An Organization for Programs in Fluid Domains

Richard P. Gabriel

A Dissertation Submitted to the Department of Computer Science
and the Committee on Graduate Studies of Stanford University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

December 1980

Contents

1 Introduction **3**
- 1.1 What is Yh? . . . 4
- 1.2 Complexity versus Simplicity . . . 5
- 1.3 Co-Evolution . . . 6
- 1.4 The Goals of Artificial Intelligence . . . 6
- 1.5 Ad Hoc to Epistemology to Ontology . . . 7
- 1.6 What are the Goals of AI? . . . 10
- 1.7 Intelligence and Communication . . . 11
- 1.8 What the Domain is About in Outline . . . 12
- 1.9 Fluid versus Essential Domains . . . 13
- 1.10 Broad History of the Domain . . . 14
- 1.11 Detailed History of the Domain . . . 15
  - 1.11.1 Generating Sentences in the Language . . . 16
  - 1.11.2 Generating Sentences that Mean Something . . . 16
  - 1.11.3 More Recent Attempts . . . 16
  - 1.11.4 Goldman . . . 17
- 1.12 What the Domain is about in Detail . . . 17
- 1.13 A View on Program Behavior . . . 18
- 1.14 Personal History of the Problem . . . 19
  - 1.14.1 A Complex System Matures . . . 20
- 1.15 Overview of the Generation Process . . . 22
  - 1.15.1 More Details on Generation Step . . . 22
  - 1.15.2 What about the Text? . . . 23
  - 1.15.3 What to do if Nothing Matches . . . 24

2 Natural Language Generation **25**
- 2.1 The First Pass . . . 25
- 2.2 The Second Pass . . . 27
- 2.3 Multiple versus Single Sentence Utterances . . . 28
  - 2.3.1 Single from Multiple Sentences . . . 28
- 2.4 The Generation Process . . . 29
- 2.5 Vertical Strategy . . . 30
  - 2.5.1 Rich Domains . . . 30
- 2.6 Cohesion . . . 30
  - 2.6.1 Endophoric Reference . . . 31
  - 2.6.2 Exophoric Reference . . . 31
  - 2.6.3 Anaphoric Reference . . . 31

2.6.4 Cataphoric Reference . . . 31
2.6.5 Substitution . . . 32
2.6.6 Ellipsis . . . 32
2.6.7 Conjunctions . . . 32
2.6.8 Collocation . . . 33

3 Representation **34**
3.1 Stratification . . . 34
3.1.1 Shallow Stratifications . . . 34
3.1.2 Deep Stratifications . . . 35
3.1.3 Example . . . 35
3.1.4 Units . . . 35
3.1.5 Procedural and Declarative Components . . . 36
3.2 The Meaning of Slots . . . 37
3.3 Some Standard Units . . . 41
3.4 An Example from Yh . . . 41
3.5 Pattern Matching on Units . . . 43
3.6 Representation and Description . . . 44

4 The Text **45**
4.1 Stratified Surface Structure . . . 45
4.1.1 Motivation . . . 45
4.1.2 Initial Remarks and Disclaimer . . . 46
4.2 Features of the Representation . . . 47
4.2.1 Here . . . 47
4.2.2 Creating . . . 47
4.2.3 Moving . . . 48
4.2.4 Deleting . . . 48
4.2.5 Appending and Merging . . . 48
4.2.6 Setting and Getting Entries . . . 48
4.2.7 Getting and Setting the Current Position . . . 48
4.2.8 Matching . . . 48
4.2.9 Building from Matches . . . 50
4.2.10 Replacement . . . 50
4.2.11 More Match-based Operations . . . 51
4.2.12 Listifying Things . . . 51
4.2.13 Mapping . . . 52
4.3 Relative Clauses . . . 52
4.4 Dire Implementation . . . 52
4.5 Transformations . . . 52
4.5.1 Transformations in Detail . . . 53
4.5.2 Problems . . . 53
4.5.3 More Problems . . . 55
4.5.4 Range of Transformations . . . 56
4.6 A Big Example Transformation . . . 56
4.7 The Lexicon . . . 60
4.7.1 Details of the Lexicon . . . 61

4.8 Denouement 62

5 Planning **64**
5.1 Planning in Generation 64
5.2 Agenda 65
5.3 Description Versus Representation 66
5.4 System-goals 68
5.5 Summary of the Matching Process 69
5.6 Example of a System Description 70
5.6.1 The Real Blind Robot Problem 75
5.7 Summary of Examples 76
5.8 Planning and Sequencing 76
5.9 The Lesson 76

6 Influence-based Matching **78**
6.1 Planning Versus Action or Participation 78
6.2 Hybrid Descriptions 79
6.3 A Review of the Situation 80
6.4 Overall Goals of Matching 80
6.5 Unit Selection 81
6.5.1 Single Descriptors 82
6.5.2 Decisions and Thresholds 84
6.5.3 Multiple Descriptors 84
6.6 The Exact Process 86
6.6.1 The Basic Matcher 87
6.6.2 The Combinatorial Pairing Function 88
6.7 Numeric Matching Continued 89
6.7.1 Pairing 90
6.7.2 Notation 91
6.7.3 Computing the Evaluation for a Pairing 92
6.7.4 Pre-conditions 93
6.7.5 Added-goals 93
6.7.6 System Influences 94
6.7.7 Unit Influences 94
6.7.8 Preferences 94
6.7.9 Soft Constraints 95
6.7.10 Level Bonus 95
6.7.11 Exceptions 95
6.8 The Full Process 95
6.8.1 What If Nothing is Selected by Now? 98
6.9 Anomalies and Examples 98
6.9.1 Non-Constant Patterns 100
6.9.2 The Other Variants 102
6.10 Special Control Structures 103
6.10.1 Passing Pairs as Arguments 103
6.10.2 Optimal Calling of a Set of Goals 104
6.10.3 Optimal Calling of a Set of Entries 104

6.10.4 Example . . . 105
6.10.5 More Control . . . 105
6.11 How is This a Good Idea? . . . 105

7 Counterinduction **106**
7.1 The Situation . . . 106
7.2 An Unexpected Consideration . . . 107
7.3 A Minimal Violation of Rationality . . . 108
7.4 The Exact Process . . . 108
7.4.1 Rating the Situation Description . . . 109
7.4.2 Example . . . 110
7.4.3 The Process Continues . . . 110
7.4.4 The Simulation . . . 110
7.4.5 Selecting the Winner . . . 111
7.5 Tuning . . . 111
7.6 An Example . . . 111
7.7 Distinctions . . . 115
7.8 Final Note . . . 115

8 Example of Generation **116**
8.1 Context . . . 116
8.2 Dutch National Flag . . . 116
8.3 The Input . . . 118
8.4 The Ball Starts Rolling . . . 125
8.4.1 Conducting an Inspection . . . 127
8.4.2 Digression on Importance . . . 128
8.4.3 Undigression on Importance . . . 128
8.5 Simple Declaratives . . . 129
8.6 Noun Phrases . . . 130
8.6.1 Very Simple Noun Phrases . . . 131
8.6.2 Uniqueness of the Noun Phrase . . . 131
8.6.3 Adjectives . . . 132
8.6.4 Prepositional Phrases . . . 132
8.7 Verb Phrases . . . 132
8.8 The Array Lives . . . 132
8.9 Those Pesky Array Markers . . . 134
8.10 Note on Multiple Nouns . . . 135
8.11 Pondering the Issues . . . 135
8.12 The Fun Begins . . . 136
8.13 The Rest of the Paragraphs . . . 137
8.13.1 Speaking of Macros . . . 138
8.14 Alternate First Paragraphs . . . 138
8.14.1 A Weird Alternative . . . 138
8.15 Conclusion . . . 138

9 Conclusion **140**
9.1 Creativity and Discipline . . . 140
9.1.1 Two as One . . . 141
9.1.2 One as Two . . . 141
9.1.3 What the Thesis is About . . . 141
9.2 Underlying Language Issues . . . 142
9.3 Observation and Participation . . . 142
9.3.1 Descriptions and Stuntmen . . . 143
9.3.2 The Problem with Puns . . . 143
9.4 Representations and Distinctions . . . 144
9.4.1 Webs Versus Monoliths . . . 144
9.5 A Commitment to Nonsense . . . 145
9.6 Bondage and Influences . . . 146
9.6.1 Counterinduction and Rationality . . . 146
9.7 Cult of the Superintelligent . . . 147
9.8 The Final Curtain . . . 147
References . . . 148

What is this?

I wrote my dissertation in 1980 and typeset it in plain, unadorned Tex—only three years old at the time. I used a now-defunct operating system with a now-nonstandard character set, and I've lost most of the macros I wrote, macros that pre-dated Latex but were intended for similar purposes. My dissertation is not great, and no versions are available on the Net, so I decided to try to typeset it in a more modern way using LaTex but guessing what some of my macros were supposed to produce given only their uses (no definitions)—I have no hardcopy to look at, so my process was to guess what they did, look at the result, and then tweak. Because most readers likely care only about the gist of what I did, I didn't spend a lot of time working over the typesetting. In particular, I didn't try to 'modernize' the typesetting; I left it in the basic style I used in 1980.

Terry Winograd was my thesis adviser; I had joined an Automatic Programming project to do their natural language generation stuff. My thesis committee was Terry, David Waltz, and Cordell Green (head of the Automatic Programming project). The night before my oral exam, Terry called up and told me he planned to vote "no." Not great news. But another student in my incoming class (Ron Goldman) had helped me prepare my talk to an absurd level (something like 20 practice runs), so I passed without any grilling by the committee. You can read the full story in **Patterns of Software**, chapter "A Personal Narrative: Stanford."

Of personal importance is that the Computer Science Department at Stanford was in the School of Humanities and Sciences while I was a student there; my PhD is from that school. Later, the CS Department moved to the School of Engineering. I am not an engineer—I know next to nothing about engineering. In the early 2000s I was invited to an event where doctoral regalia was appropriate, but the official Stanford company that supplies it would only give me regalia for the School of Engineering—I had to skip the regalia. Every month I get an email newsletter from the Stanford School of Engineering—I've programmed my mail client to delete it. It's not that I dislike the Stanford School of Engineering or engineering—it's that engineers have deep and specific training, and I would no sooner appear to claim I'm an engineer than appear to claim to be a surgeon.

After graduating, John McCarthy read my dissertation; despite declaring "it's not very good," he hired me to work on his revived Advice Taker project, where the basis of research was this statement, "in order for a program to be capable of learning something it must first be capable of being told it."

Acknowledgements

I am deeply grateful to David Waltz, my good friend and mentor, who carefully and enthusiastically provided encouragement and happiness when my work and I needed it most: to say that the experience of doing this work and putting it together in this form has been a personal tragedy is an understatement.

Rod Brooks, Ron Goldman, and Bill Scherlis read drafts at a time when "reading" meant "decoding."

My deepest gratitude is to the SAIL system, which lived for many years in the warm, foothills setting above the Bay, providing me with the freedom to pursue my dreams and make my mistakes without harsh rebuke.

Abstract

A system for generating natural language descriptions of simple algorithms from their semantic representations is presented. This system emphasizes the organization of the planning and linguistic aspects of natural language generation. The view taken is that most of the planning process is centered around making stylistic compromises based on reasoning about the linguistic abilities (and inabilities) of the system. The planning process produces an adequate and diversified set of initial sentence schemas which are then subject to stylistic and semantic transformations which manipulate the text into its final form.

The system is organized as a community of *stratifications* which communicate with each other in a fairly loose manner. Each stratification is a hierarchy of individuals, called *units*, organized in meta-object/object relationships; these relationships allow observation and description of some units by others. It is shown that a great deal of nower obtains from organizations of this type and from the interactions between the objects that exist in such systems. Further, this organization is demonstrated to be a valuable method of viewing both intelligence and the programming structures needed to produce intelligence.

Linguistically, the concept of cohesion is exploited and a small catalogue of cohesive techniques is presented (e.g., pronoun reference, ellipsis). The structure of a linguistic communication is related to the organization of the units in the system. The notion of 'hybrid matching' of symbolic descriptions with associated measures is explored, in which strength of description, general stylistic influences, and specific contextual influences affect the outcome of a match; this descriptive framework is the basis of the linguistic reasoning behavior of the system.

Abstract Version 2

A system is presented that generates natural language descriptions of simple algorithms from their semantic representations. The system emphasizes the planning, linguistic, and creative aspects of generation rather than the theory of explanations *per se*. Most of the planning process involves making stylistic compromises after reasoning about the linguistic abilities (and inabilities) of the system. The planning process produces an adequate and diversified set of initial sentences that are later subject to stylistic and semantic transformations which manipulate the text into its final form. That is, the system models deliberate writing rather than spontaneous speech.

The system is organized as a community of *stratifications* which communicate with each other informally. Each stratification is a hierarchy of individuals, called *units* (ála KRL), organized in meta-object/object relationships; these relationships allow observation and description of some units by others. The success of this system shows that a great deal of power obtains from organizations of this type and from the interactions between the objects in such systems. Further, this organization is demonstrated to be a valuable method of viewing both intelligence and the programming structures needed to produce intelligence.

Linguistically, the concept of cohesion is exploited, and a small catalogue of cohesive techniques is presented (e.g., pronoun reference, ellipsis). The structure of a text (a set of related sentences) is a function of the organization and description of the units in the system. The notion of 'hybrid matching' of symbolic descriptions with associated measures forms the basis of the linguistic reasoning behavior of the system, and the strength of description, general stylistic influences, and specific contextual influences affect the outcome of a match and, hence, the style of writing.

Chapter 1

Introduction

And would it have been worth it, after all,
After the cups, the marmalade, the tea,
Among the porcelain, among some talk of you and
me,
Would it have been worth while,
To have bitten off the matter with a smile,
To have squeezed the universe into a ball
To roll it towards some overwhelming question,
To say: 'I am Lazarus come from the dead,
Come back to tell you all'—
If one, settling a pillow by her head,
Should say: 'That is not what I meant, at all.
That is not it at all.'

Eliot

Why "The Love Song Of J. Alfred Prufrock"?

I wrote this dissertation during a personal tragedy that cost me my wife and son: after turning it in, they left me. You can read about it in my book **Patterns of Software**.

Was it *worth it, after all*?

At 6:47 A.M., July 1, 1980, I finished dedicated programming on Yh, my thesis program. This program is 35,000 lines of code and initial data, and programming on it started, though informally, 4 years ago. Although about natural language generation on the surface, this thesis is more realistically about the approaches I took in 'solving' the problems, the organization of the system *as* a system for creating artificially intelligent individuals,[1] and about the way that I created the programming environment and the ways that I have come to think about programming in that environment.

