of the SCORECARD property:

```
 EVNT33:   [Onofrio decides not to prosecute]
  VALUE ((ACTOR HUM7 <=> (*MBUILD*)
           MOBJECT ((<=> ($PROSECUTION))
                        MODE (MMODE3))
        TIME (TIME319)
  [MMODE3 = (*NEG*)]
  TOP SCLAB3
  LABEL $INVEST1
  SCRIPTID INV3
  PATHTYPE MAIN
  PATHVALUE NOM
  LASTEVENT (INV2)
  SCORECARD ARREST

 HUM7:
  CLASS (#PERSON)
  SURNAME (ONOFRIO)
  PERSNAME (ROBERT)
  GENDER (*MASC*)
  TITLE (PATROLMAN)
  OCCUPATION (*POLICEMAN*)
  SROLES (($VEHACCIDENT  . &POLSTAFF))
```

6.3 Answering Questions

To suggest the quality of the understanding SAM achieved in reading this story, we now give an annotated log of a question-answering run based on the story representation SAM built. The question-answering module, QA, is an implementation of Lehnert's QUALM [18] designed for Script-based answer retrieval. As such, it uses the Conceptual Analyzer and PP-Memory in exactly the same way as the understanding configuration of SAM. (The English generator (ENGLSH) which expresses the answers is a modification of Goldman's BABEL [14].) Additionally, it uses many of the pattern-matching and inference techniques that were developed for the Script Applier. We indicate these processes below.

COMPUTER OUTPUT	COMMENTARY
!(QA) GETTING NEXT QUESTION	SAM starts up with QA in control. QA goes to PARSER for first question.

!(PARSER)
Was anyone killed?
 	
CONCEPT:
GN1007 =
((CON (NIL TIME (TIM3))
    LEADTO
((ACTOR (#PERSON REF (INDEF))
    TOWARD (*HEALTH* VAL (-10))
    LEAVING (*HEALTH* VAL (NIL)))
 INC (NIL)
 TIME (NIL)
 MODE (NIL)))
MODE (MOD2))
 	
MOD2 = *?*
 	
PARSING TIME: 47615 GCTIME: 3385
	PARSER analyzes this as a query (*?*) about whether an event of killing occurred, involving some unspecified (REF (INDEF)) person.

!(TOK) top level PARSER atom is: GN1007 processing PP: (#PERSON REF GN1023) creating new token: HUM101 top level TOK atom for GN1007 is MEM301	TOK does its usual job.

!(QA)
NEXT QUESTION:
((ACTOR HUM101 TOWARD
       (*HEALTH* VAL (-10)))
    MODE (MOAD1))
	This is the statement of the question QA will try to answer. [MOAD1 = (*?*)]

```
(QUESTION TYPE IS VERIFY)
(SEARCHING $VEHACCIDENT-SCRIPT
        STRUCTURE)
(NOT FOUND AT SCRIPT STRUCTURE
        LEVEL)
(SEARCHING CAUSAL CHAIN
        STRUCTURES)
(SEARCHING INFERENCES OFF MAIN PATH)
```

The question class is "verify," i. e., did this occur. QA searches the story representation from the top down.
Eventually it begins pattern matching in the causal-chain episodes, and their inferences.

```
PATTERN BACKBONE MATCHED AT EVNT14
(FOUND IN INFERENCE OFF MAIN PATH)

THE ANSWER IS:
(YES ((ACTOR HUMO ISTOWARD
            (*HEALTH* VAL (-10)))
       TIME (TIME17)))
```

The conceptual answer...

```
!(ENGLSH)
YES, DAVID HALL DIED.
```

...and its expression in English.

```
!(QA)
GETTING NEXT QUESTION

!(PARSER)
Was anyone hurt?
CONCEPT:
GN1077 =
((CON (NIL TIME (TIM5))
      LEADTO
  ((ACTOR (#PERSON REF (INDEF))
          TOWARD (*PSTATE* VAL (-3))
          LEAVING (*PSTATE*
                       VAL (NIL)))
      INC (NIL)
      TIME (TIM6)
      MODE (NIL)))
  MODE (MOD4))

PARSING TIME: 35230 GCTIME: 3398


!(TOK)
top level PARSER atom is: GN1077
processing PP:
(#PERSON REF GN1093)
creating new token: HUM102
top level TOK atom for GN1077
          is MEM313
```

The second question is similar to the first...

```
!(QA)
NEXT QUESTION:
((ACTOR HUM102
      TOWARD (*PSTATE* VAL (-3)))
     MODE (MOAD3))
(QUESTION TYPE IS VERIFY)
```

...and is answered the same way.

```
(SEARCHING $VEHACCIDENT-SCRIPT
      STRUCTURE)
(NOT FOUND AT SCRIPT STRUCTURE
      LEVEL)
(SEARCHING CAUSAL CHAIN
      STRUCTURES)

PATTERN BACKBONE MATCHED AT EVNT20
(FOUND IN INFERENCE OFF MAIN PATH)

THE ANSWER IS:
(YES ((ACTOR HUM3
      ISTOWARD (*PSTATE* VAL (-3)))
    TIME (TIME25)))

!(ENGLSH)
YES, FRANK MILLER WAS SLIGHTLY
      INJURED.

!(QA)
GETTING NEXT QUESTION

!(PARSER)
Why was Frank Miller hurt?
```

This is a question about the event which caused Miller's injury.

```
CONCEPT:
GN1142 =
((CON (*?*)
   LEADTO
   ((CON (NIL TIME (TIM8))
       LEADTO
   ((ACTOR (#PERSON GENDER (*MASC*)
                    FIRSTNAME (FRANK)
                    LASTNAME (MILLER))
       TOWARD (*PSTATE* VAL (-3))
       LEAVING (*PSTATE* VAL (NIL)))
         INC (NIL) TIME (TIM9)
         MODE (NIL)))
  MODE (MOD5)))
  MODE (NIL))

PARSING TIME: 47345 GCTIME: 6810

!(TOK)
top level PARSER atom is: GN1142
processing PP:
(#PERSON GENDER GN1166
         FIRSTNAME GN1167
         LASTNAME GN1168)
creating new token: HUM103
top level TOK atom for GN1142
             is MEM325
```

```
!(QA)
NEXT QUESTION:
((CON (*?*)
   LEADTO
    ((ACTOR HUM103
        TOWARD (*PSTATE* VAL (-3))))))

(QUESTION TYPE IS CAUSANT)
(SEARCHING $VEHACCIDENT-SCRIPT
               STRUCTURE)
(SEARCHING INTERFERENCE RESOLUTION
               PAIRS)
(NOT FOUND IN INTERFERENCE RESOLUTION
               PAIRS)
(NOT FOUND AT SCRIPT STRUCTURE LEVEL)
(SEARCHING CAUSAL CHAIN STRUCTURES)
(SEARCHING INFERENCES OFF MAIN PATH)
(SEARCHING SCRIPTAL WORLD KNOWLEDGE)
(NOT FOUND IN SCRIPTAL KNOWLEDGE)
```

Because this is a question about causality, QA searches the interference/resolution data attached to the global story variable.

```
THE ANSWER IS:
(BECAUSE
    ((ACTOR STRUCTO <=> (*PROPEL*)
                    OBJECT PHYSO)
               TIME (TIME5)))

!(ENGLSH)
BECAUSE AN AUTOMOBILE HIT A TREE.

!(QA)
GETTING NEXT QUESTION

!(PARSER)
Did Miller go to the hospital?
```

PARSER interprets this sentence as a PTRANS of Miller, by Miller, to a hospital.

```
CONCEPT:
GN1217 =
((ACTOR   GN1240 <=> (*PTRANS*)
               OBJECT GN1240
               TO (*PROX* PART
    (#ORGANIZATION ORGOCC ($HOSPITAL)
                   REF (DEF)))
      FROM (NIL)
      INST (NIL))
      MODE (MOD7)
      TIME (TIM11))
GN1240 =
(#PERSON LASTNAME (MILLER))

PARSING TIME: 55307 GCTIME· 6564

!(TOK)
top level PARSER atom is: GN1217
processing PP:
(#PERSON LASTNAME GN1229)
```

```
creating new token: HUM104
processing PP
(#ORGANIZATION ORGOCC GN1266
               REF GN1268)
creating new token: ORG101
top level TOK atom for GN1217
              is MEM340

!(QA)
NEXT QUESTION:
((ACTOR HUM104 <=> (*PTRANS*)
               OBJECT HUM104
               TO (*PROX* PART ORG101))
 MODE (MOAD7)))
```

QA receives the question statement, a verification query.

```
(QUESTION TYPE IS VERIFY)
(SEARCHING $VEHACCIDENT-SCRIPT
             STRUCTURE)
(NOT FOUND AT SCRIPT STRUCTURE
             LEVEL)
(SEARCHING CAUSAL CHAIN STRUCTURES)
```

QA begins searching the story representation.

```
TRYING INFERENCE TYPE CONVEY ON EVNT24
PATTERN BACKBONE MATCHED ON QUESTION
                  CONCEPT:
((ACTOR HUM3 <=> (*PTRANS*)
             OBJECT HUM3
             TO (*PROX* PART ORG1))
      TIME (TIME29))
```

Eventually it tries a Conveyance inference on one of the concepts in the representation,

```
SUCCESSFUL MATCH ON DERIVED PATTERN
```

which works.

```
(FOUND IN MAIN PATH)

THE ANSWER IS:
(YES ((ACTOR ORGO <=> (*PTRANS*)
                  OBJECT HUM3
                  TO (*PROX* PART ORG1))
TIME (TIME29)))

!(ENGLSH)
YES, THE FLANAGAN AMBULANCE
COMPANY TOOK HIM TO THE MILFORD HOSPITAL.
```

Chapter 7
Representing Conceptual Nominals

7.1 Motivation

Script-based understanding is a process of recognizing real-world events as instances of an action imbedded in a Script. Each natural-language sentence describing such an event contains references to conceptual "nominals," people, objects, places, and other entities with concrete references. In the Conceptual Dependency system [32], conceptual nominals are called Picture Producers (PPs) because they tend to produce an image in the mind of a hearer or reader. "The Eiffel Tower," for example, summons up a visual image of a particular kind of structure, associated with a particular city. This visual impression gives us information in addition to other, "static" facts we may know about the Eiffel Tower, for example, that it is made of steel and has elevators. Part of the job of applying Scripts is to correctly assign roles and props to PPs appearing in a text.