I intend to program Yh more, adding more abilities to it in the near future, and many parts of it were written with the thoroughness necessary to be used in my research as it evolves. Many of my goals and dreams about what this program would be have been pushed aside in order to complete the degree for which it was intended. And, as a dim image of what that program was to be, it is a sad testimonial to the abilities of a single programmer. But, every line of code is written as I believe it should be, and what the

[1] By *individuals* I mean computer programs or machines that are able to respond to a rich environment or set of circumstances. This is contrasted to programs or machines that operate in a limited domain and which are not intended to be extended. Thus an individual should be subject to growth.

program does it does honestly and without shortcuts. Therefore, I want to make clear that the thesis of this research is contained in the methodology I used, in the philosophy of the endeavor, in the outlook on the problems of artificial intelligence (AI), in the representations and representation systems I used or invented, and in the way that I can gently persuade this ponderous system to consider the suggestions I give to it.

1.1 What is Yh?

Yh?

The name of my thesis system was "Yh." For my dissertation I created a special font for the name using Metafont. It looked like this:

I decided to not bother making such a font for this version because, really, who would care? Until 2024 I never told anyone except Ron Goldman and a few close friends how to pronounce the cryptic name: Yakety Hacks.

Yh is a program that generates natural language explanations of simple programs. These programs are assumed to be the output of a program synthesis system that engages in a dialogue with a user in order to obtain the specification of the program that the user has in mind. A result of this dialogue is an annotated representation of the program, data structures, auxiliary routines, and some noun phrases (in the form of parsed English) that relate parts of the program to concepts that are familiar to the user. Yh uses this representation and those references to user concepts to build an explanation of the program as understood by the synthesis system. The exact nature of the activities of Yh will be presented after some preliminary remarks on the overall approach and philosophical underpinnings of the thesis.

The main thrust of this thesis is that any large system to perform a wide variety of activities that would be called 'intelligent' by an observer of the activity will very likely be built up out of an extensive set of experts [Minsky 1977], each, perhaps, with its own range of arcane abilities. A natural language generation system is certainly a large system requiring a great deal of information about word uses, constructions, transformations, and syntax.

What this thesis addresses is the problem of how to organize such a large collection of diverse information in such a way that, although the exact structure of each 'unit' of information is non-uniform system wide, the access to that information is uniform throughout. Moreover, the problem of letting the system be able to be cognizant of its own abilities and to reason about them is considered, both problems being partially solved.

Since a system with a number of abilities and experts can be made to behave differently given which experts are active at what times, there is the further problem of 'programming' this system to demonstrate the exact type of behavior desired. This last problem has a solution that falls out of the solution of uniform access.

The starting point of this research was natural language generation within the context of a program synthesis system [Green 1977]. and there will be a considerable amount of discussion will be about that aspect of the system. In this thesis, natural language generation will sometimes be referred to as: *the domain*.

1.2 Complexity versus Simplicity

Before you begin to read this thesis I would like to make a prediction and a warning: I guarantee that when you are finished, it will have occurred to you several times that the ideas behind this thesis are organized in an excessively complex way. You will say, "why is this so complex? Surely it doesn't need to be."

Years ago, Herbert Simon [Simon 1969] gave us the parable of the ant on the beach, in which the complex behavior of the ant as it traverses the sand is viewed simply as the complexity of the environment reflecting in the actions of the ant. He says:

> *An ant, viewed as a behaving system, is quite simple. The apparent complexity of behavior over time is largely a reflection of the complexity of the environment in which it finds itself.*

He bravely goes on to substitute *man* for *ant* in the above quote and attempts to make sense of that statement. The task of creating an *intelligent* machine has evolved historically from the initial sense of simple programs demonstrating interesting behavior even though these programs are simple. This sense comes, I think, from the speed of the machine in executing the steps in our program; even though the machine does not really deliberate over the process, it proceeds with such haste that a large number of paths can be explored easily.

I remember years ago I first watched my first computer busily simulating a circuit, and I was fascinated as I changed parameters and observed the behavior of the system change in beautifully complex ways. My thought was: how easily the machine could be programmed to explore many different ways of dealing with my problems, and with great speed show me all the various alternatives and their outcomes.

So I, too, fell prey to the hypnotic effect of the computer as it hurriedly went about its business. And I, too, fell prey to the belief that simple principles could lead to complex and appropriate behavior.

Years later, disciplines of computer science arose which studied *algorithms*, soon considered the ultimate form of programming: one developes or dreams up an algorithm to solve some problem. Algorithms are short, clever things that magically produce *the answer* after a number of complex calculations. Even now, in 1980, people talk of the art or science of programming, programming methodology, and the psychology of programming, while all the time referring to dreaming up algorithms for problem solutions.

It is almost as if the notion of a *batch* (versus a *timeshared*) system were viewed as the paradigmatic framework for computing, and in this framework the programmer hesitantly submits his program to the monolithic device and anxiously awaits the detailed results at the nether end.

But algorithms are uninteresting objects from my point of view. They reflect the attitude, if not the fact, that one starts with a good idea of how one wants something to be done, or at the least, certainly, *what* one wants to be done. Writers in the field talk about 'predicate transformations' and input/output specifications. The main thrust of programming is, in this view, to get a program to take some objects and produce some others, usually with the program starting with one and ending with another. So we talk about a program to sort numbers, or to invert a matrix.

But in this thesis we treat a program as a thing, loosely speaking, that demonstrates some behavior we are interested in. Normally these programs never halt (or shouldn't) and to write them is to be in a symbiotic relationship with them—mutual learning. They exist in a world and interact with it continually, and to not do so is to be logically dead; and it is through this interaction only that we are capable of observing the *mindlike* qualities of the machine in question.

In 'The Lives of a Cell,' Lewis Thomas [Thomas 1974] says, when discussing the variety of life that goes on in each cell:

> *My cells are no longer the pure line entities I was raised with; they are ecosystems more complex than Jamaica Bay.*

He later goes on to compare the cell as an entity to the earth, not the comparison we would have thought even a short while ago.

Each cell, then, is incredibly complex, and our brains are composed of very large numbers of them, connected in complex ways; and as we attempt to learn of the interconnections between them, we only find that we are having more difficulty than anticipated.

If we posit (and almost by definition of our research we must) that the brain is equivalent[2] to the mind, then we are facing an object with overwhelming complexity and parallelism. It does not seem likely that a small (100 million word) program with a single processor is going to be able to demonstrate a substantial proportion of the behavior that a typical mind does.

1.3 Co-Evolution

Living entities co-evolve with each other and with their environment, and to think of creating an entity without a world that is matched to it is unrealistic. And if we fail to make truly *intelligent* machines, it will be because we have taken the world as we think it is and have tried to make a mind that fits it. In cases where the programs we have written exist in a world made with them (operating systems, text editors, interactive programming languages) the programs exist marvelously, yet when we work with the natural world, the results are pitiful.

The artificial part of artificial intelligence is the world that already exists, not the programs we try to write. A natural world for any behaving system is one that has co-evolved with it; if the world is provided in any other way, it is an unnatural world for the program (or individual) that will need to interact with that world. In order to match the computer to the world, we will have to produce—write or build—an intermediate layer or level of processing between the world and the computer, and this is exactly what vision, robotics, and speech/hearing research is about.

So I come to the point of saying that, even though I am constrained to leave the world as it is, the program I write must be complex to demonstrate the complexity of behavior it must; the complexity of the brain is the result of natural co-evolution with a complex world. Think of how sad we would be if the key part of our minds were reduced to an algorithm that Dijkstra could prove correct, that runs in $n \log n$ time, and that shows correct lexical nesting!

No program that is interesting from the point of view of behaving like a mind can be other than extraordinarily complex.

1.4 The Goals of Artificial Intelligence

In my view, AI has been suffering from approaching the wrong problems from the wrong direction for a number of years, perhaps since its inception. Progress is often good in some limited areas, but if the problem is to build real *mindful* objects (as opposed to *intelligent* objects), I think a different approach is needed.

Some people believe that machines should be able to plan in much greater detail than humans, and so machines are not "intelligent" unless they do plan better. This notion seems to come from the "puzzle" tradition of artificial intelligence. In this tradition the paradigmatic behavior of an intelligent system is that it is a great puzzle solver: many things that robots will have to face are like puzzles, so a good thing for them to be able to do is solve puzzles. This has led, naturally, to the question: how can knowledge be best represented to facilitate puzzle solving? The "solution" is to represent facts as objects within the

[2] In deference to the mind-body argument I have chosen the word 'equivalent' in this context to mean that there is nothing other than the brain as the object which is responsible for all of the phenomena we care to think about as belonging to the mind. Some of these phenomena may be epi-phenomena of the activities of the brain, but all behavior is traceable to the brain under this wording.

program. "Moving around" these objects within the representation then corresponds in a strong way to "moving around" objects in the real world.

A typical example from the problem-solving literature is the missionaries and cannibals problem. This puzzle, discussed in [Ernst 1969] and [Jackson 1974] has recently re-surfaced as a key example [McCarthy 1980]. Representing the objects in this puzzle is usually either done with state descriptors or with first order logic. The crucial fact of the matter is that this puzzle has a definite solution. Moreover, the solution can be recognized. Even further, it is usually possible to determine whether progress is being made towards the solution.

Thus, when life is viewed as a series of puzzles, it is natural to program representation systems to try to get things into shape for the puzzle solving race which all intelligence is believed to be.

Puzzle situations are much like traditional board games or brain-teasers. It is not surprising to witness that early researchers worked on board games (chess, checkers) as a paradigm of intelligence. Others worked on brain-teasers. These situations are amenable to formal methods since a concise statement of the relevant facts is easily statable in such languages.

The problem is that these types of problems are "good" to work on because they can be simply stated with only the essential facts represented and an answer can be recognized, both by machine and by human observers.

In real situations, on the other hand, it is often difficult or impossible to formulate a problem in terms of a single goal or a small set of simple goals. Moreover, even in those cases where such a formulation is possible, there may not be measures of progress that ensure a solution to the problem as stated. Systems based on the measurement of progress towards a single goal, then, are sometimes at a loss in real situations.

1.5 Ad Hoc to Epistemology to Ontology

This is the most important section in the presentation since if anyone has a quarrel with what I am doing, that quarrel will almost always resolve itself into one about the basic point of the research. Many papers and theses on AI talk about the technical details at length and leave the reader to speculate about where on the philosophical spectrum the researcher lies.

The history of AI is often a sequence of bold attempts with moderate successes followed by a severe retrenching. In the early days there was a flurry of activity based on largely *ad hoc* techniques with the faith that pure raw power would be able to achieve interesting results. This was predicated, as noted earlier, on the attitude that complex behavior resulted from a simple system in a complex environment. Some examples are Evans' analogy program, Guzman's scene analyzer, and the entire mechanical translation project. Though these programs were complex and performed well in their domains, they generally embodied a simple theory of the situation attacked. In the analogy program there were the hope that simple descriptions and simple differences between such descriptions would lead to analogical reasoning; in Guzman's program, a simple set of descriptors—labels on lines in line drawings—and a syntactic operation on those labels and lines would be able to interpret many scenes; and in the mechanical translation work, simple transliteration techniques and syntactic analysis were hoped to result in an expert translation system.

There were some successes at this time, but there was a barrier that soon loomed ahead of the researchers. The barrier was that the simplicity assumption was challenged at the point of *knowledge* or *semantics*. The apparent problem was that the programs did not have good knowledge about the domain that was being explored. The early programs relied on formal actions in a syntactic world. *Ad hoc* syntactic-based theories became unrealistic for larger problems.

This realization came during the late sixties or early seventies and signalled the beginning of the knowledge-based or what I call the *epistemological*[3] era. What really was happening was that the term 'artificial intelligence' was taken seriously, and the line of research became: produce an intelligent program. 'Intelligence' was possibly intended by some to refer to the difference between men and animals, and the research of these people was therefore not particularly oriented towards higher intellect. But for most, Simon and McCarthy for example, producing an 'intelligent artifact' meant producing machines that would perform well in intellectual skills—formal reasoning, puzzle solving, games (like chess), and the like. The emphasis of this line of research was on the *intelligence* part of artificial intelligence.

The research of this era focussed on inventing representations of knowledge and writing programs that would operate on these representations to solve problems in, generally, toy worlds.[4]

Of course, what most people refer to as 'intelligence' is involved closely with the idea of education or experience, though this is strictly untrue by definition; but some people are lured to confuse *intelligent* with *intellectual*, though not in a malicious way. Many of the tasks chosen as targets of research require a lot of experience, at least, with the world, and this experience is simply 'knowledge.'

Knowledge takes a long time to acquire for people, and parents are aware that it takes many weeks before any real type of awareness and knowledge of the world appears in their children. Infants spend many hours attempting to do what adults consider trivial, and what researchers spend years trying to get primitive manipulators and cameras to do. That the knowledge barrier was approached is not surprising given the difference in processing power between humans and computers, and given the difference in time that is put into each 'programming' task. A human brain, viewed as simply a processing machine, is several orders of magnitude more powerful than a PDP-10 [Moravec 1981], and 'evolution' coupled with months of day and night work by infants represents very much more 'programming' effort than the paltry man-years put in by programmers in research centers.

A great deal of progress towards producing programs that perform at even expert levels in some intellectual skills (for instance, medical diagnosis) has been made, yet there is still something wrong with the situation, and this problem is roughly centered around tolerance, creativity, judgment, adaptability, brittleness, common sense, etc. Programs that behave in the rarified atmosphere of intellect ignore the underpinnings of creativity, exploratory behavior, and the ability to think about one's own abilities which forms the foundation of intelligence.

We are still in the middle line of research which is *epistemology-based* or knowledge-based, and much time is spent struggling to gather knowledge about parts of the world that are interesting and relevant to a task to be performed. But, there is a subtle type of restriction that can be observed in such systems.

Consider an advanced knowledge-based system that reasons in some (even technical) domain. The research that Feigenbaum et al [Feigenbaum 1980], [Feigenbaum 1971], [Buchanan 1969], does is a good example, and many of the programs of that research group are highly impressive from a performance point of view. In many cases a great deal of time is spent gathering reasoning protocols and detailed knowledge of the domain. Typical results and reasoning chains are pieced together, and a good representation of those chains and that knowledge is devised. Then an inference engine is produced that, given some problem, produces an expert result of some sort.

The danger here is that the structure of the knowledge might very well be like the more sophisticated backpacking tents available today. These tents have an elaborate structure, but to compensate for the general

[3]By *epistemological* era I want to emphasize that I mean that period when the knowledge that our programs had about the world was viewed as being the critical problem. We started talking about knowledge-based system, and the word 'semantics' became crucial to the discussions of the day.

[4]A *toy* world is not necessarily a world with toys in it, such as the simple blocks world, but a world in which the scope of the domain is limited in strict ways. So, medical diagnosis *as treated by most AI research* is a toy world, although it is an important one.

lack of problem-solving ability of some backpackers, the mechanisms are cleverly put together so that even if the tent is thrown to the ground, it will nearly assemble itself. Thus, in the case of expert systems, the success is really a product of the clever nature of the representation of the knowledge in the system, which will produce a good solution from a kick by the user.

But is the system too sensitive to the representation? The answer lies in the lack of extensibility of some of the systems, where the extensibility lies not in the difficulty of adding new knowledge of the same type as already is used in such a system, but in the difficulty of telling the system things in a different representation.

We can view the entire process as one of island hopping in the familiar analogy to planning: that is, many systems attempt to plot a course from one island to another by a sequence of intermediate islands which are accessible from each other. In the epistemological system, the choice of islands is radically limited by the type of knowledge that the hops from island to island represent.

One other way to think about the island analogy is by considering the types of interfaces that occur between islands. In any sort of modular system there are objects and there is an interface between these objects. This interface represents the communication that goes on between the two modules or islands. In many systems that use an island-hopping style of planning, the conditions that exist after an island is processed, along with the requirements of the next island, exactly match the abilities of the system. The planning problem, then, is difficult because the exactness of interface must be made the case at each step rather than left open.[5]

We find that the epistemological system has as a motive engine a program that can match these nearly exact objects. This is a problem because, though there is a wide variety of knowledge represented in these systems (so-called *knowledge-based systems*), there is little flexibility in the system to apply poorly matched knowledge to a situation.

In an *ontological* system[6] we find that the interfaces have only a resemblence to one another, and the job of matching them is much harder and may involve some risks. The structural coupling of the system is reduced and powerful (and unguarantee-able) procedures must be employed.

An interesting analogy can be made that shows the traditional task of AI to seem somewhat bizarre. It is often the case that programs are written with the intention of being "expert systems," which means systems able to perform at expert human levels. Now, no one expects that these systems are expert in any other domain than that for which they was designed. Often very intricate deductions are made with carefully chosen methods; the power of these systems comes from the highly distilled nature of the problems they actually work on, and the structural coupling within the domain is sufficient to see them through many stormy occasions. But these systems lack any knowledge outside their areas of expertise, and, in fact, lack even simple common sense. They are often unable to communicate very well even about their areas of expertise. In short, they are best compared to idiot savants, who can do some things surprisingly well, but little else. Think of trying to bring up a child to only be able to diagnose orthopedic gait problems from the trunk down, and NOTHING ELSE.

One would think that there is some overall "problem" that needs to be solved in order to admit machines to the ranks of the cognizant. This problem is the *ontological problem* as over against the *epistemological*

[5]Similarly, in the case of modular programming, the the information that is passed is forced through a narrow bottleneck—normal parameter passing protocols—which are derived from mathematical notations and register and/or stack requirements in the physical computer. This bottleneck has the effect of enforcing a strict matchup between modules and thence resulting in difficulty programming the system. This is not a point that most people agree upon; it is my feeling that languages that let you decide on interfaces as late as possible (such as LISP) are better for very large systems programming than those that don't (such as PASCAL). I don't think this point will become obvious until extremely large systems are being routinely written.

[6]Very few of these systems exist: I claim Yh is one, and FOL [Weyhrauch 1978], [Weyhrauch 1974], [Filman 1976] is another. By *ontology* I mean the quality of existence or being; in particular, I mean being able to interact with the world, to apply imprecise methods to everyday problems, and otherwise to exist in a flexible manner.

problem, which has been extensively attacked in the past. Challenging the ontological problem means facing squarely the problems of creativity, uncertainty, hedging, influencing, and, not the least, being.

My own feeling is that there is an underlying sort of behavior and organization for an 'intelligent' system such that the intellectual skills we hope to give to such a system are naturally imposed or constructed on top of this underpinning. That is, developmentally, the ability to interact with a flexible world in a flexible way preceeds the ability to interact with an inflexible world, which is one way to characterize intellectual skills.

So we again begin to worry about the underlying processes that the system can experience. Early researchers worried about this too, but not as a response to the knowledge problem.

The philosophical next step taken in this overall research program is to wonder about the wisdom of accentuating the 'intelligence' in artificial intelligence. So, instead of thinking about how to program a computer to do intelligent things, we think about how to program the computer to behave in such a way that we are led to believe it has a mind.

This distinction is reflected in the terminology *mindful behavior*. Whereas before we talked about the knowledge of the system, we talk now about the mechanisms of *being*; and we find right away that a lack of terms for talking about this aspect of existence makes us sound like wide-eyed madmen. Suffice it to say that the transition from epistemology to ontology is being attempted.

The main thrust, then, of this research is to explore ways of building systems (very large systems) that are capable of exhibiting mindlike behavior and accept different types of experience in order to change

1.6 What are the Goals of AI?

A mindfully intelligent object, whether human, animal, or artifact, is one that can endure many situations and even thrive within some of them. Don't forget, even horses are more intelligent on the whole than any program yet devised. A horse can cleverly survive in a real world, yet even the most sophisticated program of any kind only survives for years at most in a very restricted situation before intervention is necessary. One class of such programs is operating systems. But even they interact in a small domain or culture of programmers and they can respond well to only a small number of relatively predictable events. Sooner or later some little detail goes wrong and the program's limited ability to make adjustments forces some expert programmer to intercede. Horses do much better.

Take the example of an intelligent natural language generation system. Such a system would, ideally, be able to take anything that could be represented within the system and produce an utterance that is related to it in a strong way. The utterance would reflect a good attempt by the system to express the represented situation. There are several ways to achieve this goal, but all of them depend on overthrowing any thoughts of producing the "correct" utterance for the represented situation.

Any system[7] has some set of abilities, usually represented in a non-uniform framework. In order to accomplish its designated tasks the system needs to discover the appropriate capabilities and use them. In a typical program (as opposed to a system) its abilities can be easily discovered with no effort at all: they are implicit in the function calls that are apparent in the program. For example, if a program needs to compute the factorial of some integer, the code, (FACT n), represents the knowledge of who the expert is in this case. On the other hand, though locating the information about abilities is trivial, when new abilities are added to such a program, many places where this information is explicitily represented need to be updated.

[7]When I refer to a *system* I generally mean a large computer program that is able to do more than one specific task. Thus, a program that sorts an array of numbers is not a system, but a text editor is, even though its specification looks so simple. A text editor has a number of operations it can do, and it interacts with a user in order to perform a loosely described task. A sorting program can sort the elements of some data structure when the data structure is properly presented to the program.

Part of the reason that programs must do poorly, at least now, is that the range of things that they can be taught is quite limited. By this I don't mean the quality so much as the quantity. We have expected to be able to take a small domain, like automatic programming, about which, it would seem, almost everything is known, and program up something which will act like a professional in that area, yet be totally ignorant of everything else.

To produce anything more than this we would have to solve and integrate solutions to a great number of problems, including many algorithmic problems in vision, hearing, and other "low level" tasks where a clever technique with a narrow range of applicability is acceptable. Accomplishing this is necessary in order to couple our machines with the world better than has been done before; and this will mean building input and output devices that perform these activities, and possibly in an algorithmic way.

So for the moment, since we cannot hope to reproduce anything like human breadth of expertise, we must content ourselves with how to produce in a small measure that which is most mindlike in a restricted domain.

1.7 Intelligence and Communication

Consider a team of specialists. If a problem they are working on is not entirely within any of their specialities, then one would expect that they could solve it after a dialogue. This dialogue would interchange information about strategies, techniques, and knowledge as well as information about what each knows, why it might be important, why some non-obvious course of action might be appropriate. In addition, one would expect bickering, propaganda, lies and cheating to also occur as each strives to further his own viewpoints.[8]

The first of these categories of information (the co-operative things) appear to be the essence of an intelligent "system" of individuals. The "intelligence" of the system looks as though it emerges not so much from the individual, though this is certainly important, but from the interactions it has. One would think that a lot of efficiency is lost in one of these interactions because of the "natural language" that is used: If a more streamlined language could be used then the intelligence of the system would increase.

First let's think about a standard programming language.[9] In a formal programming language one often sets up a conversation between a team of experts, namely the "subroutines," "functions," or whatever they are called. Each member of the team "knows" about the experts of interest to it: "knows" in that whenever such an interaction is called for with an expert on some topic, the appropriate expert is contacted. The method of conversation is quite formal and efficient. Let us think of LISP. Here one places the "important items" in specified locations. These locations represent the semantics of what the program is trying to say. One might think of it as saying: *Here is the number that I want you to factor for me.* Saying this is accomplished by putting a number in the "number to factor" position that the factoring expert has set up. The factoring expert is contacted, finds the message, performs whatever it wants and puts the "answer" in the *here's the answer you ordered, sir* slot that everyone has agreed on.

In some sense, the factoring expert doesn't really know who it is talking to since that information is hidden from view. The conversation is between an expert and an anonymous caller!

[8] One possible scenario for emerging intelligence is to provide a computer with a very large range of abilities and with descriptions of these behaviors so that the program can reason about itself. One strategy for allowing the system to reach a steady state in which its behavior is well attuned to the world is to let it 'evolve' in the Darwinian sense, meaning that individuals in this system must either be selected to survive because of useful traits or to die due to uselessness. While exploring this idea, it may become advantageous for some of the descriptions of individuals to be less than honest about what can be accomplished. However, this is only a speculation at the moment based on the hint that co-operation may not be as effective as conflict.

[9] Of course, there are a number of programming languages which are non-standard, and which do not suffer from the same problems that I am complaining about here. Three examples are MICRO-PLANNER [Hewitt 1972], CONNIVER[Sussman 1972], and PROLOG[Kowalski 1974] However, these programs are not used outside of AI applications, and "standard" programming languages and the style of thinking that goes along with them are being taught nearly universally.

These types of conversations are what the entire computation of the large program is like since each individual subroutine is nothing more than a sequence of these conversations with some ordering performed internally (control structure).

Now, if some new expert comes on the scene who claims to factor better and faster than the old expert, how would the user of the factoring expert get the information that there is a new subroutine in town? Well, usually the answer is that the user knows the name of the expert or else its location. We could try to just pull a fast one on the identity of the experts, leaving their names and/or locations the same or else we could stop the world and rebuild it.

In living systems and societies we do not usually change names or identities that often. And when new information comes along it is spread because the language we use is rich enough to accomodate such things. Imagine a programming language that could have a richer conversation along these lines. It is not that the human in a society has a much richer set of methods for finding out what he needs to know, but that the methods of communicating what each person knows are richer. A human who wants to find out how to do something simply looks it up or looks in some standard places for who might know. The difference is that the human can recognize in more different ways that someone else is claiming to know something of interest. Conversely, a newcomer can easily advertise his or her abilities and knowledge with the confidence that it will be generally understood.

Natural languages are, thus, very much unlike artificial ones, which have a strict semantics and usually a rigid syntax. A programming language or system that allowed such conversations to go on could be much easier to program and would be generally more robust, though possibly slower, although one can imagine compilers that would freeze the state of a world if that is what is needed. Alternatively, the structure of programming languages has often been designed to fit the architecture of the underlying computer, and so it might be that there is an architecture that doesn't naturally channel programming languages into traditional molds.

The programming language would then allow routines to discuss abilities, make deals, trade information, and do other interesting things. In fact, one might want to explore what the advantages and disadvantages would be in the creativity (say) of a system that allowed the routines to lie, cheat, steal, etc.

The main feature of a system written in this kind of programming language is that it would be robust and able to respond better to more situations than most programs do now. This ability should not be underestimated.

1.8 What the Domain is About in Outline

This domain of this thesis is natural language generation, a topic that has not gotten much attention in artificial intelligence circles. In a way this neglect is understandable since natural language can be generated fairly easily using ad hoc techniques. One of these techniques is the so-called 'fill-in-the-blank' technique, in which there is a pattern for a sentence or a sequence of sentences in which there are slots that are filled in as needed. For instance, one could have the sentence: "The program inputs a <blank>." Then the program would fill in <blank> with 'number,' 'list' or whatever to complete a reasonable sounding, competent sentence.

Another possible reason for the lack of interest is the fact that generation (natural language generation) does not seem to be an area in which cleverness, intelligence, or deductive ability is *obviously* necessary. To be sure, these abilities figure strongly into the activities of good writers, but they do not seem to be the hallmark of writing as they do in problem solving, robot planning, natural language understanding, and game-playing.

Generation is centered around the vague notions of creativity, choice, and judgment. The last of these, judgment, forms that basis for the most important parts of my system, Yh as it is written; only recently [Berliner 1980] has judgment become a real research topic for AI, though critics have long ago pointed this out [Dreyfus 1979].

The goal within the doamin is "deliberate writing" as part of an explanation system residing in a program synthesis system. The plan is to have the program that is acquired from a dialogue with a programmer be explained, in English, to the user so that he will have faith that the system had produced a satisfactory program. By 'deliberate' writing, I mean writing that is revised and planned using several passes of consideration over the text as it is produced, much the way a writer revises drafts of a piece of writing.

Deliberate writing can be viewed in a different light than spontaneous writing or speech, in which the constraint of left-to-right generation is emphasized. I wouldn't want to speculate on whether this constraint is realistic or not, given that people obviously deliberate while speaking on occasion and appear to be speaking spontaneously other times. Therefore, I have chosen deliberate writing rather than spontaneous speech because one can not expect to do the latter unless the former is possible too.

1.9 Fluid versus Essential Domains

Didn't Douglas Hofstadter invent *Fluid Domains*?

Maybe. Or perhaps sort of.

Hofstadter was visiting Stanford after I finished my dissertation and started working for John McCarthy. I was writing a paper called **Fluid Domains** in 1981, based on my dissertation. Doug gave me a paper or note he was working on with an epigraph from my paper. Later he came to my office to talk about it, and he remarked something like "I was looking through your directories and I found your paper on fluid domains, and I thought it was pretty interesting, so I took this quote from it—I hope you don't mind." SAIL (the Stanford AI Lab timesharing system) didn't have directory protection; anyone logged in could read anyones' files, and even edit them and create new files there. Reading other peoples' draft papers and source code was part of how knowledge was shared. This also was the philosophy at the MIT AI Lab, and this is part of why Richard Stallman came up with the idea of Free Software.

Doug was a great idea sponge and concept integrator, so I didn't mind (and still don't) that he (maybe) borrowed some of my ideas and made much more from them than I ever did.

I want to make a distinction between two of the kinds of domains that one can work with in AI research: *fluid domains* and *essential domains*. These qualifiers are meant to refer to the richness of these domains, and the implied behavior we expect to find in these domains.

In an *essential* domain, there are very few objects and/or classes of objects and operations. A problem is given in terms of this domain and must be solved by manipulating objects using the operations available. Generally speaking, there exist no more than the number of operators minimally needed to solve the problem, and usually a clever solution is required to get at the right result.