In SAM, "recognition" is a cooperative process carried out by the module which keeps track of Picture Producers (PP-Memory), and the module which knows about Scripts (the Script Applier). Identifying an input is is a two-stage process. First, the Script Applier matches the constant parts of a Script pattern against the corresponding parts of a story input, to see if the event is of the right type. Then it examines the PPs from the input to determine whether they can fulfil the function defined by the associated Script variable. In this process, the Script Applier continually uses information provided by PP-Memory, particularly data about PPs resulting from the internalization of an ELI input. Comprehension, therefore, is intimately bound up with the memory structures that PP-Memory defines for each of its tokens.

Originally, SAM dealt with simple stories about "simple" PPs. It read about "John and Mary" riding on "a bus," to get to "a restaurant." When we extended SAM to get it to read newspaper articles, however, we ran into the problem that PPs in the world come in a myriad of varieties, and newspaper writers use a number of techniques to describe them. Consider, for example, the following typical -- but very complex -- noun group describing one participant in a car accident: "Frank Miller, 32, of 593 Foxon Road, the driver."

How can we deal with this complexity? Since Picture Producers are a part of the Conceptual Dependency system, we applied the same methodology to them as CD applied to the analysis of verbs. We tried to find a small number of "primitive" entities, each with a roster of optional and mandatory "slots" or cases, into which all surface forms having the "equivalent" conceptual content could be mapped. The benefits of having such a representational scheme for PPs would then be the same as provided by the eleven CD ACTs and their associated case-frames.

In analysis, for example the conceptual "frame" provided by each primitive PP-class would operate as a source of predictions about words and phrases to be seen further on in the input stream. (The process which exploits this information is a sub-part of ELI called the Noun Grouper (NGP). NGP is described by Gershman in [13].)

In inference tasks, each PP-class has a characteristic inference process designed to fill in the gaps in a PP-description. In particular, reference specification would be speeded up because each new entity created would carry along the information, immediately available, about which kinds of PPs it cannot be; and which slots to look at for contradictions. For example, the PP-class "human" has slots for gender, personal name and surname. The first time the PP "John Smith" is bound to a Script variable, all three of these slots would be filled in. If a reference to "a truck" appeared in a subsequent input, the Script Applier would never consider it for the role "Smith" is bound to, since it belongs to a different PP-class. "Smith," "John" and "he" would be accepted, because the properties of the associated PP form a subset of the ones that the Script Applier received for "John Smith." The process which does this is called Rolemerge, and was discussed in Chapter 4.

Finally, the generation process would be enhanced because rules about "what to say" could be formulated for each PP-class, independently of the others. "Pronominalization" of PPs in generation, as an example, would simply become a process of specifying which subset of slots in the PP-frame should be expressed each time a surface description is needed. For example, once the appropriate story representation has been built by the Script Applier, the token corresponding to "Frank Miller," above, would contain all the information needed to express him as:

Frank Miller, age 32, residence at 593 Foxon Road, New Haven, Connecticut, the driver of the car.

The driver, a New Haven, Connecticut man.

Miller, the 32-year-old driver.

He.

(Many other realizations of this PP are also possible in SAM.)

The world of Picture Producers that SAM deals with in newspaper stories can conveniently be divided into three parts: actors, things and places. Each of these broad classes is subdivided into a small number of "primitive" PP-classes. Each primitive class, in turn, defines a limited set of slots, called a conceptframe, which an input may fill. For example, "Frank Miller" belongs to the primitive class of "persons" within the class of actors. The PP-class #PERSON (a preceding "#" designates a PP primitive) has the following conceptframe:

```
#PERSON:
  PERSNAME                 personal name
  SURNAME                  family name
  AGE                      age
  RESIDENCE                the place where this person lives
  GENDER                   sex
  TITLE                    a title such as "Mr" or "Premier"
  OCCUPATION               a long-term occupation, such as
                             "medical doctor"
  EMPLOYER                 the organization the person works
                             for
  FUNCTION                 a short-term function, such as
                             "driver"
  POLITY                   a political unit the person serves
                             in an official capacity
  FAMILY                   a pointer to the person's family
                             group
  HUSBAND/WIFE/SON/etc.    a pointer to another person related
                             to this one
```

Note that some of the slots in a PP-description may contain PPs of the same or other classes. For example, the RESIDENCE slot may be filled by a "structured physical object," the building where the person lives. The POLITY slot may point to a political entity such as the state or nation an official person (e. g., the President of Liberia) serves. The EMPLOYER slot is usually filled by a reference to the organization the person works for, as in "the man from UNCLE" or "the phone company man." The HUSBAND/WIFE slot, if filled, contains a reference to another person having the named relation to this one. For example, ELI would build the following structure for "John's wife Mary:"

```
(#PERSON
    GENDER (*FEM*)
    PERSNAME (MARY)
    HUSBAND (#PERSON
                GENDER (*MASC*)
                PERSNAME (JOHN)))
```

In addition to the conceptframe, each PP in SAM may possess one or more "general" properties which are defined by PP-Memory as a part of its process of internalizing an input from ELI. One of these properties, REFERNT, points to the long-term memory structure possessed by PP-Memory for a PP which is also a permanent token. An example of a permanent token is "Hua Kuo-feng:"

```
HUMO:
  CLASS (#PERSON)
  PERSNAME (KUO-FENG)
  SURNAME (HUA)
  REFERNT (!HUM110)
```

where the permanent token !HUM110 contains additional information such as:

```
!HUM110:
  CLASS (#PERSON)
  PERSNAME (KUO-FENG)
  SURNAME (HUA)
  GENDER (*MASC*)
  TITLE (CHAIRMAN)
  OCCUPATION (*HEADOFSTATE*)
  POLITY (!POL105)
```

Note the reference in !HUM110 to the permanent token for "Communist China:"

```
!POL105:
  CLASS (#POLITY)              [this PP is a primitive
                                 "political unit"]
  POLTYPE (*NATION*)           [of the "national" variety]
  POLNAME (COMMUNIST-CHINA)
  POLGOV (*CP*)                [this polity is a Communist
                                 dictatorship]
  CAPITAL (!POL106)            [with capital at Peking, a
                                 municipal political unit]
```

When PP-Memory identifies a permanent token, it copies over all the information available from the referent onto the new token it has just created.

Other "general" PP properties are syntactic ones computed by PP-Memory which refer to the expression of the PP either as seen by the Analyzer or realized by the Generator. For example, the SURFSLOTS property identifies the PP-slots which were explicitly filled by an input. For "Hua Kuo-Feng," this would be:

```
HUMO:
  SURFSLOTS (CLASS PERSNAME SURNAME)
```

The SURFSLOTS property is used in formulating answers to questions such as "Who is Hua Kuo-Feng?" Properties used by the Generator include: (1) REFDEF, which tells whether the token is to be expressed definitely, as in "the Eiffel Tower;" (2) NODET, which tells when a determiner is not to be used, as in "Mt Everest;" and (3) SURFIRST, which tells it whether the SURNAME of a person is to be expressed before the PERSNAME, as is the case for Oriental names. This information exists in PP-Memory as a part of the lexical information associated with its permanent tokens. "Hua Kuo-Feng," therefore, would look like this, after internalization by PP-Memory:

```
HUMO·
  CLASS (#PERSON)
  PERSNAME (KUO-FENG)
  SURNAME (HUA)
  GENDER (*MASC*)
  TITLE (CHAIRMAN)
  REFERNT (!HUM110)
  OCCUPATION (*HEADOFSTATE*)
  POLITY (!POL105)
```

SURFSLOTS (CLASS PERSNAME SURNAME)
SURFIRST (T)

We give the details of the PP-classification used in SAM in succeeding sections. We should note, however, that the representational scheme, while practical, has several shortcomings. For one thing, it is by no means complete. (A more detailed system for representing physical objects is given, for example, by Lehnert [18], which is useful for certain kinds of low-level inferencing tasks.) Furthermore, it is untidy. We have found it convenient to define special-purpose PPs for various Script domains. A further criticism is that our classification is anti-hierarchical. We know, for example, that people are physical objects too, and both actors and things can be said to define a kind of "place." Neither of these facts appears directly in our classification. We can only answer the first two objections on pragmatic grounds. The scheme has been found useful for doing the "complete" job of understanding, from analysis to response. across a wide variety of knowledge domains.

The third aspect of the representation scheme, however, is a deliberate one. We built it in because of our observation that PPs fit into Scripts, not because they inherit features from abstract classes, as in a "semantic memory," but because of how they function there. In Scripts, actors "do" things, they "manipulate" objects while acting, and the doing always goes on in a well-defined "somewhere."

This view of representation is an "episodic" one, which we believe is characteristic of human memory, as well. As a simple example of this, consider how a person is likely to construct an answer to the question "What are the fifty states of the US?" (This example, and the episodic/semantic memory distinction, are discussed in detail by Schank in [31].) Typically, a person would reply by consulting a remembered image of the US, and trying to fill in the empty spaces. Alternatively, he might consult his episodic knowledge of the "What states have I been in on trips?" variety. It seems very unlikely that people have a memory node labelled "States of the US," with an attached list, which they access when a question like this is asked.

We have already suggested how a good way of representing PPs has important advantages for the analysis and generation processes in SAM. From the point of view of SAM's memory and inference mechanisms, we were interested in a classification of PPs which would facilitate (1) finding out what function it was serving in an a Script; and (2) identifying an already-bound PP no matter how the story described it. The PP-classes used by SAM, therefore, highlight two important kinds of slots, corresponding to indicators in the surface form of what might be a given PP's (1) function and (2) tag (unique name). Consider, for example, the structure ELI would build for:

"Frank Miller, 32, of 593 Foxon Rd, the driver"

```
(#PERSON PERSNAME (FRANK)
         SURNAME (MILLER)
         AGE (32)
         GENDER (*MASC*)
         FUNCTION (*DRIVER* REF (DEF))
         RESIDENCE (#LOCALE
                        STREETNUMBER (593)
                        STREETNAME (FOXON ROAD)
                        STREETYPE (ROAD)))
```

The PP-concept for this person has two tag-pointers (PERSNAME and SURNAME), one function-pointer (FUNCTION), and secondary AGE, GENDER and RESIDENCE slots.

We discuss persons, and the other primitive PP-classes, in succeeding sections. In each case we will concentrate on how each class fits into one or more Scripts, and what kinds of inferences each class organizes in a Script context.

7.2 Actors

Active entities in Scripts are more complete realizations of the ACTORs of Conceptual Dependency. CD representation is centered on the idea of ACTORs performing various ACTs, symbolized by the two-way dependency relation ACTOR ⇔ ACT. Previous work in CD (e. g., [32]) has concentrated the ACTs, since the existence of a small set of ACTs could be exploited in analysis to define a conceptual "frame" for a thought, usually derived from the surface verb; in inference, with a characteristic inference process tied to each ACT; and in generation, with a discrimination net per ACT for selecting a surface verb and case frame. With the application of Scripts to newspaper stories, however, we found a serious need for ways of representing the PPs underlying multi-component noun groups, of which those describing ACTORs are the most complicated.