A typical essential domain is the missionaries and cannibals problem. In this problem there are three missionaries, three cannibals, a boat, a river, and the proviso that the six people need to get across the river, and if the cannibals ever outnumber the missionaries, the result is dinner for the cannibals.

Now the problem with this is that it is assumed this is taken as some formal situation, and things that people would really do, such as looking for a bridge or thinking about some people swimming, is not allowed. This is a hard problem for most people to solve, and yet these same people can speak English; AI yearns to solve the former in order to get at the latter, it seems.

Other essential domains resemble puzzles and simple games; often mathematics looks like this.

An important feature of these situations is that it takes great cleverness or intelligence in the classic sense to solve them; they are called *essential* because everything that is not essential is pruned away and we are left with a distilled situation.

In a *fluid* domain, there are a large number of objects and/or classes of objects and a large number of applicable operations. Typically these situations are the result of a long and complex chain of events. Generally there are a lot of plausible looking alternatives available, and many courses of action can result in a satisfactory result. Problems posed in this type of domain are open-ended and don't have a very recognizable goal.

A typical fluid domain is natural language generation. In such a domain there are a large number of ways of doing various things, beginning with inventing phraseology out of whole cloth, and progressing towards very idiomatic statements. There are many ways to start the process, many ways to proceed once started, and many different ways to decide that one has completed the task. A feature of such a domain is that judgment is much more important than cleverness, and recognizing a situation as familiar is the crux, and there is little need for a fancy planning mechanism.

Natural language generation can be put into the essential domain mold by making some restrictions on how things are done. For instance, one can limit the output to single sentences. Sometimes people think with this restriction, that there is less of a problem, or that the essentials of the problem come to the fore. I believe that this assumption makes the problem much harder than it needs to be, though the difficulties which arise in that 'simplified' case are worth a lot of consideration.

In a full-blown fluid domain, the situation is that we have a large number of building blocks that can be pasted together in various ways. The idea is to choose those blocks which can fit together well. Since there are many blocks, one can almost start off in any direction and find blocks that will curve things the way we want. The real problem is with using judgment to paste the blocks together and to keep the spiralling course of things headed in a good direction.

In an essential domain, though, we still have the problem of pasting blocks together, but we have so few of them that we also have to carefully select these blocks, and we have to locate a strategy for piecing them together before we can even start the pasting.

So now there are two problems where there was only one before. We haven't gained in simplicity, but have taken the problem of intensely clever planning and turned it into a very large system management problem, for where we once had a dozen ways to do things, we now have hundreds, and in order to use the blocks in a non-degenerate way, we must be able to explore in this space of choices and modify the system so that new paths come to light.

In some ways it is amusing to think about the ways that people have made problems harder than they needed to in the name of simplification. It is often said that one can easily afford to limit oneself to single sentence output, because with judicious use of punctuation, one can turn any multi-sentence text into a single sentence text. The claim is that there in no loss of generality. But to accompany this lack of loss, there are further simplifications which make the problem a puzzle once more. It is like saying that the goal of the research is to discover the ways of organizing a system that will be an expert at using tools to put together a car, but then limits the tools to a hammer and a screwdriver for simplicity.

1.10 Broad History of the Domain

The problems of generation have fallen into two categories traditionally: 1) producing sentences that belong to the language of interest ([Yngve 1962], [Klein 1965a], [Klein 1965b], [Friedman 1969a], [Friedman 1969b], [Winograd 1972], [Meehan 1976], [Goldman 1974]) and 2) producing a plan to say something reasonable about some represented meaning (but without actually saying anything). A third

concern has been to wonder about the relationship between the text produced and the represented thing, which is taken to be the meaning.

Historically the first of these turned out to be fairly easy to do because there was no concern with making the text produced have a coherence or even a meaning at all. Various methods—transformational grammars, context free languages, and augmented transition nets, for instance—have been used as ways of representing valid English sentences.

Since the surface structure can be produced with little difficulty (in a pinch), many people concentrated on the problems of meaning, which is the problem of representation in traditional AI circles. As Goldman [Goldman 1974] points out, the easier it is to generate text from a representation the harder it is to reason about or otherwise manipulate that representation. His thesis explored this idea in some depth and will be discussed further in the next section.

Generally speaking, a *representation* is some objects that reside in a computer, a set of operations on those objects, and a set of mappings between those objects and those operations onto the 'real world' such that when those operations are performed on those objects, the mappings produce true statements about the real world. In short, a representation is just a model of the world in mathematical terms.

An interesting point about representations is that they rely on some individual who examines the representations and the real world and makes the judgment that the correspondence is faithful. Thus, a representation requires three things: an entity in which the representation resides, a world, and an observer.

Much has been said recently [Weyhrauch 1978] about the role of the observer in interesting situations. To a large extent this thesis is about this role as well as the role of the *participant* in cognitively interesting behaviors.

1.11 Detailed History of the Domain

When I started this work 5 years ago there was no one that I knew of who was working on producing text about some represented situation with a good surface structure or style. Now there are only a few people who are beginning their own research into the area [Mann 1980] [McKeown 1980].

Moreover, the research I am doing centers around the notions of judgment and creativity, while much of the other natural language research is focussed on the information transfer from machine to human reader. Thus, the two lines of research differ by one being of the *means* and the other the *ends* of communication.

The interesting part of research in English language understanding and generation comes from the fact the English is a unique language among those spoken and written in being grammatically simple and rich in borrowed words. It is largely formless and immense—ill-mannered. The following is a quote from "The Reader Over Your Shoulder" by Robert Graves and Alan Hodge:

> *The general European view is that English is an illogical, chaotic language, unsuited for clear thinking; and it is easy to understand this view, for no other European language admits of such shoddy treatment. Yet, on the other hand, none other admits of such poetic exquisiteness, and often the apparent chaos is only the untidiness of a workshop in which a great deal of repair and other work is in progress: the benches are crowded, the corners piled with lumber, but the old workman can lay his hand on whatever spare parts or accessories he needs or at least on the right tools and materials for improvising them.*

In the next paragraph they say:

> *English has always tended to be a language of 'conceits': that is, except for purely syntactic parts of speech, which are in general colourless, the vocabulary is not fully dissociated from the imagery out of which it developed—words are pictures rather than hieroglyphs.*

Research into mechanizing English understanding or generation must recognize this fact or else be locked into studying a pseudo-English rather than the real thing.

1.11.1 Generating Sentences in the Language

Early work in English generation dealt with generating strings in the language; people such as Yngve [Yngve 1962] and Friedman [Friedman 1969a] [Friedman 1969b] wrote programs that would produce sentences in English. The former used a context-free grammar and a random number generator; the latter used a transformational grammar, though she was more interested in making better transformational grammars than in conveying meaning from a representation.

1.11.2 Generating Sentences that Mean Something

Winograd's [Winograd 1972] program, SHRDLU, produced simple answers to questions using basically a fill-in-the-blanks approach along with some heuristics for pronoun introduction and duplicate noun phrase elimination. Sentence structures were rigidly chosen, and a noun phrase generator that would distinguish objects adequately by combing the data base for features and distinctions allowed natural sounding English to be produced about a very simple world. Nevertheless, in spite of the fact that this effort was clearly secondary to the effort put into the understanding of English, the results were quite good.

Simmons [Simmons 1972a] [Simmons 1972b] [Simmons 1973] generates paraphrases of English sentences from semantic nets using an Augmented Finite State Transition Networks [Woods 1970] (ATN) generator, which generates from left to right. Transformations on the semantic net coupled with the nondeterminism of the ATN result in paraphrasings. The semantic net is organized in the tradition of Fillmore's case grammars [Fillmore 1968].

Klein [Klein 1965a] [Klein 1965b] paraphrases paragraphs using a context-free grammar and 'dependency' relationships. The latter makes explicit the dependencies of words on other words within a text. Text is parsed into constituent and dependency structures. The original text is discarded, the dependencies are augmented by addition of dependency links between equal nouns, and a new text is semi-randomly produced that preserves dependencies, using transitivity of dependencies and 'paragraph' outlines (noun/verb pairs for each sentence, derived from the input text or from human input).

1.11.3 More Recent Attempts

More recently, the main contributors to the field of English generation have been Neil Goldman [Goldman 1974], James Meehan [Meehan 1976], Dave McDonald [McDonald 1979], Doug Appelt [Appelt 1980], Bill Mann and James Moore,[Mann 1980], Kathleen McKeown[McKeown 1980], and Phil Cohen [Cohen 1978].

Of these, Appelt, McKeown, Cohen, and Meehan are concerned with planning what to say rather than concentrating on the text that is produced. Meehan's program, TALE-SPIN, actually generates English, but in a very straightforward way, producing short, simple sentences. Since his aim is at story-telling, there is no blame associated with an ad hoc generator to test the important ideas.

Appelt uses problem-solving techniques derived from Sacerdoti [Sacerdoti 1977] to plan a sequence of speech acts in the domain of interest.

McDonald is concerned with devising a satisfactory psycholinguistic theory of left-to-right generation (speech).

Mann and Moore appear to be more interested in the surface structure, though their work is mainly still in the planning stage at this time. The method they use is to break up the situation (or whatever is to be expressed) into *protosentences* which are placed in an unordered set. Each protosentence could be stated in one English sentence, but quite badly. The interesting part of their proposal is the module which grabs pieces from this set and produces English text. The items in this set can be ordered and examined; *advice*

can be attached to any of the items. A *knowledge filter* is called upon to isolate those protosentences that should not be expressed because the reader already knows their contents.

There is a set of *aggregation* and *preference* rules for combining protosentences into larger English sentences; a numeric score is produced for each proposal for aggregation, and (modified) hill climbing occurs until no more improvement is given. The actual sentence generator uses a simple context-free grammar and a process for introducing pronouns and distinguishing noun phrases developed by Levin and Goldman [Levin 1978].

The main difference between my approach and that of Mann and Moore is that they do not stress the interaction between actual sentence generation and the construction of the plan for the text. In their view, as in many others,' the plan for expression and all stylistic considerations take place before words are chosen, which I feel strongly is a serious error. In many cases sentences and other utterances are planned on the basis of a set of words that 'come to mind' upon thinking of some situation. In terms of producing good English effectively by machine, Mann and Moore's could well be a more practical approach, but in order to appreciate the interactions and subtleties of generation, mixing planning with action is a better assumption.

1.11.4 Goldman

Neil Goldman is the big name in English generation, and his thesis [Goldman 1974] ranks as the best work in the field to date. He starts with a *Conceptual Dependency* representation in the style of Schank and produces a single sentence that expresses that structure. The main point he makes is that a representation that is good to reason with may be difficult to generate English from.

His basic technique is to construct a semantic net-like structure, called a *syntax net*, for the conceptual dependency structure (CD). This syntax net is built by looking at the action specified in the CD, choosing a verb, retrieving a case frame for that, and then filling in the parts. From the Syntax net, an ATN, much like that used by Simmons, is used to generate the surface structure. Choosing the words is by a discrimination net which looks for entries that satisfy all of the *defining characteristics* of the word to be chosen. Paraphrases result from choosing from the set of acceptable words.

The difference between this work and mine is that he deals with single sentence utterances and never considers the surface style of the resulting English. The interaction between word choice and planning substructures in the sentences is restricted mainly to noun-verb choices. That is, in cases where one is able to choose a verb to represent an action or a noun to represent the event of that action (collide vs collision), the structure of the sentence can be constrained by this choice, which can be simply a word choice.

For example, from Goldman's thesis, if the situation is that a collision between two cars took place because the street was wet, if the causality expression is *because* then one might expect the sentence:

The Ford collided with the Chevy because the street was wet.

while if *because of* is chosen, the sentence could be:

The Ford collided with the Chevy because of the wet street.

Moreover, if the nominal form, *collision* is chosen, then the causality expression might be restricted to *resulted in* and *resulted from*. The point being that word choice can influence syntactic structure; this point is discussed but not addressed in Goldman's thesis, while it is a seminal consideration in this work.

1.12 What the Domain is about in Detail

The program that I wrote takes an internal representation of a program as it is produced from a program synthesis system [Green 1977] and produces a simple explanation of that program and its data structures. The explanation is not meant to reveal some dark secret about the algorithm since it is assumed that the

algorithm was given to the system by the person who will read the explanation. Rather, the explanation will talk about the final form of the program derived from the algorithm, which will be of interest because the initial specification—verbal dialogue—is not assumed to present the program in any particular order; that is, the focus of the dialogue may not follow the locus of control very well, and data structures can be introduced anywhere by the user or automatically by the synthesis system.

Therefore, one assumption is that the reader has a good familiarity with the subject, and that the program does not need its teleology explicated. The net result of this situation is that it is the surface structure of the text that is the main interest; and as a program that might reside in a practical program synthesis system, its main function is to provide confidence that the synthesis system understands the task at hand well.

The program that is explained is represented in roughly a property-list based semantic net. Yh assumes a KRL-style [Bobrow 1977] representation for the program, and it also assumes that there may be entries in the representation for specific wording that was used in the dialogue with the user about the program during its specification to the program synthesis system. The essentially parsed surface structure for the text used by the user can and should be integrated into the final text in order to provide an anchor for the reader's understanding.[10]

The specific point of interest in this system, at the level of detail presented so far, is that there is a code-following explanation of the algorithm along with a purely data-driven explanation of the data structures. In terms of behavior, the data structure explanation is more interesting because the plan for presenting that material is not hard-and-fast as it is in the case of the algorithm, where there is less choice in the matter. Moreover, the ways of presenting algorithmic content are highly stylized, as in the case of a FOR-LOOP, in which you can basically only say something like: "for each element, i, in the <something>, <the program> <does something>," and limited variants.

In fact, program explanation is a fairly dull domain for natural language generation, as would be mathematics, since these areas have a stylized tradition of presentation. As a topic for a practical system, on the other hand, such a restricted domain is just right since one does not have to worry about subtlety.

As far as possible, then, research time was spent thinking about surface structure and other stylistic issues.

1.13 A View on Program Behavior

Although I have presented the outline of the problem in a standard input-output manner, with the program Yh sitting in the middle massaging the data as it flows through, this is not how I think about the way Yh works, nor is it the way that I believe an AI program should be viewed. Yh is a program that is meant to exist beyond a single invocation of it, and it is meant to have a 'state of mind' in the form of its current set of beliefs (or parameter settings and resident data structures if you wish). This 'state of mind' is modified as the program works, so that running it through its paces after it has run awhile will not usually produce the same behavior it produced initially. In this sense I view the process of generation as putting a disturbance in Yh which causes it to behave in certain ways, a side effect of which is the production of text. The disturbance can be thought of as producing a number of requests for activities that need to be undertaken; as best as the system knows it will attempt to perform these activities until the desire or need for them subsides. These desires and needs are represented in an internal description language; this language and its uses will be described in Chapters 5 and 6.

At that point the text is spilled and Yh awaits the next perturbation. However, the internal changes that are made may be permanent and reflect its growing set of beliefs about how to best respond to outside influences.

[10] Exophoric reference is the intention of these entries.

1.14 Personal History of the Problem

I believe it is often a good idea to relate as part of a piece of research the history of the investigation. Usually results are presented, and the final views in those results are not the ones that guided the process. The situation is similar to that of mathematics in which the final neat proof rarely reflects the crazed ideas and approaches that were actually taken during the search for a solution.

The main idea behind this research occurred to me in the summer of 1977, while thinking about minds and Cartesian dualism. That main idea was *metaphor*. A metaphor is a process of considering one thing to be like another for the purposes of understanding, manipulating, and describing that thing.

However, there was a step before the strict metaphor step, which I called the 'meta' step. This had to do directly with understanding the mechanism behind the ability of the mind to introspect while still being a mechanical object—which, after all, is the fundamental assumption behind AI research.

In the book, 'Concept of Mind,' Gilbert Ryle [Ryle 1949] argues that to posit a machine-like mechanism behind the mind means that there can be no privileged position from which the individual introspects his own thoughts. If there were privileged positions—the *Ghost in the Machine*—which could observe the activities of some part of the mind (introspection), he argues, then there would be an act of observing, and an act of observing that observing, and so on. If this chain is ever broken, then, he argues, there are some mental processes not accessible to consciousness, which means there is some fundamentally mysterious 'privileged' position; this corresponds to the *ghost* whose actions are responsible for mental behavior but whose activities are beyond observation. Pushing the explanation of mental events to this ghost is not an explanation, but a delaying tactic, in his view. This infinite regression is not possible in a finite organism, so it cannot be the case that the mind is organized that way.

Therefore, unless there is to be a mind separate from the body, then introspection is no different than ordinary observation in a literal sense. So we know our own states in exactly the same way that other people do, but since each individual is able to see a larger percentage of clues such as yawning, which indicates sleepiness, for instance, each person is able to seem as though he were in a privileged position, but this is an artifact of more observations and not better or fundamentally different ones.

In other words, introspection is nothing more than ordinary observations, the difference between 'internal' and 'external' observations is that some observers are in a position to see more of them than others. Hence, since I am with myself more than you are, I know myself better due to more observations.

The key moment was when I decided that the mind might have a privileged position AND must NOT have introspection as a special quality of observation. That is, the type of observations in so-called 'introspection' are exactly like ordinary observation, but the objects of observation are unavailable to other observers.

The two views, so carefully shown by some philosophers to be inconsistent were thus assumed to be consistent and consequences were derived.

So the picture I had was one of parts of the brain having as objects of observation other parts, and that the brain was *stratified* in *observer-participant* layers. This observer-participant pairing I mistakenly (maybe) called the meta-object pairing in the traditional mathematical jargon.

If the brain is a uniform miasma of neurons and connections (in fact, the entire nervous system is uniform), then it is plain 'the brain' observes the world, but it is not plain how the brain observes itself. Think of a program that has some data base of stored facts; each object in this data base can not properly refer to other objects unless that data base is non-uniform or organized in some non-obvious way. That is, in such a data base, there are things which point to other things and things that are pointed to, hence a hierarchy, a heterarchy, or at least some structured organization.

This, of course, does not address any implementation issues, since the memory in a computer is entirely uniform, and the data base ultimately resides there. However, the organization of the memory is different

than the structure of the memory. The organization of memory is different from the structure in that structure is the physical makeup of the system while the organization is the intended pattern of interaction of objects residing in or emergent from the structure. In many ways, this is simply stating once more that representations require several participants: the physical object containing the representation, the real world, and an observer that makes the distinction. That there is a representation in the computer when we deal with it is mostly a product of the observer (the programmer) and not of the computer or of the program. Our *interaction* with that program convinces us of the representation, and NOTHING ELSE.

So, I posited a layering of the brain or mind where each layer is viewed as a observer of the one below. The last layer is an observing layer too, but it observes the world, given some vague notion of observation. With each layer having as its object of observation and discussion some other layer, we have another problem to face, the very problem that haunted Ryle: there are not a infinite number of layers. Therefore we eventually must admit that there is at least one purely behavioral layer and one unobserved layer. Just as there are observers, there must be participants, and in the discussion of observer-participant relationships, the participant is often lost in the noise.

At this point we shown how a mind could self-observe by imposing some organization on the mind in terms of these layers of activity. Interesting and introspective, even self-reflexive, behavior can be seen to be the result of a no more mysterious relationship than appears between the mind and the world, and this latter, well-known relationship is as hidden or privileged with respect to the outside world as is the former depending on circumstances. The imposition of consistency on an inconsistent set of assumptions is achieved.

The problem, now, becomes one of naming or jargon: what should I call the observer/participant relationship, given that it is somewhat vague? There already is a prefix that people have been using for relationships like the one I had in mind: *meta*. Thus there is the *meta-level* of some *base-level*.

The prefix, *meta*, on the other hand, has a somewhat different meaning, especially as used by mathematical logicians. This meaning tends to imply that the base-level, as viewed by the meta-level, consists mainly of syntactic objects only. A standard example is that a base-level object is 'house' and meta-level statement or observation about this object is that it has 5 letters. Now, this tends to give the impression that the only types of statements possible about the base-level objects are statements about their structure.

The difference between the impression above and what I wanted to do was that instead of thinking about 'house' as a linguistic/syntactic entity I wanted to think of it as a semantic entity primarily, so that rather than just talking about the word 'house' as a clump of letters, I could talk about it as a concept of an object which had doors and rooms, viewed as a physical object, and as a concept that could enter into communicative activity, viewed as a word.

In short, the meta relationship can be more fully described as observer/participant. The meta-object includes information such as the description of the object (its function and its structure), descriptions of the slots, rankings of the importance of the slots, pointers to alternative descriptions of the unit, and how to 'invoke' the unit.

The participant object is a behavioral object in that it acts while the observer object comments.

1.14.1 A Complex System Matures

Given the idea of a stratification of sorts, what is to be done with it? Where would the goals or desires of the system be kept, along with its current beliefs about the situation? How would the system get started and how would it decide what to do?

When faced with the problem of making an AI program have certain characteristics, it has been traditionally the case that objects with the names of those characteristics are created and operations that mimic

some facets of the behavior of those characteristics are developed. In Yh the desires and beliefs are contained in a *situation description* in an internal description language.

The basic outline of the system is given in the figure on the next page.

Where is that diagram?

I could not find any source for this figure, and I don't have a hardcopy of my dissertation to use to create one afresh.

Organized as a collection of individuals with descriptions of these individuals existing in meta-objects, the main question concerns how these individuals are selected to act—the question of sequencing. In outline, there is an agenda-like structure to enforce a strict order on some events. Additionally there is a *reactive* component, which is the implementation of the participant portion of the system. In this component there is a description which represents the current attention of the system, and a *matching* mechanism enables the system to select individuals to act as a reaction to that description. I also call this the *behavioral* system.

In this matching scheme, participant objects are selected on the basis of the descriptions of them provided by the observer objects. These objects will henceforth be called *units* after KRL.

Invocation of procedure-like objects does not proceed in the manner common to programs in which the interconnection graph is known beforehand (who can call whom, what data to pass and how, etc), but proceeds by determining the interconnection graph on the fly, and by discovering information-passing protocols as they are needed.

The behavioral system is based on a *hybrid matching* technique that uses a simple locate-by-description mechanism as an indexing scheme. Hybrid matching is matching descriptions made up of *descriptors* and associated numeric *measures*, resulting in a match that pairs descriptors and has a final measure of strength for the match.

The problem is to think of good ways to invoke the units, if they had a procedural component, much in the flavor of languages like PLANNER, where one did not need to know part of the interconnection graph for the system. In a way, this is replacing pointer methodology with name methodology.

This scheme uses a general description of the state of the knowledge of the system and a method of comparing the descriptions of possible reactors to that description. This process is viewed both as a mini-planning process and as a matching process.

The system picks the object with the best match. Hybrid matching is also considered as a sequencing method in some applications, so there is a multi-tiered agenda that can guide and examine the state of the general description in order to judge progress and make up better plans. In fact, a plan can be viewed as a sequence of these descriptions which is then 'reacted to' in order to obtain interesting behavior. In other words, unlike most plans which are of the form 'do x1, then x2...' or even 'do x1 and if c1 then ...,' a plan in this system is a hierarchy of strongly directed clusters of loosely clustered activity. So, the plan might look like: '<general description$_1$> then <general description$_2$> ...' where each <general description$_i$> contains the remnants of <general description$_{i-1}$>, those parts that could not be satisfied. One way of paraphrasing this plan is "thrash around as best you can, using pattern matching on full descriptions in order to achieve this situation, then add to that situation this new set of goals and conditions and do that." The hope was that in this way, some of the detailed complexity of a planner could be buried in a sophisticated pattern matcher, and the actual planning could be simplified.

The generation system itself is organized into groups of units with specific abilities. There is knowledge about how to express facts about programs and data structures, how to make noun phrases, verb phrases, relative clauses, etc; there are observers that watch the initial generation and propose transformations and other actions that modify the text, trasnformations, and text representation specialists. Finally there are scheduling experts that form and modify the agenda and situation description.

In short, there is a large number of experts in the system which must be uniformly accessed.

1.15 Overview of the Generation Process

Remember that Yh is designed to take an annotated program and produce a simple explanation of that program. The following is an outline of the process of generation. Refer to the figure in the previous section.

Yh examines the program, isolating the data structures and code parts, including sub-programs. The program is represented as units in the underlying representation, but with the code represented as annotated LISP code. One agenda item is posted for each data structure and major routine; each agenda item is now a topic to discuss. This corresponds to the gross planning stage, in which the topics and their order are decided. This is a *knowledge-based* step, in which experts in expressing programming language constructs and simple programs produce the plan. The range of programs that Yh is able to discuss consists of the data structures: n-dimensional arrays, lists, sets, numbers, and string; and the control structures: conditionals, lambda-binding, assignments, simple arithmetic, and function calls.

At this point, each agenda item is in the form of descriptions of the sentences to be produced along with descriptions of the modifiers, relevant information about the situation, and advice, either from earlier parts of the generation process or from the user; this advice is intended to be mainly style-modifying information.

Next each agenda item is examined to locate major features. Based on the features located, a sentence schema is chosen. For instance, simple declarative, passive, or compound sentences will be selected, depending on the information about the statement to be made.

Each agenda item is processed until quiescence. This means the agenda item is a *description* in an internal description language of the things to be expressed along with measures of the degrees or desirability and accomplishment for these things. A *pattern matcher* is called upon to locate among the descriptions of abilities of the individuals (units) in the system actions to attempt. Normally, sentences are produced at this point, incorporating modifiers and related statements to be made if possible. Other sentences may be planned or inserted at this point.

1.15.1 More Details on Generation Step

Each sentence is generated left to right all the way down to word choices. Observers—other specialized units—keep track of references to similar objects. By this I mean references to the same object (using identity) and references to objects that are described similarly. Recall there is an internal description language, and a pattern matcher that measures closeness of match; this matcher is used to compare similarity of object by similarity of description.

Modifiers (over and above those inserted by standard NP programs) are added at this point. Specialists for the objects being described may modify the sentence at this point.

New sentences are planned if the measures of leftover modifiers are sufficiently high. When producing noun phrases, for instance, it may happen that advice about verbosity may cut off some of the planned adjectives for a noun phrase. These leftover adjectives may be important enough that special sentences for them are planned or else they may be assimilated into other, related sentences. In Chapter 8, examples of this will be given.

The observation of the activity of generating phrases and syntactic structures may result in a number or proposed collapsings of sentences, substitutions of pronouns for noun phrases, multiple subjects or direct objects, or word choice changes. These proposed collapsings or transformations are evaluated for feasibility and possibly performed as the last stage of generation. This action may propose further transformations, and the process continues until quiescence.

1.15.2 What about the Text?

Representing the text in such a way that the operations required for deliberate writing could be performed without too much problem turns into a real trouble spot. On one hand, I want to do all of the traditional syntactic operations on trees wherever that was nice, and on the other hand I want to do some string-like operations. Since there is no doubt about the 'semantics' or meaning of any part of the output, I do not feel that a fully parsed representation is necessary. And I want to be able to use a good language for describing the transformations that could be performed.

The transformational system I have in mind should not be confused with Transformational Grammar in linguistics. What I have in mind is a methodology in which an initial set of sentences is produced to convey some information; this initial set is then *transformed* to the final set of sentences by increasing the cohesion of the text and rendering unambiguous that which was ambiguous. The string transformation aspect comes from thinking that a good method for specifying re-writes on a paragraph is necessary. This leads to the observation that often exact substrings as well as changes such as:

NP which RELCL VP => NP that RELCL VP

are needed. This implies that something more like surface structure should be the main representation. So a *stratified surface structure* representation is used that allows quasi-string matching to occur. Quasi-string matching refers to matching a given level of a tree structure (fixed depth) as if that level were a string, although the matching truly occurs on a tree structure.

A stratified surface structure is a data structure that is a short, fixed depth tree in which the rows (daughters at the same depth) can be accessed like a string. In a way, this is in a similar vein to the observer-participant relationship mentioned earlier, in that I envisioned a simple sentence where each word had an observer (the class of the word), and sequences of classes had an observer (the phrase of these classes/words) etc. Each observer would tell not only what sorts of things their protégés were, but talk about what they represented and how they came to be where they were.

This data structure is nothing more than a standard tree structure with daughter to parent pointers and sibling to sibling pointers (adjacent at the same level) added.

Relative clauses might appear to be a major problem, since they create a downward component, eliminating the major feature of the representation, However, the downward spanning correlated well with the idea that higher level meta-units were responsible for the introduction of relative clauses as a disambiguating factor. The relative clauses, then, were represented by pointers to other stratified surface structures, which is simply adding a deeper subtree onto the tree in the standard way. In practice, one could allow a fixed depth (Yh doesn't) with little problem by limiting the surface structure to a certain depth, but this would eliminate some interesting sentences. On the other hand, who has ever seen and understood well a sentence with more than 5 embedded relative clauses?

The problem is to model deliberate writing, in which transformations are an obviously good idea. In spontaneous speech, though, it appears that this would mean that the speaker would have to buffer speech for a while. This could be true since for all we know people may process a great deal subconsciously before speaking. Also, even a transformational[11] system can be encouraged to produce left to right with some chunking of output. As in the general planning scenario above, the plan is a sequence of chunks that are reacted to as they are produced. Currently the chunks are somewhat homogenized; that could be changed.

[11] In the sense used here.