What kinds of ACTORs do newspaper stories refer to? ACTORs seem to fit into two broad classes, animate entities and physical forces. These classes are subdivided, in SAM, into:

Class of ACTOR	Example
Animate Actors:	
(1) persons	Mao Tse-Tung
(2) groups	John and Mary
(3) organizations	The New York Times
(4) polities	California
Physical Forces:	
(1) simple forces	gravity
(2) mechanical forces	a car's engine
(3) complex forces	Hurricane Emma

7.2.1 Persons

The most familiar type of ACTOR is, of course, a person. The Scripts a person participates in can be distinguished from one another on the basis of setting.

One important setting is the home. Here we have simple Scripts for activities such as eating, getting dressed, preparing for bed, etc. More complex Scripts are associated with a person's job, that is, with the "office" setting. Examples include typing, giving reports, running machinery, going on a coffee break, etc. Stories often refer to a person's role in a Script of this kind, as in "John, the bank teller" (by function), or "Dean Applegate" (by title). Often the person is being characterized as an "agent" of a family, organization or polity, as in "IBM sent a customer engineer to Brooklyn." A third class of Scripts that a person has a role in are those Transactions in which he engages as a member of the public, as in "John had breakfast at Tiffany's." Such Transactions include not only interactions with commercial organizations such as restaurants and hotels; but also, as we explained in Chapter 3, with service and governmental organizations such as the Sanitation Department and the IRS.

The Script Applier, in understanding a story, has to identify each reference to a PP that occurs in a text, and to create a token for a Script variable which a story hasn't explicitly bound. For persons, "identification" involves determining the person's tag and Script function. A person's tag is given by the PERSNAME (with associated GENDER) and SURNAME slots on the token(s) created for it. The Script Applier doesn't consider that it completely "knows" the person until it has a filler for at least one of these slots. For example, in a story beginning "A New Jersey man was killed in an accident," the token for the dead man initially contains only information about his RESIDENCE. In reading subsequent inputs, the Applier will still be looking for this person's tag, and thus will be able to accept: "David Hall was pronounced dead at the scene." Once the tag has been determined, the system assumes that further references to the tag will form a subset of the originally defined slots. Thus, we may read about "Hall" (CLASS SURNAME) or "he" (CLASS GENDER REF).

When instantiating a story episode, the Applier may need to create a token for a person only implicitly referenced by the story. When this happens, it causes PP-Memory to create a token of the appropriate function In reading "John went into a restaurant and ordered a hamburger," a token for the unnamed "waiter" will be created which looks like this:

```
HUM1:
 CLASS (#PERSON)
 SCRIPTROLES ((&RWAITER . $RESTAURANT))
 GENDER (*MASC*)
 ELEX (WAITER)
 SLEX (MOSO)
```

The definition of "waiter" includes the information that this token filled the variable role &RWAITER in $RESTAURANT. In cases where the

person fills multiple roles in a story, SCRIPTROLES is updated accordingly. For example, "David Hall" may simultaneously fill the "driver" role in $DRIVE and the "dead person" role in $VEHACCIDENT. In this case, HUM1 would get the property:

```
SCRIPTROLES ((&DRIVER . $DRIVE)(&DEADGRP . $VEHACCIDENT))
```

The ELEX and SLEX properties give the output generators (here, for English and Spanish) a base lexeme with which to express this token: as "a waiter" and "el muso," respectively. (Note that the selection of GENDER could just as well have been "feminine." The Script function is sexless.) Sometimes there is no explicit function word for a Script variable. For example, all the ELEX contains for the person who discovers a car crash and calls the police is "someone."

7.2.2 Groups

Single persons cooperating closely to perform a Scriptal activity make up a group. Each member of a group can perform any of its characteristic activities, and the notion of "control" or "leadership" within the group is understated. A professional sports team (e. g., the Yankees), a police squad (e. g., the SWAT team), representatives of authority (White House spokesmen), and persons engaged in a conversation are examples of groups. Groups often act as agents of controlling organizations. For example, "the police department sent the riot squad." In many cases there is a Script which is closely associated with the group. The Script may be a "simple" one, as in "Patrolman Jones and his partner walked the beat," or it may be a Transaction, as in "John and his girl friend went to the movies."

The conceptframe associated with a group is:

```
#GROUP:
  TYPE         what kind of a group is it
  GRNAME       the group's name
  NUMBER       how many members
  MEMBER       defines one member of the group
  RESIDENCE    the place where the group lives
  GROCC        the Script the group does
  EMPLOYER     the entity the group works for
```

Groups are specified in inputs in a number of different ways. The commonest way is by naming its MEMBERs:

```
"John and Mary"
(#GROUP
    MEMBER (#PERSON PERSNAME (JOHN) GENDER (*MASC*))
    MEMBER (#PERSON PERSNAME (MARY) GENDER (*FEM*)))
```

After PP-Memory finishes with this, we have:

```
GROUP0:
  CLASS (#GROUP)
  MEMBER (HUM0 HUM1)        [John and Mary]
  NUMBER (TWO)
```

PP-Memory will always try to make an inference about the number of the group, since this is useful for reference specification. For example, a further reference to this group in ELI output might be:

```
"both"
(#GROUP
   REF (DEF)
   NUMBER (2))
```

A more complicated group is formed by a family. A typical ELI representation for one might be:

```
"John Gavin and his wife Mary"
(#GROUP
   MEMBER (#PERSON
               PERSNAME (JOHN)
               SURNAME (GAVIN)
               GENDER (*MASC*))

   MEMBER (#PERSON
               PERSNAME (MARY)
               HUSBAND (#PERSON GENDER (*MASC) REF (DEF))
               GENDER (*FEM*)))
```

Note that ELI has not made any inference about Mary's last name, or the referent of "his." After tokenization, we would have:

```
GROUP1:
 CLASS (#GROUP)
 MEMBER (HUM2 HUM3)       [John and Mary]
 NUMBER (TWO)
 TYPE (*FAMILY*)
 GRNAME (GAVIN))

HUM3:                     [This is Mary]
 CLASS (#PERSON)
 PERSNAME (MARY)
 SURNAME (GAVIN)
 GENDER (*FEM*)
 HUSBAND (HUM2)
 FAMILY (GROUP1)
```

Again, PP-Memory attempts to fill in the name, type and number of the group, since the Script Applier may need these properties to specify the reference, for example, in "the Gavins." "the Gavin family" and "both Gavins."

Other ways a group may be specified is by RESIDENCE or FUNCTION. Suppose we have an initial reference to a family: "Enver and Mrs Hoxha." Since these are permanent tokens, we would end up with a token

in which the RESIDENCE is the same as the RESIDENCE of its members.

```
GROUP2:
 CLASS (#GROUP)
 NUMBER (TWO)
 TYPE (*FAMILY*)
 MEMBER (HUM4 HUM5)
 RESIDENCE (POLIT2)

POLIT2:
 CLASS (#POLITY)
 POLTYPE (*NATION*)
 POLNAME (ALBANIA)
```

The group inferences that PP-Memory has made would then be used by the Script Applier to solve the reference problem raised by "the Albanian party:"

```
GROUP3:
 CLASS (#GROUP)
 REF (DEF)
 RESIDENCE (POLIT1)      [this is the nation-polity, Albania]
```

Groups described by function include:

```
"an emergency squad"
(#GROUP
  REF (INDEF)
  GROCC ($RESCUE))
```

and:

```
"a police patrol"
(#GROUP
   REF (INDEF)
   GROCC ($PATROL)
   EMPLOYER (#ORGANIZATION
                ORGOCC ($POLICE)))
```

A limiting case of a group is a single person. For example, if we have read that "John Gavin died," and are asked "How many people died?", we have a question which is literally about a group:

```
"How many?"
(#GROUP
   REF (DEF)
   NUMBER (*?*))
```

In SAM, the processes which the Script Applier uses to find a single member of a previously mentioned group are also used by the question-answerer to make the group inference needed above.

7.2.3 Organizations

An <u>organization</u> is an assemblage of persons or groups existing to interact in a certain specified way, via a Script Transaction, with the general public. Aside from the characteristic Transaction, an organization also may have a distinguished "executive," a collection of characteristic agents, and a well-defined residence. The "police department," for example, has a "chief of police," sends "squad cars" or "riot control teams" to various places, and lives in "the station house." The token created by PP-Memory for the permanent token "Branford Police Department" would be:

```
ORGO:
 CLASS (#ORGANIZATION)       [the police department]
 ORGOCC ($POLICE)
 ORGNAME (BRANFORD)
 EXECUTIVE (HUM6)
 RESIDENCE (STRUCT2)

HUM3:                        [executive of the organization]
 CLASS (#PERSON)
 TITLE (POLICE CHIEF)

STRUCT2:                     [its residence]
 CLASS (#STRUCTURE)
 TYPE (*BUILDING*)
 PARTOF (POLIT3)

POLIT3:                      [where the residence sits]
 CLASS (#POLITY)
 POLTYPE (*MUNIC*)
 POLNAME (BRANFORD)
```

The conceptframe for organizations is:

```
#ORGANIZATION:
  ORGNAME         organization name
  ORGOCC          primary Transaction
  EXECUTIVE       controlling person or group
  RESIDENCE       where the controller "lives"
  STAFF           pointer to an agent of the
                     organization
  EMPLOYER        pointer to the organization
                     or polity controlling it
```

The most important feature of an organization, from the Script point of view, is its "occupation," or ORGOCC. The ORGOCC, which is always a Script, is picked up by PP-Memory and sent off to the Script Applier in the form of a processing suggestion for a Script context to be tried for the current input. For example, if we had "A fire broke out at 293 Elm Street last night. Engine Co No 9 came to the scene," the Script $FIREDEPT would be dispatched to the Applier as a candidate for processing the second Conceptualization. The typical way texts first refer to an organization is by ORGNAME, which, if the organization is a permanent token, implies the ORGOCC:

```
"Leone's"
ORG1:
 CLASS (#ORGANIZATION)
 ORGNAME (LEONE'S)
 ORGOCC ($RESTAURANT)
```

Thereafter, a story will typically refer to the organization by its occupation alone, as in "the restaurant."

Another reference to an organization in text which requires an inference is via the EMPLOYER slot on a group or person. A example (ELI output) is:

```
"an IBM salesman"
(#PERSON
  OCCUPATION (*SALESPERSON*)
  GENDER (*MASC*)
  EMPLOYER (#ORGANIZATION
              ORGNAME (IBM)
              ORGOCC ($RETAILSALES)))
```

After internalization, Script Applier would receive two tokens, one for the person, one for the organization: the first linked to the second via EMPLOYER, the second to the first (by PP-Memory inference) through the STAFF slot.