1.15.3 What to do if Nothing Matches

I put a lot of faith and responsibility in the hybrid pattern matcher I talked about a few sections ago. I hope you will soon find out why I felt that this faith was justified when I describe the actual matching process that is done. There are many ways to influence the match, and there is never a truly failing match, only ones with stronger or weaker measures. Measuring the strength of a match is referred to as *partial matching*. In this system, the strength of the match is actually computed, and I think that in doing so, I have built a pattern matcher that approaches some simple planning mechanisms in its local hill climbing ability, and one which makes the programming of a very large system easier than most people ever thought possible.

An interesting addendum to the hybrid matching process is that sometimes a good match is hard to find. A match must be made or the system stops working. So, we do a bit of a search with bounds on the branching factor and depth. Resource allocation is necessary since we do not want to search forever, especially when the matches are expensive. So the idea of *counterinduction*, an allocation strategy to be discussed in Chapter 7, came up. Many people will find this strategy puzzling, and it will take some real convincing on my part to get you to believe that it is worth trying. I think, though, that this strategy, in this specific situation of last resort is a good thing to try, and represents exploratory behavior on the part of the system.

As I said in an earlier section, this system is very complex, and the system itself, without considering the natural language generation aspects of it, is a worthwhile object of study. I hope to convince you that, at the least, my system, called Yh, is an amusing alternative approach to some of the other problems plaguing AI at this moment in history. I don't know whether it will be a road that will be explored further.

Chapter 2

Natural Language Generation

In this chapter an example of how to generate a sample paragraph is given, although it cannot be done by Yh. The point is to show the style of generation and the general ideas behind the kind of generation that is attempted by Yh. There is no attempt made to explain in detail how things are represented and where the meanings are kept, only to give a general impression.[1]

2.1 The First Pass

The example is the first paragraph in this chapter; namely, the paragraph before this one. The first thing that is done is that an overall plan is formulated, which is a plan consisting of a sequence of contents for simple sentences. In Yh there is a chunking style based on the units (defined in the Introduction), and it is assumed that this chunking size maps into approximately the right size for a single, relatively simple sentence. These chunks are then generated in a left-to-right manner.

To reiterate, this is an example of the style of generation rather than an example of anything that Yh has actually done. It is intended to demonstrate what is meant by deliberate writing, to give a feeling for the kinds of considerations that give rise to the stylistic processing of the system, and to illustrate some of the kinds of linguistic knowledge Yh has. The representations used here bear no resemblance to those used in Yh. The plan given below is in a notation designed to be easy to read and includes many deductions from the representations actually employed. In fact, it is not possible to adequately represent the full plan for the above paragraph in this notation without it appearing that blatant cheating is how Yh accomplishes what it does.

For this example the initial plan is assumed to exist, and no attempt is made to explain how it might have come to be the way it is. In other words, the entries seen below are the results of a large amount of processing. The point of this discussion is to present the general structure behind the transformational portion of the system. Later, when a real example is given in which the plan is produced by Yh, as well as the English, that will be shown at that time. Here is the initial plan in outline:

1. Subject: I; Action: Give; Direct Object: Example of [Action: Generate [Modifier: How]; Direct Object: Sample Paragraph].

2. Subject: Yh; Action: Can not do; Direct Object: This example Emphasis: This example.

3. Subject: One of several points; Action: Show; Direct Object: Style of generation.

[1] This chapter is meant to be suggestive of the some of the things that a natural language generation could do in order to generate stylistically pleasing text. It is not meant to be a description of what Yh actually does, although the types of considerations presented herein were the motivations behind it.

4. Subject: Next of several points; Action: Show; Direct Object: General ideas [Modifier: About [Subject: Yh; Action: attempt; Direct Object: Kind of generation]].

The rest of the plan is not shown since it requires a richer notation.

The above notation should be clear, but the actual intent—what it is in the system that corresponds to these entries— within the system itself is not. Each entry after Subject, Action, Direct Object, Emphasis, and Modifier is a unit in the system, and the sub-entries enclosed in "[ ]" pairs is the result of tracing links in the unit slot entries along paths of relevance. Though shown here in terms of semi-linguistic entries, this is not the case in Yh itself, but there is some provision made for such entries to exist.

One thing to notice right away is that there is a predeliction for active, simple declarative sentences, which this notation indicates. In addition, several general admonitions are given; one is that the first person singular should be avoided—not shown in the above notation. Another is that the general context should be given, which in this case is simply the chapter that the paragraph appears in.

This plan is then put on the agenda in the order given with generation proceeding left-to-right. The first sentence generated is:

I will give an example of how to generate a sample paragraph.

In this sentence the subject and action produces straightforward surface structure, but the embedded clause has some minor problems, in the form of producing a subordinate clause. The generation of the main clause is done by a general purpose declarative sentence generator, while the subordinate clause is generated by a specialist in incomplete clauses using adverbial modifier: in short, a 'how-to' specialist. Since there is no prejudice against using idiomatic phrases and clauses, the specialist does the job. By prejudice is meant something technical: namely, in Yh there are a large number of experts, some corresponding to idiomatic items. These experts have associated descriptions which mention that they correspond to idioms. Within the descriptive language is the ability to express 'prejudice' against using these idioms, or against using words with sexual connotations for that matter. These mechanisms will be discussed later.

Progress on this sentence is deemed complete, and we move on to the second sentence. This sentence is:

Yh cannot do this example.

Since there is a request to emphasize the example, a notation is made for this sentence to be examined later so that the emphasis can be changed; normally this means a passive transformation. There was a choice of producing the passive sentence now, but this is generally speaking not done since things that happen later can affect this decision. In other words, sentences that *can* be the result of a transformation are generally produced untransformed, and the transformation is explicitly made later on.

Moreover, this sentence is linked to the last by a noun phrase for the same thing, 'the example.' There is yet another semantic relationship, which is that it is a negative statement about something discussed in that sentence. The connections are noted by observers of the generation process, who schedule actions after the first pass to deal with these connections.

The noun phrase 'this example' is produced by scanning backwards for any other noun phrases that have been generated that refer to the same object. In this case, identity of units is used as the criterion for sameness, but in general, this can be the result of a pattern match. In general a noun phrase is the result of producing text about some central unit and possibly some satellite units; determining whether two noun phrases refer to exactly the same object can be done by testing the identity of these central units. Scanning the text so far and examining the meanings of various noun phrases generated or known about at this point can be a help in making decisions about which determiner and other adjectives to use with nouns.

The next sentence is straightforward and is:

One point is to show the style of generation.

Proceeding to step number 4, the main clause goes through the declarative generator with some difficulty only in the verb phrase construction. The subordinate clause is straightforward, but ends up with the focus incorrect:

Another point is to show the general ideas behind the kind of generation that Yh attempts.

This sentence also shares a related noun phrase, 'the points,' one of which is the subject of the previous sentence and the other one of which is the subject here.

The final three sentences are assumed to have been produced with the provision that they started from similar initial representations and ended up with similar surface structures. The ability of Yh to transform sentences that start with similar representations but undergo different processing to result in dissimilar surface structures is limited. However, Yh as a natural language generation system is only a demonstration of the feasibility of the organization for large systems explored in this thesis.

The last three sentences are:

There is no attempt made to explain in detail how things are represented. There is no attempt made to explain in detail where the meanings are kept. There is an attempt made to give a general impression.

These sentences are closely related by noun phrases and similar wording. Not only, as generation goes on, is there concern about the actual referents of the wording, but the actual wording itself can cause a re-planning of the entire structure.

The current paragraph looks like:

I will give an example of how to generate a sample paragraph. Yh cannot do this example. One point is to show the style of generation. Another point is to show the general ideas behind the kind of generation that Yh attempts. There is no attempt made to explain in detail how things are represented. There is no attempt made to explain in detail where the meanings are kept. There is an attempt made to give a general impression.

2.2 The Second Pass

The second pass over the text attempts to process the suggestions made on the first pass. In fact there may be more passes than two.

The first thing that happens in this pass is that the admonition about using the first person singular is acted upon; one way to do this is to make the sentence passive and then drop the prepositional phrase with the original subject. This is also a good opportunity to introduce the context with a simple prepositional phrase at the beginning. This sentence is then:

In this chapter an example of how to generate a sample paragraph is given.

The second sentence again needs to be made passive in order to emphasize 'the example' over 'Yh.' 'The example' is referred to in both sentences, which means that the second noun phrase referring to it is turned into 'it.' Also, since the second sentence represents a negative statement about the same topic as the first, an 'although' is inserted at the beginning. The relatedness of topic, 'the example,' in the first two sentences causes a collapsing of the two in a weak sense. In addition, this collapsing serves to reduce the effect of two passive sentences in a row. The second sentence, then, becomes:

although it cannot be done by Yh.

which is then combined with first to yield the new first sentence:

In this chapter an example of how to generate a sample paragraph is given, although it cannot be done by Yh.

The next two sentences share a common basis for subjects, namely the set of points which has been split into two parts by the initial planning process. The sum of these individual points is *the* point, so these sentences are collapsed, with the common subject 'the point.'

The verb phrases are combined, and the direct objects are conjoined with 'and' to yield:

The point is to show the style of generation and the general ideas behind the kind of generation that is attempted by Yh.

The last three sentences share a common subject and verb phrase, but with the last one a negation of the first two. The general framework is:

<np> <vp> *some direct object* . <np> <vp> *some other direct object* . <np> <not> <vp> *some third direct object*.

The result will be something of the form:

<np> <vp> *some direct object* and *some other direct object* <contrasting word> <not> <vp> *some third direct object*.

The verb phrases in question are, *no attempt is made to explain in detail* and *attempt is made to give*, with the first appearing twice. The collapse of these first two occurrences is simple, but collapsing these with the third is a bit of a problem if, in fact, there is no specialized knowledge about the phrases in question. A combination of formal methods and specific knowledge can be brought to bear: the largest common subphrase is *attempt is made to*. This is ill-formed because it ends with half of an infinitive, which must also be stripped off. Thus, the idea of collapsing phrases by locating the largest common subphrases and criticizing it can be used. The technique really used, though, is to use the largest common subphrase within the context of larger linguistic units.

The final three sentences combine, using 'only' as the contrasting word, to:

There is no attempt made to explain in detail how things are represented and where the meanings are kept, only to give a general impression.

2.3 Multiple versus Single Sentence Utterances

Yh produces multi-sentence text rather than single sentences. The ability to spread out a meaning over several sentences allows the generation system to create cohesive text, which is a phenomenon mainly of multiple sentences. Part of the problem of creating cohesion is solved with the aid of revision techniques, in which it is noticed that similar objects are being discussed or similar phrasing are being used. This criticism is followed by a repair phase that attempts to rectify any bad situations in the text.

The claim is that cohesion is the product of self-observation and layered behavior. Moreover, the claim is that with a rich set of abilities, and the freedom to choose, the idea of *judgment* figures heavily in the behavior of the system. The goal in creating Yh was to quantify judgment, not define what good judgment is.

2.3.1 Single from Multiple Sentences

If one is to generate some fairly complex situation with a single sentence, then a large amount of planning needs to go on in order to ensure that no ambiguities crop up. For instance, suppose that one wanted to say something like:

A car broke down during the snowstorm.

The driver tried to fix the car.

He didn't have the right tools.

He borrowed them from someone who's car had broken down nearby.

He fixed his car.

A single sentence for this event would be something like:

During the snowstorm the car broke down and was fixed by borrowing the right tools from a nearby car that had also broken down when it was discovered that the correct tools were not in the first car.

Think of how this could be generated without any intermediate multiple sentences. It is unlikely that a schema for this sentence would be present in any system. And if it were, then it is always simple to construct other examples which would require schemata of arbitrary complexity.

It is of course imaginable that a single sentence generator could generate this sentence and sentences like it from scratch, using sophistcated planning techniques.

One of the points of this thesis is that if one allows multiple sentences from the beginning, the approach of making a rough plan with many islands that are locally planned using other, descriptively based techniques, and with the awareness of what is being said at what point, that the power of the generator will be increased easily, *once the threshold of the complexity is transcended.*

When a wide range of techniques, each well suited to some moderately limited class of problems are combined, the problems of cleverness and close planning are replaced with judgment and choice in a large domain.

2.4 The Generation Process

Generation, in the view presented in this thesis, is a planning and execution problem in a rich domain in which many types of actions and choices are available. The main thrust, then, is to make good judgments and to try to recognize familiar situations so that specific knowledge can be brought to bear at the opportune moment.

To accomplish this, the technique of doing several passes with a formulate and transform paradigm is used. A combination of both purely syntactic and semantic information is used to perform the transformations required. Both simple and complex sentences can be initially generated, so that, for instance, passives are not obtained only through transformations.

The entire process is couched in the observer-participant framework discussed earlier: there is a behavioral component which produces in a fairly simple and straightforward manner a version of the text that can pass as understandable, though idiosyncratic. The process of doing this level of generation, then, is subject to the observation of other processes, which notice both semantic and syntactic facts about the text, and which make recommendations about modifications to that text.

Thus, in real generation, the expectation is that very large sentences are produced with only relatively small portions generated by contemplative techniques, and, in the case of writing versus speech, subtle transformations are made as well as large scale, plan-altering ones.

At once it is plain that the first pass, and the left-to-right portion of that pass, involves generating down to actual word choices, since the choice of an earlier word will affect the choice of a later one. The choice of a word or set of words can also signal a real or potential conflict (ambiguity) between one part of a paragraph and another, information that might be weeded out in a later pass.

To become convinced that word choice can drastically alter the plan for an entire paragraph, consider what happens if a number of key verbs are missing but the event forms are not. For example, suppose the word 'collide' is missing but 'collision' is present. Any plan that expressed the action 'collide' would need to express the event of 'collision' instead.

2.5 Vertical Strategy

The idea of producing several sentences or parts of sentences one after another in order to express some situation is called the *vertical strategy*. It is the strategy used, roughly, by very young children to express themselves, especially when the only linguistic knowledge they have is in the form of words. One could expect to find that, developmentally, the vertical strategy is horizontalized through the use of collapsing transformations on existing text, and through the need, arising out of unspecified channels, to eliminate potential ambiguities by making references that are obviously the same identical (with pronouns, etc).

2.5.1 Rich Domains

In a rich domain where there are many ways to do things, the main issue is judgment and choice, and removing or lowering the preference of some abilities over others will yield a difference in behavior, not a real breakdown. When one removes from Yh the fancy stereotyped structures, its behavior returns to the simple declarative with transformation model, and when the declarative sentence and noun phrase generators go away, simple word choice only is available, and the behavior is child-like in some ways, in that it says simple words to express itself.[2]

This behavior is not a coincidence, but was planned into the organization of the system; the planning is the result of a wish for robustness. The widespread use of full descriptions of the abilities of individual units and the use of a hybrid pattern matcher that measures the closeness of a match and controls behavior in Yh helps the system locate any relevant actions in the face of inability.

2.6 Cohesion

This section describes some of the cohesion techniques that Yh uses. Examples of these techniques are given as well as some of the techniques that Yh uses to accomplish them. Again, since Yh, as a generation system, is quite primitive, there is no attempt made to demonstrate a wide sophistication of linguistic ability. There is a massive body of literature cataloguing, for instance, the exact uses of conjunction and when it is applicable; Yh is able to perform only the smallest number of them. The idea behind the generation abilities of Yh is to test the organizational ideas in the system and to see how they, combined with the limited linguistic capabilities programmed in, result in interesting linguistic behavior.

Cohesion is the characteristic of a set of sentences which cause it to be a *text* rather than a set of disjoint sentences. This characteristic is a semantic rather than a syntactic one in that it is a network of interrelated references that make a cohesive whole, rather than the presence or absence of shortened or pronominalized referents.

Linguistically speaking, a *text* is any piece of writing or speech that forms a unified whole, regardless of length [Halliday 1976].

As in the example about apples in the previous section, the cohesion of a piece of writing can come from the use of signals of *co-reference*—two phrases that refer to the same object. There are certain linguistic techniques for doing this, such as pronouns, similar (but not necessarily identical) wording, use of definite articles, etc.

The existence of text means that it is not likely that one is able to say everything that is desired with only one reference to the participants. Hence, there is a need to continually refer back to things (or forward to things) during a discourse.

[2] An example of this behavior will be given in Chapter 8.

2.6.1 Endophoric Reference

This is reference to something that has been introduced in the text earlier—it is a textual reference. The dichotomy is between endophoric and exophoric reference, with all other types of reference falling under the category of endophoric.

2.6.2 Exophoric Reference

Exophoric reference is a situational reference, and makes use of the common context of speaker and listener. Thus a sentence like:

That certainly is a green iguana.

contains the exophoric reference, *that*, which presumably refers to an iguana in the common view of the speaker and listener. The only attempts at exophoric references in Yh occur when the program that is generated, and its subordinate parts, are referred to. Thus, a phrase like *the program* is an exophoric reference, and the fact that the reader understands this reference is built into Yh.

Additionally, if there is any *uninterpreted* text associated with parts of the internal representation of the program, this is taken as an exophoric reference and used as such. Such text is the result of the parse of the English dialogue between the user and the synthesis system; this text serves no purpose other than to introduce an abstract object that the synthesis system will need to possibly represent and define operations on during the synthesis phase. The parsed form of the text is available to Yh. So, with the example in Chapter 8, *the flag* is a phrase known by Yh to be some way of referring to an object in the world that the reader is familiar with and that is related to an array that was generated by the synthesis system to represent that external object in the program. When referring to the array, then, Yh will use this exophoric reference for clarity.

2.6.3 Anaphoric Reference

Anaphoric reference is referring back to some previously introduced referent, using one of several techniques. Generally this technique uses some pronoun-like phrase to do the reference or a shortened or substituted form for the noun phrase.

In Yh this is handled by the observers of the initial generation; the model is that most anaphoric references are produced in deliberate writing by generating noun phrases for the same thing and noticing that this is occurring. One or the other of two such references may be patched with a pronoun. When this happens the intermediate text is scanned to determine if any intervening noun phrases can be confused with the pronominalized one by invoking the pattern matcher (see Chapter 6) on the descriptions of these noun phrases. Distance between the references is also taken into account, where distance is defined in terms of the number of intervening words and intervening noun phrases.

A related problem is that of determining whether two noun phrases interfere with each other by referring to different things which can be phrased similarly. With the same pattern matcher the similarities and differences can be identified, and distinguishing characteristics expressed. This particular type of behavior is not performed well by Yh, and a clearer discussion can be found in [Levin 1978].

2.6.4 Cataphoric Reference

Cataphoric reference is like anaphoric reference, but refers forward to the reference that will clarify identity. Thus in

This will surprise you: I never thought iguanas could be so friendly.

this is a cataphoric reference to the sentence following the colon.

Cataphoric reference is handled very much like anaphoric reference at the observation level: the decision about which reference to pronominalize can go either way, though referring forward is usually only when the forward referent is some larger piece of text that is being introduced.

2.6.5 Substitution

Substitution is referring to some object by substituting some other phrase for the full descriptive phrase. An example is:

My iguana is sick; I should borrow a lively one.

where *one* is substituted for *iguana*.

Again this is handled by the same mechanism as anaphoric reference. However, this type of reference is not limited to references to the same object, but applies to references to the same class of objects. So the test for appropriateness of substitution requires that the second reference be to one of the generic type of the first reference. This is accomplished by comparing the descriptions of both units.

In this case the original sentence would have been:

My iguana is sick; I should borrow a lively iguana.

The system notices that *my iguana* is a particular member of the class *iguanas*, though *a lively iguana* is a subclass of *iguanas*.

2.6.6 Ellipsis

Ellipsis is substitution by a null string. Some forms of collapsing phrases in things like appositive gerund lists are done this way.

Joan brought some iguanas; Catherine some cattle prods.

Ellipsis falls into the category of a *collapse* in that it is the result of noticing that substantially the same surface structure is being re-generated or that the same basic sentence schema appear in two places. As in the case of substitutions, the overriding principle is the elimination of repetitions in text. It is the nature of English that it is not possible to remove repeated object references though it is possible to remove repeated action references.

As in the case of substitutions the mechanism for collapse is the observer who notes that the same phrasing is being generated. Later a collapse of the appropriate type can be scheduled if necessary.

2.6.7 Conjunctions

Conjunctions are used extensively in Yh to collapse sentences and phrases. They are used to bring closer together separate noun and verb phrases, putting conjunctions in to emphasize commonality.

I have an odd affection for cattle prods and green iguanas.

Conjunctions include such bridging words as *yet, but, so, however, since,* etc.

This is simply another collapse, but one in which object references are combined with conjunctions, to more explicitly show the commonality of action. Normally, conjunctions are appropriate most when there is a doubling of verb phrase and either subject, direct, or indirect objects (and predicate adjectives).

One of the other considerations for collapsing is the distance between the two candidates. One expects that normally there is some intervening facts that need to be presented before the second sentence can be stated. Thus, there is a question about how much of these facts should be left questionable in the reader's mind in order to perform the collapse. For example, the paragraph may be:

The program inputs a list of numbers. The numbers are sorted. If the smallest number is less than 0 the program exits. Otherwise the program inputs a list of symbols.

Here the first and last sentences could be collapsed to:

The program inputs a list of numbers and possibly a list of symbols.

At this point an explanation of why *possibly* is called for. Yh is only able to judge this case with heuristics about the interference caused by the collapse. In the full generation system one might want to be able to save the state of the generation and test out the re-write of the remainder of the paragraph, which might end up being:

The program inputs a list of numbers and possibly a list of symbols. The list of numbers is sorted and if the smallest is not less than 0 the symbols are read in; otherwise the program exits.

2.6.8 Collocation

Collocation is an interesting sort of cohesive technique, which is available to Yh for use. It involves, mainly, word choice, and is the tendency to choose words that have a slight descriptive leaning in a particular direction.

His love for animals was unusual for someone raised in the city, and not just warm-blooded ones, but cold-blooded as well. It chilled her to think of the passion he demonstrated at times, the heat of longing that flowed through him when he was near the helpless creatures, feverishly petting them. But what froze her heart the most was the time he called room service and ...

This example points out not only collocation, but the other types of cohesion. The collocative aspect is the influence to use words that refer to heat and cold, or have that connotation, such as *warm-blooded, cold-blooded, chilled, feverishly, passion, and froze.*

The idea is to make the text cohesive by giving its parts a similar tone. In some ways, Yh is best at this sort of reference, which is mainly located on the word choice level. The descriptions of the words in the lexicon are the richest in the entire system, and the ability to allow the choice of earlier words influence the choice of later words is powerful. Hence, collocation is relatively simple.

The word choice problem is central to English generation, especially when non-technical prose is the aim. Words in English are like pictures rather than like hieroglyphs, as pointed out earlier. In Yh extensive descriptions of the meanings and connotations of words are provided, and the choice of an appropriate word is a matching process of the requirements for the word and the descriptions of them. The view of words is that each one is a fuzzy region and not a point in the space of meanings; the latitude to select within regions constrains the structure of a text the same way the structure of the text, likewise, constrains the choice of words. There is an interaction between these choices and constraints in both directions, to which Yh attempts to respond.

Chapter 3

Representation

Yh is organized as a collection of objects called either 'individuals' or 'units.' They are called individuals because of their communication abilities and their autonomy, and units because of their packaging nature. These objects are able to communicate with each other in moderately interesting ways and are independent of each other in that the structure of each others' contents is not generally not available. 'Packaging' means that these objects are considered to be the smallest grain size relevant to the overall organization of the system.

3.1 Stratification

The basic model of cognition used is called the *stratification* model. It is called stratification because the model is organized in different layers or levels of activity. These levels are composed of *units*, which are individuals with certain abilities. Most individuals or units have an associated meta-unit, which is a semantic *observer* of the unit. By observer is meant a unit whose object of expertise is the semantic content of the *base-unit*. To say that the meta-unit observes in the traditional sense is imprecise because there is no concurrent way for one object in a system to "see" activity.[1] This doesn't mean that the effects of a unit cannot be observed in a good sense.

3.1.1 Shallow Stratifications

Though possessing an unimpressive name, these stratifications form the core of the system, having a more immediate impact on the system as a whole than *deep stratifications*.

A shallow stratification is a *tower* of unit, meta-unit, meta-meta-unit, . . . units which share the base-unit, meta-unit relationship all the way up in the obvious manner. This type of arrangement provides a closer coupling between units than does the deep stratification (see next section), which is an amorphous relationship. The exact role of the objects in a shallow stratification tower will be made clear later in this chapter. In the rest of this thesis *unit* and *stratification* will be often confused, but with the context it should be clear whether the tower or the base-unit of the tower is meant. If the term *stratification* appears with no modifiers, it is assumed to mean *shallow stratification*.

[1] The way that observation takes place is by communication—message passing—or by watching the description of the current situation. The former is supported by primitives in the underlying system and is a description based message passing mechanism. The latter is based on a description maintained by the system of the current goals and facts of interest to the system. By observing the changes in this description parts of Yh can monitor other parts.

3.1.2 Deep Stratifications

As pointed out above, there is a close coupling of units which are related by the 'meta' relationship. In fact, one may consider that a set of units related this way form one component or *tower*, with responsibilities divided among them. Deep stratifications are sets of shallow stratifications, organized in a very loose way with the common point of being observers of activity of lower stratifications. Thus, the organization of the system is layered according to these deep stratifications. In Yh, these are roughly organized along the lines of traditional linguistic entities in the standard hierarchical ways—words, phrases, clauses, sentences. Higher level stratifications are for adjectival phrases, relative clauses, and other modifiers of the original levels.

Since the observational groups generally form a tower of observers/observees without loops, these groups form a second, deeper stratification.

More specifically, there are units that generate noun phrases, and there are units that observe this generation. These two groups form two stratifications, with the former being a higher level stratification than the latter.

The reason for this type of breakdown is that some sophisticated linguistic behavior is the product of observation of a simple-minded generator giving rise to modifications. So, when a phrase is generated that is ambiguous, an observation is made of this, and a higher, deep stratification level modifies the ambiguity.

Therefore, there are two types of observation that take place in Yh: the observation and description of units by meta-units, forming shallow stratifications, and the observation of groups of units by other groups. In short, then, deep stratifications are the mechanism of observation by one part of the system of another.

3.1.3 Example

In Yh an example of a shallow stratification is the unit which represents a word (say a noun) and its meta-unit which contains the description of that word (in the system-wide description language to be discussed later) and the methods for interfacing that word with the rest of the text representation. So, for a simple noun, the base-unit contains the various forms of the word and little else. The meta-unit contains the description mentioned along with pointers to synonymous words. The interface in the meta-unit is able to place the word in the representation given various specifications of the location for insertion.

The lexicon (set of known words and phrases) is a deep stratification that is the object of observation of most of the rest of the linguistic part of Yh. The transformations known to Yh form another deep stratification that observes the first pass of generation. The transformation system observes the initial process of expressing and proposes transformations. This is done by observation rather than by embedding this information in the initial pass because one goal of Yh is the separation of concerns: the transformation stratification must be able to be improved and added to without having to alter the first pass.

3.1.4 Units

Units are data structures in the KRL, Frame [Minsky 1975], and FRL [Roberts 1977] style in that there are slots and fillers. These slots contain data (fillers) as well as descriptions and procedures of a standardized nature. Each deep stratification has a set of locally meaningful slots and filler semantics, as well as defaults within the stratification. However, in terms of the implementation, the slots are whatever the user of the system wants given that the small set of system-wide slots contain the right sorts of fillers. In a deep stratification that represents some facts or objects in the world, there may be slots that convey the standard inheritance that KRL, for instance, does. In deep stratifications that are internal (e.g., represent knowledge Yh has about itself) inheritance may be absent, though there is local consistency of slots within each stratification.

Not only does the meta-unit contain some description of the unit, but it performs certain prescribed functions for it. Namely, it 'advertises' the object unit's purpose in two different ways. The first way is in the form of simple patterns, to be described in Chapters 5 and 6; the second is in the form of a description language based, in part, on this pattern language. These descriptions are used to match desires against abilities, and is, hence, a method for the system to reason about its own abilities.

The meta-unit also describes the contents of some of the slots in the object unit. This description is not a description of the syntactic qualities of the slot, but of the semantic usage of it. Since meta-units are still units, they have slots and fillers, with the difference being that, mainly, slots of meta-units are filled with descriptions rather than data. A *description*, as used here, refers to an object in the system description language; as such, it is strictly data, but it is data with a standard set of uses. Namely, through the descriptive system, Yh is able to reason about itself and use units from stratifications with differing standard slots. Sometimes the slots of the meta-unit describe qualities of the object unit as a whole, and other times the slots describe qualities of the slots of the base-unit with the same name. If a meta-unit and a base-unit contain slots with the same names, it is assumed that the meta-slot describes the slot; if the name of a slot in the meta-unit does not have a correspondingly named one in the base-unit, it is assumed that the meta-slot describes the base-unit as a whole.

The distinction made is that the meta-object is in the relation of an observer or commentator rather than a meta-object in the standard mathematical terminology. This facet will be discussed later in this chapter. The correspondence between a unit (and what it represents) and reality (for example, the existence of that object in the real world) is explicit in the meta-unit and never implicit in the unit. By this is meant that the closed domain assumption is not made: if some fact is absent (where a fact is represented in some standard way via units) then it is NOT assumed false. Furthermore, the fact that some unit representing something is present MAY NOT mean it is true. The explicit 'word' of the meta-unit can make a 'fact' true or false.