7.2.4 Polities

The highest class of animate actor, and the hardest one to characterize, is the <u>polity</u>. A polity defines a unit of government such as a city, state or nation, or a supranational body such as the U. N. The conceptframe defined in SAM for polities consists of:

```
#POLITY:
  POLTYPE                  polity type
  POLNAME                  polity name
  POLGOV                   government type
  EXECUTIVE                controller of the polity
  RESIDENCE                where the controller lives
  PARTOF                   next level polity
```

For example, the token produced by PP-Memory for (the permanent token) "Connecticut" would contain:

```
POLIT7:
 CLASS (#POLITY)
 POLTYPE (*STATE*)
 POLNAME (CONNECTICUT)
 POLGOV (*DEM*)          [the state is nominally a democracy]
 EXECUTIVE (HUM7)        [pointer to the governor]
 RESIDENCE (POLIT8)      [pointer to the municipality,
                           Hartford, where the executive is]
 PARTOF (POLIT9)         [pointer to the national unit]
```

Although polities at first sight may seem to be like super-organizations, their most important activities are complex, mostly non-Scriptal ones having to do with "politics" and "coercion." Organizations, by contrast, are always associated with definite Transaction Scripts, which they use in dealing with the public. We cannot do more here than to list some of the more obvious properties of polities, and show how these are reflected in text.

First of all, a polity always has a well-defined residence, where its executive lives, and a distinct area, populace and set of organizations which it controls. References to a polity's activities may use the executive or residence in place of the polity itself, as in:

"New York State announced..."
"Albany announced..."
"Governor Carey announced..."

Sometimes the executive is a group, as in "the Politburo." The "chain of command" thus defined for each polity is the basis for the Agency Inference discussed in Chapter 5.

A polity's agents are nearly always organizations. These organizations are of two varieties: ones concerned with the control of its own area (e. g., the police or IRS); and those which deal with other political units (e. g., the military or diplomatic service). When an organization is given as an agent of a polity, the link is provided through the EMPLOYER slot, as in:

```
"the Connecticut state police"
(#ORGANIZATION
    ORGOCC ($POLICE)
    ORGNAME (STATE)
    EMPLOYER (#POLITY POLTYPE (*STATE*)
                      POLNAME (CONNECTICUT)))
```

7.2.5 Forces

The last kinds of ACTOR appearing in Scripts are forces. These ACTORs are "inanimate," since they have no intentions, as such, but simply act. SAM makes use of three kinds of forces. There are simple forces, such as gravity. There are forces generated by artificial mechanisms such as engines. Finally, there are complex forces, large natural phenomena such as hurricanes and earthquakes.

Simple forces are those capable of making objects move, chiefly gravity, the wind or waves ("primary" forces); and the force which is carried by objects in motion ("motive" forces). Each of the primary forces can naturally be thought of as the ACTOR in a PTRANS Conceptualization, giving motive force to the OBJECT PTRANSed via a PROPEL Instrument. For example, we may have:

"A tree fell to the ground."

```
((ACTOR (#NATFORCE TYPE (*GRAVITY*) <=> (*PTRANS*)
                OBJECT (#PHYSOBJ TYPE (*TREE*))
                TO (#GEOFEATURE TYPE (*LANDSURFACE*))))
```

(Here "ground" has been rendered as a "geographical feature." See Section 7.4 for further discussion.) Simple forces always act when their disabling conditions are removed. For gravity, the disabling condition is a condition of support.

Mechanical forces are generated by engines and other artifacts which operate, in most cases, under human control. Control in the case of a mechanical force, is expressed by the person's executing a Script. So, for example:

"John drove the car to Boston"

```
((ACTOR (#PERSON PERSNAME (JOHN) <=> (*PTRANS*)
    OBJECT (#STRUCTURE TYPE (*CAR*))
    TO (#POLITY POLTYPE (*MUNIC*) POLNAME (BOSTON))
    INST ((<=> ($DRIVE DRIVER "John")))))
```

(Vehicles are treated as "structured" objects, as discussed in Section 7.3.) If we read about a PTRANS/PROPEL Conceptualization involving an object imbued with mechanical force, we can usually infer a loss of control. Suppose we have:

"A bus ran into a tree."

```
((ACTOR (NIL) <=> (*PTRANS*)
    OBJECT (#STRUCTURE TYPE (*BUS*))))
           LEADTO
((ACTOR (#STRUCTURE TYPE (*BUS*) <=> (*PROPEL*)
        OBJECT (#PHYSOBJ TYPE (*TREE*)))) .
```

The inferences from the PROPEL event are, first, that the ACTOR in the PTRANS is the bus's engine, and:

"driver lost control"

```
(((<=> ($DRIVE DRIVER "someone" VEHICLE "bus"))
    MODE (*CANNOT*))
```

Adult language users are aware that the causes of natural phenomena such as hurricanes and earthquakes are complex and not well understood. References to them in text treat them as if they were actors. For example, we might have:

"A massive earthquake struck northern Italy"

```
((ACTOR (#NATFORCE TYPE (*EARTHQUAKE*)) <=> (*PROPEL*)
    OBJECT (#POLITY POLTYPE (*NATION*)
                    POLNAME (ITALY))) MANNER ("violent"))
```

7.3 Physical Objects

The second main classification of the world of PPs in SAM consists of physical objects. By "physical object," we mean artifacts, things which have been manufactured by humans for their own use. Thus, we are excluding volitional entities such as people, and geographical entities such as mountains, although they too are physical objects. From the Script point of view, people and other animate entities are more naturally considered as possible roles in Scripts, geographical entities as possible settings for Scripts.

SAM uses a twofold sub-classification of PPs. There are "simple" physical objects, whose internal structure is of no interest in Scripts, and "structured" physical objects, which have parts which can house important Script activities.

7.3 1 Simple Objects

Simple objects belong to the PP-class #PHYSOBJ, which has the conceptframe:

```
TYPE          what kind of object it is
FUNCTION      what its function is
OWNER         who it belongs to
PRNAME        the object's identifier
```

Note that we are not saying that "simple" objects don't have parts, just that the parts are never singled out in the Scripts the object is a prop in. For example, a fork has a handle and tines, but a person uses one as a unit. The object "a fork" is specified by TYPE:

```
"a fork"
  PHYSO:
    CLASS (#PHYSOBJ)
    TYPE (*FORK*)
    REF (INDEF)
```

Objects can also be specified by FUNCTION, as in:

```
"a utensil"
  PHYS1:
    CLASS (#PHYSOBJ)
    FUNCTION (*UTENSIL*)
    REF (INDEF)
```

or by PRNAME:

```
"John's Rolex"
  PHYS2:
    CLASS (#PHYSOBJ)
    TYPE (*WATCH*)
    PRNAME (ROLEX)
    OWNER (HUMO)           This is John
    REF (INDEF)
```

The latter case also illustrates the use of the OWNER slot. The PRNAME, if filled in for a simple object, indicates either the manufacturer, as above, or the place of origin, as in "Irish handkerchief."

7.3.2 Structured Objects

For SAM, an object has structure if it has parts which figure independently in a Script. The most interesting structured physical objects are conveyances and buildings. These have components which are of interest because scenes of a Script can go on there. The conceptframe for the PP-class #STRUCTURE is the same as for simple objects, with additional slots for PARTs and Scripts the structure can be the SETTING for. For example, a simple representation for a passenger train would be:

```
STRUCT0:
 CLASS (#STRUCTURE)
 TYPE (*PASSTRAIN*)
 PART (#STRUCTURE TYPE (*TRAINENGINE*))
 PART (#STRUCTURE TYPE (*TRAINCAR*))
```

This, of course, is grossly oversimplified. For example, we know that trains typically have more than one car However, it does get at the important points about the parts of a train. There is an engine, where the motive force which moves the train is generated and controlled, and a passenger car, where various parts of $TRAIN are acted out. (In practice, PP-Memory would assign each substructure its own token)

The SETTING property is often used to designate different rooms of a building by the Script which the room houses. So, for example, "a bank" would look like:

```
STRUCT1:
 CLASS (#STRUCTURE)
 TYPE (*BUILDING*)
 SETTING ($BANK)
 PART (STRUCT2)
 PART (STRUCT3)

STRUCT2:
 CLASS (#STRUCTURE)
 TYPE (*ROOM*)
 SETTING ($WALKUPTELLER)

STRUCT3:
 CLASS (#STRUCTURE)
 TYPE (*ROOM*)
 SETTING ($MAKELOAN)
```

This bank has distinguished spaces for the tracks of $BANK in which people deal with tellers or arrange for loans.

7.4 Places

Every PP can be said to define a "place." Mobile ACTORs carry their places around with them, while the place associated with a fixed structure sits still. In addition to these, SAM uses several other kinds of fixed locations. There are "simple" locations, such as addresses, locations defined by geograpical entities such as lakes and mountains, and the special "places" associated with roads.

The representation of these kinds of entities, and their associated inferences, are not as highly developed as those for actors and objects. The main reason for this is that they are usually not strongly identified with a Script.

7.4.1 Simple Locales

Abstract references to location are handled in SAM by means of the primitive PP-class #LOCALE. Most references of this type simply refer to the place where something is happening in an unstructured way, as in:

"The scene of the accident"

```
LOCO:
 (#LOCALE REF (DEF)
          REL ((<=> ($ACCIDENT PLACE LOCO))))
```

or:

"Where?"

```
 (#LOCALE SPEC (*?*))
```

A special case of the simple locale is the address of a person or organization, when this is given in terms of street number, street, etc. For example, ELI might create the following description:

"Joe Doakes, of 593 Foxon Road"

```
(#PERSON "Joe Doakes"
      RESIDENCE (#LOCALE STREETNUMBER (593)
                         STREETNAME (FOXON ROAD)
                         STREETTYPE (STREET)))
```

PP-Memory, on receiving this, should make an inference about the existence of a building (#STRUCTURE) at that address, or in the town or state the building is presumably in (e. g., "a New Jersey man"). This inference, however, was not needed to solve the reference problems in the stories discussed in this thesis.

7.4 2 Geographical Features

A second class of generalized places consists of geographical features both on land and sea. These may be named:

```
"Mt Everest"

(#GEOFEATURE GEOTYPE (*MOUNTAIN*)
             GEONAME (MT EVEREST))
```

or not:

```
"a lake"

(#GEOFEATURE GEOTYPE (*LAKE*)
             REF (INDEF))
```

Note that GEONAME needs to be quite explicit: we could have "Long Mountain" or "Lassen Peak " Organizations and persons, on the other hand, have specific naming rules. Sometimes the REFDEF or NODET properties have to be assigned by PP-Memory to block the generation of strings such as "the Lake Michigan" or "Caspian Sea."

7.4.3 Links

The final class of abstract places comprises the "links" which connect towns together. The most obvious examples are the various kinds of roads. Examples of the PP-class #LINK include:

```
"Interstate 91"

 LINK0:
  CLASS (#LINK)
  LINKTYPE (*ROAD*)
  LINKNUMBER (91)
  ROADTYPE (INTERSTATE)

"southbound tracks"

 LINK1:
  CLASS (#LINK)
  LINKTYPE (*TRAINTRACK*)
  DIRECTION (SOUTH)

"ship channel"

 LINK2:
  CLASS (#LINK)
  LINKTYPE (*SHIPLINK*)
```

Links typically appear in the VIA slot in PTRANS Conceptualizations. For example, we may have "John took Route 44 to Providence."

AD-A056 080 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 6/4
SCRIPT APPLICATION: COMPUTER UNDERSTANDING OF NEWSPAPER STORIES--ETC(U)
JAN 78 R E CULLINGFORD N00014-75-C-1111
UNCLASSIFIED RR-116 NL

3 OF 3
ADA
056080

END
DATE
FILMED
8 -78
DDC

7.5 Miscellaneous PPs

In addition to the above, SAM also makes use of a small number of "abstract" PP-classes to take care of complex entities which occur in several different Scripts. The paramount example is "money," which is useful everywhere. For example, "five dollars" would be represented as:

```
(#MONEY TYPE (*CASH*) AMOUNT (5US))
```

and "a twenty-dollar traveler's check" would be:

```
(#MONEY TYPE (*TCHECK*) AMOUNT (20US)
        SOURCE (#ORGANIZATION ORGOCC ($BANK)))
```

This representation marks the traveler's check as having been obtained by execution of one track of the bank Script. A Master Charge card would look like:

```
(#MONEY TYPE (*CREDIT*)
        SOURCE (#ORGANIZATION ORGOCC ($CREDIT)
                              ORGNAME (MASTER CHARGE)))
```

Another abstract PP is a "meal," which is characterized by parts, each one of which is a food of a different kind. So, for a "hamburger and coke," we would have:

```
(#MEAL DRINKPART (#PHYSOBJ TYPE (*SODA*))
       EATPART (#PHYSOBJ TYPE (*HAMBURGER*)))
```

The final type of abstract PP used by SAM is the "contract," a high-level agreement entered into by polities. A military treaty is an example of this PP-class, as is:

```
"the US-Japan Economic agreement"
(#CONTRACT TYPE (*ECONOMY*)
       MEMBER (#POLITY "US")
       MEMBER (#POLITY "Japan"))
```

Chapter 8
Finale

"Begin at the beginning " the King said, very gravely, "and go on till you come to the end: then stop."
Lewis Carroll, Alice's Adventures in Wonderland.

8.1 Why Did We Do This?

Why have we done what we did? First of all there was the practical desire to have a working story understander. Before SAM there had been no systematic attempt to deal with the problems inherent in understanding "real" stories. Text-processing systems in existence (e. g., [32] and [42]) were designed to understand isolated sentences or to establish references between pairs of sentences in some sort of null context. With SAM on the other hand, the length of the story is not really a problem. It's as easy to read ten sentences as two provided the Script contains sufficient detail.

We have said repeatedly that SAM embodies a theory of context: what one is and how it can be used in understanding. In the beginning, we looked at simple contexts. We started SAM off with simple, made-up stories, about eating in restaurants and riding in subways. None of these early stories were of a kind that you'd want to tell a friend, or even a child. Getting SAM to do them, however, taught us what the essential problems in story understanding were. The techniques we developed for doing pronominal reference, making causal connections, using time- and place-setting, and organizing expectations went over essentially unchanged when we decided to adapt SAM to reading newspaper stories. So, in a sense, understanding a text about a car accident or a state visit is no harder than reading about a bus ride. The context is bigger, but not essentially different.

Because of our experiences with newspaper stories, we are convinced that SAM is that rarity among Artificial Intelligence programs, an extensible system. Each Script that we make up for a new domain we want to do is built in exactly the same way as previous ones. When a new Script is added to SAM, it is applied just as the others were. The Script-managing methods we evolved appear to make retrieval of the Script the system needs reasonably efficient.

This is not to say that SAM is a practical system. It is slow and, as is the case with multi-module programs, fragile. It is slow, however, because it tries to do the job right, to understand every word it reads, to a depth that makes it possible to generate outputs which a human would find reasonable and appropriate. Fragility, we feel, will be an inescapable problem with the advanced AI systems of the future. In a research effort directed at an area as large as text understanding, the only reasonable way to proceed is to design a system which consists of a community of experts, each accessing its own specialized knowledge base as it tries to contribute to the problem at hand. (This fact has

already been discovered in speech-processing research. Consider, for example, the "knowledge sources" of the Hearsay II system [12].)

8.2 What Else Could Be Done?

8.2.1 A Laboratory for Inference

SAM is an expectation-based understander. Having read some text, it makes predictions about what it will see next. An input, however, very frequently is not what was expected, exactly. In Chapter 5 we described a collection of inference processes designed to iron out discrepancies between "equivalent" concepts. We feel we have only scratched the surface of this problem here. There is a vast array of low-level inferences which people seem to make, not even knowing they make them, as they read things, which a text understander will also have to make if it is to arrive at the same conclusions.

As an example, consider the following two sentences:

(8.1) A passenger train hit a freight train in northern Mexico.
(8.2) Mary sent a letter to John in northern Mexico.

The question here is, what exactly does the phrase "in northern Mexico" locate? In (8.1) it seems clear that the collision between the trains took place in northern Mexico, although a prudent analyzer would assert only that "a freight train" is in northern Mexico. In my reading of (8.2), we probably shouldn't infer any more than that "John" is in northern Mexico. When should inferences about locations be made, and when not? Can we do it on the basis of the primitive ACT involved? For example, the PPs involved in a PROPEL, as in (8.1), must be physically near each other -- we don't allow action at a distance. On the other hand, given the existence of telephone, telegraph, radio and the postal service, it seems dangerous to assert that PPs involved in MTRANS and PTRANS ACTs, as in (8.2), are anywhere near each other.

Questions like these are part of ongoing research. The point that needs to be made is that a program like SAM is a natural facility, first for finding out what inferences seem to be needed, then for figuring out how to get them done. In this sense, SAM has served, and will continue to serve, as a _laboratory for inference_. The program forces the researcher to face up to a problem he didn't know was there. Then, when the methods he devises to cope with the problems that come up become unwieldy, it forces him to do the job right. This is because SAM is a complete system. It tries to do the whole job, from natural-language input to output. Problems that can be ignored in a system which works with hand-coded input, or which never attempts to express its results, have sometimes spectacular consequences for a complete system.

8.2.2 A Model of Reading

At each point in the design of SAM we tried to use our intuition as to how people read stories to decide on how SAM should. Our idea was to build an understander that was not only effective at what it does, but also a reasonable model of a person reading stories.

We have only partly succeeded in this effort. SAM understands in a three-stage process. First, it constructs an inference-free meaning representation for a sentence, then it makes whatever inferences it can about the PPs in the representation, then it tries to find the Conceptualization in a Script. There is no evidence that people operate in the same way. Indeed, a person seems to have absorbed the meaning of a sentence and integrated it into what he knows about the world as soon as he finishes reading.

If we want to make SAM a better model of a person who is carefully reading a newspaper, we need (among other things) to integrate the analysis process more closely with the memory processes responsible for identifying PPs and applying Scripts. While we cannot completely define at the moment what "integration" means, there are two places where closer cooperation seems possible.

First, in the analysis of noun groups, the analyzer and PP-Memory could interact to try to identify the referent of a noun group as the analysis process is ongoing Suppose, for example, we were reading:

(8.3) Chairman Hua Kuo-Feng of the People's Republic of China denounced the Soviet Union in the United Nations.

"Chairman Hua Kuo-Feng" forms a natural noun group, based on a name, for which a referent could be sought. Assuming PP-Memory has the permanent token for "Chairman Hua," it could instruct the analyzer on what it should expect to see next. For example, the Chairman's "occupation" may be mentioned, either as head of Communist China or of the Chinese Communist Party. These expectations, arising from an identification of the PP by PP-memory, would assist the analyzer in the analysis of the prepositional phrase "of the People's Republic."

A second memory interaction could take place when "denounced" is analyzed. Now we have the "kernal" of a complete Conceptualization:

```
((ACTOR HUMO <=> (*MTRANS*)
  MOBJECT ((CON ((ACTOR (NIL) <=> (*DO*)))
                LEADTO
              ((ACTOR HUMO TOWARD (*ANGER* VAL (6))))))))
```

It might be possible to rule out a number of Scripts with this kernal, in the case where no Script is active. If an appropriate Script, say, $VIPVISIT is available, it could make predictions, based in what it has already read, about further things to be seen. For example, it might specify "United States" or "Russia" for the as yet unspecified ACTOR Hua denounced. If it had read that Hua was in New York City, it might

predict "the UN" as a possible place for the "denouncing" to happen.

8.2.3 Scripts and Plans

Script-based processing is by no means the whole story in text understanding. SAM, as a top-down understander, can only comprehend what is contained in one of its Scripts, that is, a story about commonplace activities in a rote, stereotyped knowledge domain. It has no way to learn new Scripts, and no way to cope with the unexpected. People, however, operate quite handily in situations where the things that happen are only partly predictable. They can also read stories about such situations.

One important class of story problems which SAM cannot handle are those which arise out of people's desires and conflicts. Here we have to find ways to interpret people's actions in terms of their underlying motivations The work of Schank and Abelson [34] represents one AI approach to this problem, based on a small set of very general procedures called Plans. Plans characterize people's standard desires and and their preferred methods for satisfying them. Using Plans it is possible to make the appropriate connections in stories about people pursuing goals. An early version of Plan theory is implemented in a story understander called PAM [41].

It seems clear that a more powerful story understander than any presently in existence could be built from some combination of Script- and Plan-based processing We don't yet know how to build such an understander. We can say at least this much, however. What Script Applying could contribute to such an understander is knowledge of context: who the characters are, what the situation is they are acting in, what things in a story are explained by the Script.

To see how this could help, consider the following story:

> (8.4)
> Spillane walked into the Cafe Budapest and sat down. While the waiter was taking his order, the owner, a notorious Mafia figure, approached them threateningly.

The first sentence of course, refers to our friend $RESTAURANT. The beginning of the second sentence continues on an accepted chain of happenings, but the part about the menacing approach of the owner is not explained by the Script. Here we need a higher order of understanding, which knows what threats are for, to make sense out of this. and to predict what may happen next. What the Script could do for such a processor is to fill in the context that the story is going on in: a dining room in a Hungarian restaurant, owned by a person who is also a Mafioso Because it is a restaurant, rather than an alley, this understander should probably predict that harsh words between the owner and Spillane are about to be seen. To make this prediction. the processor would have to know what the referent of "them" is. This is an answer the Script could provide because of the ordering event it has already seen. The Script could also supply the information that waiters are an accepted role in restaurants, and that they are not customarily

threatened by their bosses.

The lesson here, and the message this thesis has consistently tried to convey, is that the process of comprehension and prediction is informed at every point by reference to context. In SAM we have tried to pin down this slippery notion, to give it substance in the episodic context that Scripts can provide. If we've succeeded, we've brought the long-standing dream of a non-human, but human-like intelligence a step closer to reality.

Appendix 1
Representation and Notation

A.1.1 Introduction

This Appendix presents a brief discussion of Conceptual Dependency (CD) meaning representation as it is used in the SAM system. In addition to the primitive ACTs of CD [32], we describe the types of causal relations between events [34] and the syntax of CD representation in LISP linear format.

A.1.2 Conceptual Dependency, Briefly

Conceptual Dependency (CD) is a system for representing the meaning of sentences in terms of a small number of "primitive" ACTions and STATEs. The class of sentences which the theory is intended to handle are those that refer to the "overt" physical and mental activities of human actors, and the "stative" attributes of these actors which the actions can affect. The fundamental assumption that the theory makes is that cognitive processing of sentences and text is language-free, that is, the memory processes operate on the meaning of sentences, not on the lexical expression of that meaning. Therefore, sentences which have the same underlying meaning will have the same representation in memory, although the lexical strings encoding that meaning are different.

Conceptual Dependency theory constitutes the basic framework for the computer simulation of human text processing which is the subject of this thesis. First of all, it provides the representational system which the analyzer uses to map surface strings into their underlying "thoughts," or Conceptualizations. Secondly, the processes of inference and memory search are facilitated because the elements of the memory store are stated in the same terms as the input Conceptualizations. The classes of inference, in particular, bear a well-defined relation to the primitive ACTs appearing in Conceptualizations. (This point was discussed Chapters 4 and 5.) Finally, the existence of a small class of primitives simplifies the task of generation, since the selection of the surface verb is made through a discrimination net tied to each of the ACTs.

CD representation is discussed in detail in [32]. However, since it is so important for the discussion to follow, a brief description of the primitive ACTs is given below. Each ACT has a conceptual case-frame with a required actor slot, and optional object, directional/recipient, and instrumental cases.

ATRANS

The transfer of ownership, possession, or control. ATRANS takes an actor, object, source, and recipient. "John gave Mary the book" is an ATRANS with actor "John," object "book," source "John," and recipient "Mary." "John got the book" is an ATRANS with actor "John," object "book," and recipient "John."

PTRANS

The transfer of physical location. PTRANS requires an actor, object, origin, and destination. "John ran to town" is a PTRANS with "John" as actor and object, and "town" as destination.

MTRANS

The transfer of information. An MTRANS can occur between animate entities or between memory locations within a human. Human memory is partitioned into three mental locations: the CP (Conscious Processor) which holds information we are consciously aware of; the IM (Intermediate Memory) where information from the immediate context is held for potential access by the CP; and the LTM (Long Term Memory) where information is stored permanently. MTRANS requires an actor, object, source, and recipient. Sources and recipients are either animals or mental locations in a human. "Tell" is an MTRANS between people, "see" is an MTRANS from eyes to the CP, "remember" is an MTRANS from the LTM to the CP, and "learn" is an MTRANS to the LTM.

MBUILD

The thought process which constructs new information from old. MBUILDs take place within the IM, receiving input from the CP and placing output in the CP. "Decide," "conclude," "imagine" and "consider" are all instances of MBUILD.

PROPEL

The application of a physical force. If movement takes place because of a PROPEL, then a PTRANS occurs as well as a PROPEL. PROPEL requires an actor, object, origin, and direction. "Push," "pull," "throw" and "kick" are all actions which involve a PROPEL.

INGEST

The internalization of an external object into the physical system of an animal. INGEST takes an actor, object, origin, and destination. "Eat," "drink," "smoke" and "breathe" are common examples of INGEST.

EXPEL

The act of pushing an object out of the body. EXPEL requires an actor, object, origin and destination. Words for excretion and secretion are described by EXPEL. "Sweat," "spit," and "cry" are EXPELs.

Many ACTs require an "instrumental" action on the part of the actor. Instrumental ACTs provide the "means" by which the more important ACTs (i. e., the TRANS and MBUILD ACTs) are accomplished. The following ACTs are used primarily in instrumental Conceptualizations. Each of these ACTs requires an actor and object. MOVE requires an origin and destination as well.

MOVE

The movement of an animal involving some bodypart. MOVE is instrumental to actions like "kick," "hand" and "throw." It can also occur noninstrumentally as in "kiss" and "scratch."

SPEAK

Any vocal act. Humans usually perform SPEAKing actions as instruments of MTRANSing.

ATTEND

The act of focusing a sense organ toward some stimulus. ATTEND is almost always instrumental to MTRANS. E. g., "see" is an MTRANS from the eye to the CP with instrument ATTEND eye to object.

GRASP

The act of securing contact with an object. E. g., "grab," "let go" and "throw" each involve a GRASP or the termination of a GRASP.

A.1.3 Causal Chains

Causal Chains ([28], [34]) are sequences of Conceptualizations in which each Conceptualization is connected to the next one by a causal relation. The primitive ACTs of Conceptual Dependency have as their results one or more STATEs. These STATEs can in turn enable other ACTs. Causal chains, in their full form, consist of alternating ACTs and STATEs. Conceptual Dependency uses six different kinds of causal links.

1. Result Causation (abbreviated "r"):
   relates an event based on a "physical" CD ACT (e. g., PTRANS, ATRANS) to the immediate result of that event. For example:

John ATRANS book to Mary

r

Mary POSS book

2. Enable Causation (e):
relates a STATive Conceptualization to the physical CD ACT it is an enabling condition for. For example:

John IS in New York

e

John PTRANS from New York

3. Reason Causation (R):
relates a "mental" CD act (MTRANS, MBUILD) to a action which follows it. For example:

John MBUILD (John PTRANS from New York)

R

John PTRANS from New York

4. Initiate Causation (I):
relates an action to a mental ACT which follows it. For example:

John PTRANS to restaurant

I

John MBUILD (John PTRANS to table)

5. Leadto Causation (L):
connects events in a causal chain which is not fully expanded. Leadto Causation indicates that the intervening events are not being spelled out. For example:

John PTRANS to restaurant

L

John MTRANS to waiter
(John WANT (John INGEST meal))

6. Cancause Causation (C):
This is a version of Leadto Causation for hypothetical events, especially causally connected events which appear in the MOBJECT of an MTRANS. Like Leadto, the explicit causal connections between the events have not been given:

"John said if he went to a restaurant, he could sit down"

```
John MTRANS
        (John PTRANS to restaurant

                    C

         John MOVE John to chair)
```

7. Concatenated Causals:
describe connections between events in which certain intervening events or states have been left out. Unlike Leadto, the missing causal links are known. For example:

```
John PTRANS to table

        re

John MOVE to chair
```

indicates a missing STATE, John's being at the table.

A.1.4 CD in LISP Format

CD representations are often shown for purposes of exposition as two-dimensional graphs (see, e. g., [32]). However, to understand how the various functions in SAM work, and to read the outputs that are presented, it is necessary to understand how the CD graphs are represented in linear list format within the program. Conceptualizations and the PPs which appear in them are encoded in SAM in LISP data structures conforming to a well-defined syntax (Note 1). Giving the syntax, directly, however, isn't very enlightening. What we will do instead is to present a series of CD LISP structures of increasing complexity, pointing out the underlying structural features as we go along. In each case, we will be showing simulated ELI output, before tokenization by PP-Memory. The token-assignment process is related to the ELI form in a straightforward way, described in Chapter 4.

----

1. This discussion assumes some knowledge of the programming language LISP, which is widely used in AI applications. An elementary presentation of LISP, sufficient for understanding the examples in this thesis, is given in Winston [45].

The simplest CD entities are Picture Producers (PPs). As we indicated in Chapter 7, every PP handled by SAM must belong to one of the "primitive" PP-classes, and contain a subset of the predefined "slots" which are characteristic of the class. The PP "John Smith," which belongs to the class #PERSON, is defined by the following list structure, called a "PP-concept:"

```
(#PERSON FIRSTNAME (JOHN) LASTNAME (SMITH) GENDER (*MASC*))
```

In the above, the CAR of the list (called the "form" of the PP-concept) gives the primitive class, the CDR (called the "modifier" of the PP-concept) defines the slots of the class which the input has filled. The Modifier of a PP-concept consists of pairs, each pair being an (atomic) slotname and a "filler" for the slot. Every slotfiller is a list structure. Slotfillers come in several varieties. The simplest kind records essentially "surface" properties of a PP, such as FIRSTNAME and LASTNAME, which are stored, as in the above, just as they appeared in the input string. Another kind of slotfiller gives a feature of the PP selected out of a "contrast set," one member of which every PP belonging to the class must contain. A typical contrast set used by SAM is GENDER, which consists of (*MASC* *FEM* *NEUT*).

When an atom in a slotfiller is surrounded by "*," it may mean that the associated property is "primitive," that is, not analyzed further in the representational system. This is the case for the features of the GENDER class, and also of the primitive ACTs of Conceptual Dependency (*PTRANS*, *ATRANS*, etc.). The second possibility is that the filler points to a "feature-cluster" in PP-Memory, defining the Scripts the PP can have a role in. For example, "a car" defines the cluster:

```
(#STRUCTURE TYPE (*CAR*) REF (INDEF))
```

where the filler *CAR* corresponds to the possible places this PP can fit in a Script, for example:

```
*CAR*:
  SCRIPTROLES (($DRIVE . &VEHICLE)
               ($AUTOSHOWROOM . &SALE-OBJ))
```

In this case, the TYPE slot is a shorthand way of referring to all the possible functions the PP can have in a story.

Another possible slotfiller is a Script name. This is particularly the case for PPs of the class #ORGANIZATION, which nearly always have an associated Script:

```
"Metropolitan Museum"
(#ORGANIZATION ORGNAME (METROPOLITAN) ORGOCC ($MUSEUM))
```

A final type of filler is an imbedded reference to another PP. Imbedded PPs follow the same syntax as other PPs:

```
"John Smith, of New Haven"
(#PERSON FIRSTNAME (JOHN) LASTNAME (SMITH) GENDER (*MASC*)
         RESIDENCE (#POLITY POLNAME (NEW HAVEN)
                            POLTYPE (*MUNIC*)))
```

PP-concepts provide descriptions of Picture Producers, but say nothing about what the PP may be doing in the world. Full Conceptualizations describe either actual or potential events or states in which PPs may be involved. As such, Conceptualizations always contain a CD ACT or STATE (signalled by "<=>" or "IS," respectively). For example, we may have the ACT-Conceptualization:

```
"John went to New Haven on Friday"
((ACTOR PP1 <=> (*PTRANS*) OBJECT PP1
            FROM (NIL)
            TO (*PROX* PART (#POLITY POLTYPE (*MUNIC*)
                                     POLNAME (NEW HAVEN))))
  TIME (TIM1))

PP1: (#PERSON FIRSTNAME (JOHN) GENDER (*MASC*))
TIM1: ((WEEKDAY FRIDAY))
```

or the STATE-Conceptualization:

```
"John was angry"
((ACTOR PP1 IS (*ANGER* VAL (-3))))
```

As with PP-concepts, the CAR of a full Conceptualization is called its Form, the CDR its Modifier. In the ACT-Conceptualization given above, the Form describes the event itself, which is based on a PTRANS, and the Modifier places a time modification on it. Both the Form and Modifier parts of the Conceptualization consist of pairs of slots and fillers. Additionally, the fillers in a Form may contain references to PPs, either in list-structure format, or, as in the above, "by name." In the PTRANS-event, for example, both the ACTOR and OBJECT slot of the Form have the same filler, PP1. If a mandatory slot in the case-frame defined by an ACT happens not to be filled by a particular input, the associated filler will be empty (NIL).

Certain slots in the Form are filled by full Conceptualizations. Examples are INSTrumentals, and the MOBJECT of an MTRANS Conceptualization. Here's an example containing both cases:

```
"John told Mary to get out of the car"
((ACTOR PP1 <=> (*MTRANS*) FROM (*CP* PART PP1)
            TO (*CP* PART PP2)
            INST ((ACTOR PP1 <=> (*SPEAK*)))
            MOBJECT ((ACTOR PP2 <=> (*PTRANS*) OBJECT PP2
                             TO (NIL)
                             FROM (*INSIDE* PART
                                   (#STRUCTURE TYPE (*CAR*)
                                               REF (DEF)))))))
```

```
PP1: (#PERSON FIRSTNAME (JOHN) GENDER (*MASC*))
PP2: (#PERSON FIRSTNAME (MARY) GENDER (*FEM*))
```

Fillers in the Modifier part of a Conceptualization customarily refer to an aspect of the event, and will not contain PPs. If the event in question is negated, or only potential, this is indicated through the MODE slot in the modifier. For example, we may have:

```
"John can't go"
((ACTOR PP1 <=> (*PTRANS*) OBJECT PP1
            TO (NIL) FROM (NIL))
  MODE (MOD1))

PP1: (#PERSON FIRSTNAME (JOHN) GENDER (*MASC*))
MOD1: (*CANNOT*)
```

Finally, Conceptualizations may contain sub-Conceptualizations connected together by causal links. In this case, the causal antecedent is associated with a CONcept slot, the successor with a slot naming the type of causal relation. For example, we may have:

```
"John went because he was angry"
((CON
    ((ACTOR PP1 IS (*ANGER* VAL (-3))))
  LEADTO
    ((ACTOR PP1 <=> (*PTRANS*) OBJECT PP1
                TO (NIL) FROM (NIL)))))

PP1: (#PERSON FIRSTNAME (JOHN) GENDER (*MASC*))
```

Appendix 2
Other Example Stories

As SAM evolved, it became capable of handling more and more complex stories, progressing from simple texts referring to single Scripts, through stories which accessed several Scripts in a simple fashion, to edited versions of newspaper articles in which complicated situations are described. This section presents examples of both "made-up" and "real" stories. Stories (A.1) through (A.4) were processed by an early version of SAM just as shown. The remaining stories are typical of what SAM is currently equipped to read. Some of these texts have not been parsed by ELI, but the rest of SAM produced the summary and question-answering results given on the basis of simulated ELI output.

Here is a story making reference to a common Script, $RESTAURANT, in which the Script is stopped midway, or "aborted":

Story A.1
JOHN WENT TO A RESTAURANT. HE SAT DOWN. HE GOT MAD. HE LEFT.

Long Paraphrase:
JOHN WAS HUNGRY. HE DECIDED TO GO TO A RESTAURANT. HE WENT TO ONE. HE SAT DOWN IN A CHAIR. A WAITER DID NOT GO TO THE TABLE. JOHN BECAME UPSET. HE DECIDED HE WAS GOING TO LEAVE THE RESTAURANT. HE LEFT IT.

Here, SAM has inferred that John got mad because no one came to take his order, as sometimes happens in restaurants. (A.2) has more complex examples of "interfering" events which can happen there:

Story A.2
JOHN WENT TO LEONE'S. 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.

Question-Answering:
Q1: DID JOHN ORDER A HOT DOG?
A1: YES.

Q2: DID JOHN EAT A HOT DOG?
A2: NO THE WAITER TOLD JOHN THE MANAGEMENT WAS UNABLE TO GIVE IT TO HIM.

Q3: WHAT DID THE WAITER SERVE JOHN?
A3: THE WAITER SERVED JOHN A HAMBURGER.

Q4: WHY DIDN'T JOHN EAT THE HAMBURGER?
A4: BECAUSE THE HAMBURGER WAS OVERDONE.

Q5: WAS JOHN ANGRY?
A5: YES.

Q6: DID JOHN PAY THE CHECK?
A6: NO JOHN WAS ANGRY BECAUSE THE HAMBURGER WAS OVERDONE AND SO HE LEFT LEONE'S.

By contrast, (A.3) is a typical boring text about eating in a restaurant which follows the Script closely

Story A.3:
JOHN WENT INTO A RESTAURANT. THE HOSTESS SEATED JOHN. THE HOSTESS GAVE JOHN A MENU. JOHN ORDERED A LOBSTER. HE WAS SERVED QUICKLY. HE LEFT A LARGE TIP. HE LEFT THE RESTAURANT.

Short Paraphrase:
JOHN WENT TO A RESTAURANT AND HE ATE A LOBSTER.

Question-Answering:
Q1: WHY DID THE HOSTESS GIVE JOHN A MENU?
A1: SO JOHN COULD ASK THE WAITER FOR A MEAL.

Q2: WHY DID JOHN GO TO A RESTAURANT?
A2: SO JOHN COULD EAT A MEAL.

Q3: DID THE WAITER GIVE JOHN A MENU?
A3: JOHN GOT THE MENU FROM THE HOSTESS.

Q4: WHY DID JOHN LEAVE THE WAITER A LARGE TIP?
A4: JOHN REMEMBERED THE WAITER SERVED A LOBSTER TO HIM QUICKLY.

Q5: DID JOHN PAY THE BILL?
A5: JOHN PAID THE CHECK.

Under control of the global Script Situation $TRIP, (A.4) calls up several Scripts, $BUS, $SUBWAY and $RESTAURANT, following a relatively rare path in one Script because of a strange event in an earlier one:

Story A.4:
JOHN WENT TO NEW YORK BY BUS. ON THE BUS HE TALKED TO AN OLD LADY. WHEN HE LEFT THE BUS, HE THANKED THE DRIVER. HE TOOK THE SUBWAY TO LEONE'S. ON THE SUBWAY HIS POCKET WAS PICKED. HE GOT OFF THE TRAIN AND ENTERED LEONE'S. HE HAD SOME LASAGNA. WHEN THE CHECK CAME, HE DISCOVERED HE COULDN'T PAY. THE MANAGEMENT TOLD HIM HE WOULD HAVE TO WASH DISHES. WHEN HE LEFT THE RESTAURANT, HE RETURNED TO NEW HAVEN.

Short Paraphrase
(Chinese Translation using Harvard Romanization)·
JOHN TZUOH CHARNG TWU CHIH CHE DAW NEOU LUE CHIUH LE. TA TZAY CHARNG TWU CHI CHE LII. TA GEN I WEY LAO TAY TAY TARN I TARN. TA TSORNG CHARNG TWU CHIH CHE LII CHIUH CHUH LE. TA SHIEH SHIEH SY JI. TA TZOUH DIH SHIAH HUOO CHE DAW LEONE'S CHIUH LE. TA TZAY DIH SHIAH TIEE LUH CHE LII. I GE PAR SHOOU TOU LE CHYAN. TA TSORNG DIH SHIAH TIEE LUH CHE CHUH CHIUH LE. TA JINN LE

LEONE'S. TA CHR LE I DA LI TSAY. TA TSORNG CHAR FARNG NADAW I GE JANQ DHAN. TA FA SHYANN TA BU KEE YII FU CHYAN. CHAR FARNG JIAW TA BIH SHIU YAW SHII BEI TZ. TA TSORNG FANG GUAR LII CHU CHIUH LE. TA TZUOH CHARNG TU CHI CHE DAW SHIN GANG CHIUH LE.

Question-Answering:
Q1: WHO WENT TO NEW YORK?
A1: JOHN WENT TO NEW YORK.

Q2: WHY DID JOHN GO TO NEW YORK?
A2: BECAUSE JOHN WANTED TO GO TO LEONE'S.

Q3: WHY DID JOHN GO TO LEONE'S?
A3: BECAUSE JOHN WANTED TO HAVE SOME LASAGNA.

Q4: HOW DID JOHN GET TO NEW YORK?
A4: JOHN TOOK A BUS TO NEW YORK.

Q5: HOW DID JOHN GET TO LEONE'S?
A5 JOHN TOOK A BUS TO NEW YORK AND THEN HE TOOK THE SUBWAY TO LEONE'S.

Q6: WHAT DID JOHN EAT?
A6: JOHN HAD SOME LASAGNA.

Q7: DID JOHN ORDER LOBSTER?
A7: NO JOHN ORDERED SOME LASAGNA.

Q8: DID JOHN PAY THE CHECK?
A8 NO JOHN DIDN'T HAVE ANY MONEY.

Q9: DID JOHN LEAVE THE WAITRESS A TIP?
A9: NO JOHN DIDN'T HAVE ANY MONEY.

Q10: WHY DIDN'T JOHN PAY THE CHECK?
A10 JOHN DISCOVERED THAT HE DID NOT HAVE ANY MONEY AND SO HE HAD TO WASH DISHES.

Q11: WHY DIDN'T JOHN HAVE ANY MONEY?
A11: BECAUSE A THIEF PICKED JOHN'S POCKET.

The next example is a slightly edited version of a train wreck story from the New York Times:

Story A.5:
A passenger train carrying tourists collided with a freight train today in northern Mexico. 17 persons were killed and 25 persons were injured.

Summary:
A PASSENGER TRAIN HIT A CARGO TRAIN IN NORTHERN MEXICO ON TUESDAY. EIGHTEEN PEOPLE DIED. TWENTY-FIVE PEOPLE WERE HURT.

Question-Answering:
Q1: How many people were injured?
A1 TWENTY-FIVE.

Q2: How many people were killed?
A2: EIGHTEEN.

Q3: Why did the people die?
A3: BECAUSE A PASSENGER TRAIN HIT A CARGO TRAIN.

The remaining stories have not been parsed by ELI but the rest of the system produced the outputs shown on the basis of simulated ELI output. Story A.6 is an actual account of a motorcycle crash from the New Haven Register. This story was processed under control of the Script Situation $VEHACCIDENT:

Story A.6:
Alan L Plucinski age 17, of Fairfield, died Sunday morning in a motorcycle accident when the cycle he was driving crashed into a utility pole on Dalton Woods Road Plucinski was pronounced dead on arrival at Milford Hospital, police said.

Summary:
A CYCLE HIT A POLE NEAR DALTON WOODS ROAD TWO DAYS AGO. ALAN PLUCINSKI, AGE 17, RESIDENCE IN FAIRFIELD, CONNECTICUT DIED.

The phrase "two days ago" is inserted by SAM because it is arranged, by convention, to be reading the newspaper on Tuesday, in New Haven, Connecticut.

Here is another accident story, this time about a car falling into a river.

Story A.7:
Jane Jones died yesterday of multiple head injuries which she suffered in an accident on Thursday. Her car fell into the Connecticut River near Route 80.

Summary:
AN AUTOMOBILE FELL INTO THE CONNECTICUT RIVER FROM A BRIDGE NEAR HIGHWAY 80 FIVE DAYS AGO. JANE JONES WAS CRITICALLY INJURED. SHE DIED YESTERDAY.

Story A.8 is a final car accident report containing an unexpected event:

Story A.8:
A Pennsylvania man and his wife returning from a Cape Cod camping vacation were killed in a violent crash on the Connecticut Turnpike Saturday morning. The victims have been identified as John Gavin, 47, and his wife Mary, 41, of Morristown, Pa. Both were pronounced dead at the scene by a medical examiner. Mrs Gavin's son 19-year-old son, Grant Butler, miraculously escaped injury in the one-car accident.

Summary:
AN AUTOMOBILE HIT SOMETHING NEAR THE CONNECTICUT TURNPIKE THREE DAYS AGO. JOHN AND MARY GAVIN, RESIDENCE IN MORRISTOWN, PENNSYLVANIA, DIED. UNEXPECTEDLY, GRANT BUTLER, AGE 19, WAS NOT HURT.

The final example is a slightly edited version of a story about an oil spill from the New York Times:

Story A.9:
A Liberian tanker ran aground off Nantucket Island yesterday, the Coast Guard said. The ship broke up and dumped 1 million gallons of crude oil into the Atlantic.

Q1: What was the tanker carrying?
A1: OIL

Q2: Where was the oilspill?
A2: NEAR NANTUCKET ISLAND

BIBLIOGRAPHY

1 Abelson R. P. (1969). Psychological Implication. in Abelson, R. P. et al. (eds.). Theories of Cognitive Consistency. Rand-McNally, New York

2. Bar-Hillel, J. (1964). Language and Information. Addison-Wesley, Reading, Massachusetts.

3. Bartlett, R. (1932). Remembering: A Study in Experimental and Social Psychology. Cambridge University Press London.

4. Bower, G. H. (1976) Comprehending and Recalling Stories. Div. 3 Presidential Address, Washington: American Psychological Association, Sept 6. 1976

5. Bransford, J. D. and Franks, J. J. (1971). The Abstraction of Linguistic Ideas. Cognitive Psychology, Vol 2, pp. 331-350.

6. Brown, J. S., and R. R. Burton. Multiple Representations of Knowledge for Tutorial Reasoning, D. G. Bobrow and A. Collins (ed.). Representation and Understanding. Academic Press, New York.

7. Charniak, E. (1972). Towards a Model of Children's Story Comprehension. (thesis) AITR-266 M.I.T. Cambridge, Mass.

8. Charniak, E. (1975). Organization and Inference in a Frame-like System of Common Knowledge. Proceedings from Theoretical Issues in Natural Language Processing. Cambridge, Mass.

9. Charniak. E. (1978). A Framed Painting -- Representation of a Commonsense Knowledge Fragment. Cognitive Science, Vol. 2, No. 1, (in press)

10. Cullingford, R. E. (1975). An Approach to the Representation of Mundane World Knowledge: The Generation and Management of Situational Scripts. American Journal of Computational Linguistics. Microfiche #44.

11. Cullingford R. E. (1976). The Uses of World Knowledge in Text Understanding. Proceedings of the Sixth International Conference on Computational Linguistics. Ottawa, Canada.

12. Erman L. D., and V. R. Lesser (1975). A Multi-Level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge. Proceedings of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi. USSR.

13. Gershman, A. (1977). Analyzing English Noun Groups for Their Conceptual Content. Computer Science Research Report No. 110 Yale University.

14. Goldman, N. M. (1975). Conceptual Generation. _Conceptual Information Processing_. Ed: Schank. North-Holland, Amsterdam.

15. Hewitt C. (1970) PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot. MIT AI Memo 168 (revised).

16. Johnson-Laird, P. (1974). Memory for Words, _Nature_, Vol. 32, No. 3.

17. Kintsch, W., and D. Monk (1972). Storage of Complex Information in Memory: Some Implications of the Speed with Which Inferences Can Be Made. _Journal of Experimental Psychology_, 94.

18. Lehnert, W. (1977). The Process of Question Answering. (thesis). Department of Computer Science Report No. 88, Yale University.

19. Minsky, M. (1975). A Framework for Representing Knowledge. _The Psychology of Computer Vision_. Ed: P. H. Winston. McGraw-Hill, New York.

20. Norman. D. A. and Rumelhart, D. E. (1975). _Explorations in Cognition_. W. H. Freeman and Co., San Francisco.

21. Quillian, M. R. (1968). Semantic Memory. _Semantic Information Processing_. Ed: Minsky. MIT Press Cambridge, Mass.

22. Rieger, C. (1975a). Conceptual Memory. _Conceptual Information Processing_. Ed: Schank. North-Holland, Amsterdam.

23. Rieger, C. (1975b). The Commonsense Algorithm as a Basis for Computer Models of Human Memory, Inference, Belief and Contextual Language Comprehension. _Proceedings from Theoretical Issues in Natural Language Processing_. Cambridge, Mass.

24. Riesbeck, C. (1975). Conceptual Analysis. _Conceptual Information Processing_. Ed: Schank. North-Holland Amsterdam.

25. Riesbeck, C. and Schank, R. (1976). Comprehension by Computer: Expectation-Based Analysis of Sentences in Context. Research Report #78. Department of Computer Science, Yale University, New Haven, Ct.

26. Riesbeck. C. (1977). Delayed Interpretation Using Processing Notes. _Proceedings of the Fifth International Joint Conference on Artificial Intelligence_. Cambridge, Massachusetts.

27. Rumelhart, D. E. (1976). Understanding and Summarizing Brief Stories. In D. LaBerge and S. J. Samuels (eds). _Basic Processes in Reading: Perception and Comprehenson_. Lawrence Erlbaum Associates, Hillsdale, New Jersey.

28. Schank, R. C. (1973). Causality and Reasoning. Technical Report #1. Istituto per gli Studi Semantici e Cognitivi, Castagnola, Switzerland.

29. Schank, R. C. (1974a). Adverbs and Belief. Lingua. Vol. 33, No. 1, pp. 45-67

30. Schank, R. C. (1974b). Understanding Paragraphs. Technical Report #6. Istituto per gli Studi Semantici e Cognitivi. Castagnola, Switzerland.

31. Schank. R. C. (1975a). Is There a Semantic Memory. Technical Report #8. Istituto per gli Studi Semantici e Cognitivi, Castagnola, Switzerland.

32. Schank, R. C. (1975b). Conceptual Information Processing. (ed.) North-Holland, Amsterdam.

33. Schank, R. C. (1975c). The Structure of Episodes in Memory. Representation and Understanding: Studies in Cognitive Science. Eds· Bobrow and Collins. Academic Press, New York.

34. Schank, R. C. and Abelson, R. P. (1977). Scripts, Plans, Goals and Understanding. Lawrence Erlbaum Assoc., Hillsdale, N.J.

35. Schank, R. C. and Colby, K. M (1973). Computer Models of Thought and Language. W. H. Freeman and Co. San Francisco.

36. Schank, R. C. and Yale A.I. Project (1975). SAM -- A Story Understander. Research Report #43. Department of Computer Science, Yale University, New Haven, Ct.

37. Schmidt, C. F., et. al. (1976). Recognizing Plans and Summarizing Actions. Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior, pp. 291-306, Edinburgh, Scotland.

38. Scragg, G. W. (1975). Answering Questions about Processes. Explorations in Cognition Eds. Norman and Rumelhart. W. H. Freeman and Co. San Francisco.

39. Stutzman, W. J. (1976). Organizing Knowledge for English-Chinese Translation, Proceedings of the Sixth International Conference on Computational Linguistics. Ottawa, Canada.

40. Tulving, E. (1972). Episodic and Semantic Memory. Organization of Memory. Eds: Tulving and Donaldson. Academic Press, New York.

41. Wilensky, R. (1976). Using Plans to Understand Natural Language. Proceedings of the Annual Conference of the ACM. Houston, Texas.

42. Wilks, Y. (1976). A Preferential, Pattern-Seeking, Semantics for Natural Language Inference. Artificial Intelligence, Vol. 6.

43. Winograd, T. (1972). Understanding Natural Language. Academic Press, New York.

44. Winograd, T. (1975). Frame Representations and the Declarative-Procedural Controversy. _Representation and Understanding_. Eds: Bobrow and Collins. Academic Press. New York

45. Winston P. H. (1977). _Artificial Intelligence_. Addison-Wesley, Reading, Massachusetts.

46. Woods W., _et. al._ (1972). The Lunar Sciences Natural Language Information System: Final Report, Report No. 2378, Bolt, Beranek and Newman, Cambridge, Massachusetts.