3.1.5 Procedural and Declarative Components

A unit is both procedural and declarative in its basic nature. That is, there is a slot named 'method' which is explicitly intended to be executed by the interpreter for the language underlying things.[2] Other slots are primarily declarative by intention. For instance, meta-units contain 'purpose' slots, which describe in simple terms the purpose of a unit. These purposes are normally used for call-by-intention by the procedural portions of the language. This is akin to planner-style calling. Other slots have stereotyped usages. For example, there is a 'level' slot which states the level of the unit in the overall scheme. These levels are incremented for each level of meta-unit. The level is used as a heuristic measure of 'applicability' in that the higher the level number, the more general the applicability of the unit, and the more likely it will be used in a situation where a search for the best method is in progress. The slots 'meta' and 'β' refer to name pointers to the meta and object units respectively.

Not all units need to have a procedural component (and such things as the agenda and system-goals units do not have procedural parts). The meaning and usage of these specialized units will be discussed in Chapters 4, 5, 6, and 7.

The set of units in the system are not exactly in a network since no pointers are used, and names are only used to refer to meta units and other closely related clusters. Referring to other units is done via descriptions in the internal description language, and identification is accomplished through the pattern matcher (Chapter 6).

[2]There is a programming system or language that was written to support the unit structures and the various other facilities that will be described later in the thesis. The system and language is called META. An interpreter for this language was written in order to allow good control over the behavior of the system for experimentation. For instance, several multi-processing primitives were used in the discovery (by me) of searching techniques eventually coded in the underlying LISP.

3.2 The Meaning of Slots

The meaning of a slot is, implementationally speaking, up to the user of a system based on this representation; however, like any interesting system, there are a number of pre-determined meanings for some slots and some units.

Later some standard units will be described. The format of the descriptions of the slots is: "<slot-name>:<description of contents>." Examples of these slots will appear later in the chapter and in Chapters 5 and 6, this section being an exhaustive listing of the standard slots.

Level: The 'level' slot refers to the supposed generality of the unit. In the stratification model, already briefly discussed, the higher the level a unit (or a stratification) is, the more activities it surveys. The level slot, in actuality, is used in a very limited way by the search and invocation algorithms. The overall level at which a unit exists is a rough indication of the deep stratification that that unit is on. As more of the system is described, the various deep stratifications will be pointed out.

Meta: This is the name of the meta-unit for a unit. It may be absent, which indicates that only default slots appear or else some other special knowledge is available. Sometimes it is the case that a unit has what amounts to a meta-unit but does not explicitly know about it. Normally, though, this is handled with the ALTERNATE-METAS slot.

The meta unit describes both the semantics of the unit as a whole and of the slots in particular. Often, only default slots are used, which normally do not have descriptions of them in meta-units, the semantics being defined operationally.

As will be seen, a unit may have several meta-units, which are used for alternate interpretations for the base-unit.

β: The β slot points to the base-unit for this unit. Normally it implies that there is a meta slot in the unit the β names. As in the case of the META slot, this is a name not a pointer.[3]

Purpose: The purpose slot contains information for indexing the units. As will be pointed out, there is a fairly complex unit matching and choice facility which uses some of the default slots mentioned below. The purpose slot is used to index into the data base of units in order to avoid the expensive matching that can occur on irrelevant descriptions.

This slot contains a list of descriptions in the form of a tree structure with, basically, constant data and free variables that can match non-constant data. Each description is, then, either an atom or a binary tree (usually a list). Though internally treated differently, these atoms and trees form a keyword which can be used to retrieve this unit. An example entry in a purpose slot is:

```
(...(transform a ? into a ? *)...)
```

which states that this unit has something to do with transforming a something into a something else etc. So a list that would match this could be:

[3] Pointers and names are conceptually different things which can be implementationally indistinguishable at some level. That is, in most implementations, a name, as used to refer to another object, is really a pointer to a symbol (or an atom in LISP) that is then the name of the referent. Conceptually, a name is a an object in a system such that finding the referent requires further computation, while a pointer is the caching of the result of that process. So, in the case of a name being a pointer to a symbol, that symbol then must point somehow to the referent. The 'somehow' embodies the processing mentioned above. In addition, the untangling of referents occurs at runtime and not at load time, which means that it is easier for the system to self-modify. In Yh the former is useful because that enables the referents to vary according to context, and the latter enables the system to self-modify a bit.

```
(transform a declarative into a passive using ad hoc techniques)
```

Remember that the purpose slot is only an index in locating candidates for the full pattern match, and that a much more extensive and interesting match is done in the general case of identification.

Method: This is the entry for the procedural part of a unit, if present. The code that appears here can be anything that the underlying Lisp can evaluate, but it is normally written in a language that the system interprets specially. This language is basically a lexically scoped Lisp-like language much like Scheme, but is also equipped with several parallel primitives, some co-routine primitives, some special interrupt facilities, and the ability to invoke units based on various types of descriptions.[4]

This programming language will be explained as needed in the main part of the thesis.

Reply-method: It is possible, in the underlying language, to have two units carry on what amounts to a conversation. This conversation proceeds by a message passing protocol. Essentially, a unit uses either the primitive indexing scheme or one of the more complex matching schemes to choose a unit to converse with. Then messages are passed back and forth, with processing occurring at will in each unit. Normally, a unit will take a message, match it against the various things it knows about, and return what it believes is a reasonable response. The reply-method is the slot containing the code that carries on this conversation. In the descriptions of generation, the roles of some of these conversants will be discussed.

Goals: This slot contains a description of the unit in full terms. It is part of the description that is used in the sophisticated matching process talked about in Chapter 5. This entry is a list of descriptors, where each descriptor is of the form `(<pattern> . <value>)`. `<value>` can be any expression that returns an integer, but is usually simply an integer.

Dotted pairs & computer memory size

Lisp people might notice I use dotted pairs where straight lists would be easier to read. Dotted pairs take up less memory space than a full list with two elements. The PDP-10 I was using in the late 1970s had 256 !kilowords of memory, which is roughly one megabyte. One megabyte! By way of contrast, the computer I'm typesetting this on has 192 gigabytes of memory. Future readers might wonder how I could get by with just 192 gigabytes. It was tough.

In short, what this slot contains is an "advertisment" about the abilities of the base-unit (this is normally found in a meta-unit). This description is encouraged to be as broad as possible, with the onus on the matcher to wade through the possibilities.

This advertisement or description is normally in the meta-unit because it is a statement about the base-unit and, hence, properly belongs in the meta-unit. However, in specialized applications, such as in most of the lexicon for Yh, where efficiency is more of a problem, one puts this description in the base-unit in order to avoid defining a meta-unit. As will be seen in later chapters, one of the sequencing techniques is to match goals against abilities, which means matching some data structure containing the system description

[4]The original intent of this programming language was to be able to control the underlying language to a greater extent than I could control the underlying Lisp, although at the time of this research I was the maintainer of that Lisp. I wanted the various multi-processing features, but I put a much higher importance on them at the start of the research than I do now. Currently, only some of the co-routine type of multi-processing, which I use for conversations between units, is really necessary for Yh to work. The major interesting part of the language at this point is the call-by-matching-descriptions primitives, which are widely used. Much of the code in Yh proper is compiled MacLisp. However, as a methodology for exploring that mysterious technique, counterinduction, the language proved quite valuable. A minor consideration was to make the code more easily transportable.

of desires and facts with the advertisements of the units. The unit that holds the most appropriate advertisement is then invoked, and this is normally the meta-unit. The rationale is that the meta-unit may want to observe the activities of the base-unit and take corrective action. Thus, there is no necessary reason to put the advertisments in the meta-unit.

Identity: This slot is in exactly the same format as GOALS but is used to describe the identity of the unit rather than the functions the unit performs. This point is very important: Yh has no built-in knowledge about what sorts of things exist in its 'data base' of units, so that some mechanism is needed to find this out. There is, however, a built-in language of descriptions that is used to describe these units. By matching these descriptions Yh is able to discover its own abilities. The description of the unit is what is meant by the 'identity' of the the unit.

Pre-conditions: The pre-conditions slot contains a description of the same form as the GOALs slot, but its contents is a description of the pre-conditions that ought to be true in order for this base-unit to work best. 'Best' in that a pre-condition does not NEED to be true in order to invoke a unit, though it is nice.

Added-goals: These are the updates to the system description (a specialized unit which describes the state of the system and what it is trying to do). again, it is used by the matcher and by the counterinduction process.

The PRE-CONDITIONs and GOALs are also used in the counterinduction process and will be described in Chapter 7.

Constraints: The unit selection process roughly attempts to match abilities against a situation, where the emphasis in the situation is on what is desired to happen. The matching process referred to above can be influenced by other things. This slot and the next several are such influences. Without an understanding of the matching process, the descriptions of these slots is difficult, so only a very general comment will be made about each one.

The constraint slot contains predicates that must be true for the match to succeed.

Bindings: This is the set of bindings that the matching process is expected to make if the match is successful. The matching process is also a method of function calling and parameter passing.

Soft-constraints: The matcher works with a measure of strength of a match. A soft-constraint is a predicate and a measure such that if the constraint is true, the measure is added to the final value for the match. The measure associated with the predicate can be a function that returns an integer.

Countergoals: This slot contains a list of patterns, such that if any of these patterns match any of the objects paired with items in the GOALs, the measure is added as above. This is intended to be an exception to the rule, so if, in a simple blocks world application where blocks are moved around on other blocks and on a table, a unit will put a block on another, but not on a specific block without difficulty, a countergoal can be used to reduce the measure of the match.

Preference: This slot is a list of the form `(...(<name> <predicate> <amount>)...)` such that for each entry, if the predicate is true then `<amount>` is added to the measure. Additionally, there is a queue of unit names and preference names along with factors and thresholds such that at regular intervals the factors are applied appropriately to the preference in the unit named, and when the threshold is passed, the preference

is deleted. Thus, there is a decay mechanism associated with the preference slot, which is not available for the soft-constraint slot. Additionally, countergoals and influences (see below) can be decayed.

Pairing-function: In the matching process, the items in the system description are paired with those in the GOALs. The situation description is a specialized unit that is used to hold a description of the important aspects of the current situation; this unit is matched to unit descriptions in many places. There is a default pairing function, but if the user wants another one to be used when matching this unit, the name of the function is in this slot.

Influences: This slot is of the form `(...(<pattern> . amount)...)` and is like COUNTERGOALs except that the pattern is matched against various things in the system description, adding `<amount>` to the measure.

All of the measures mentioned in this section can be functions if the flag `-functional-measure-` is true.

Descriptors: This slot appears in meta-units, and is a list of the form:

```
(...(<unit-name> . <amount>)...)
```

such that `<unit-name>` is a unit which is a prototype for the base-unit, and `<amount>` is the measure of the strength of this descriptor.

Rankings: This slot occurs in the meta-meta-unit of a unit and is of the form:

```
(...(<slot-name> . <amount>)...)
```

where `<slot-name>` is the name of a slot in the base-unit and `<amount>` is the measure of importance of this slot in the representation of the object (if there is one) that this shallow stratification represents.

Description: This slot occurs in the base-unit and is essentially a user-defined slot, which is assumed to be of the form:

```
(...(<pattern> . <amount>)...)
```

which is intended to mean something like each `<pattern>` is some description of the object represented and `<amount>` is the strength of the belief in this description. One difference between this and DESCRIPTORS above is that the semantics of DESCRIPTION are user-defined while the entries in DESCRIPTORS are other unit names; another is that the DESCRIPTORS slot is a name-pointer to a prototype for the unit which has some more general information. A practical example is that if some unit represented a particular elephant, the DESCRIPTION slot might say that this one is green, while the DESCRIPTORS slot would say it is an elephant. The rationale for the location is these slots is that the DESCRIPTION slot comments on the world ("is a green elephant") while the DESCRIPTORS slot comments on the unit ("represents an elephant").

Workspace, Global-workspace: These slots (and the associated plurals: WORKSPACES and GLOBAL-WORKSPACES) are useful for repeated reference to a slot in some other unit by the procedural component of some unit. When invoking a METHOD slot, any entries in these slots in the meta-unit make the names of the slots in the units available by simple variable reference. Thus, if any unit with some slots is attached as a workspace, one can change the values of the slots with, essentially, `(setq <slot-name> <value>)`.

A distinction is made between a *local* and a *global* workspace. This distinction is that a global workspace leaves the unit in its natural habitat, while a local workspace copies the entries from the named unit and retains them in a disembodied manner. Since there can be many workspaces from many units active at any time, the slot names are pooled, and the contents reflecting the most recently added workspace. Assignments refer to all workspaces and reference takes the first found, where the local workspaces are searched ahead of global workspaces; then each category is searched in most recently added order.

3.3 Some Standard Units

There are a small number of standard units built into Yh. These units have mainly to do with sequencing, system description, and the text data structure. The first two will be described in detail in Chapters 5, 6, and 7, and the text data structure in the chapter on stratified surface structure, which is Chapter 4.

Agenda: This is really a shallow stratification of indefinite height, which is the main sequencing mechanism in Yh. The base-unit contains an agenda as one of its slots, the agenda tower also being a deep stratification with meanings for the slots known to itself. Each time a new task is to be performed (taken off the base agenda), the highest level in the stratification for the agenda is run. The intent is that the meta-agenda will re-order things on the base-agenda. In fact, the meta-agenda has been used in the generation examples (and other examples) in this thesis only to take responsibility for deciding when an agenda item in the base agenda is completed or not completable at all.

The idea of having a meta-agenda to allow a system to actively observe and reason about its own sequencing abilities is also used by Stefik [Stefik 1980].

System-goals: This unit, again a shallow stratification of indefinite height, contains the description of the *attention* of Yh. This unit contains entries for the facts that are believed to be true, the extent of that belief, the activities that are desired to occur, the extent of that desire, and various other influences on the behavior of the system, and the extent of those influences.

This unit represents the secondary sequencing technique, which is to establish a set of system-goals and let the *reactive component* of Yh respond to that set. This will be described in Chapters 5, 6, and 7.

Output-buffer: This is the unit that contains the text that is being generated. This unit is almost always a global workspace and the stratified surface structure primitives are simply applied to the appropriate slot.

3.4 An Example from Yh

The following units are from the input of the Dutch National Flag program description. Not all of the entries will be described In detail since the semantics is operationally defined.

```
{dta-str1
 level 1
 meta meta-dta-str1
 markers
 (dta-str2 dta-str3 dta-str4)
 type array
 element-type (one-of (R B W))
 dimension 1
 length N
 first-element 0
 last-element (1- N)
 name flag1
 base 0-based}

{meta-dta-str1
 level 2
 β dta-str1
 element-type
 (((an element-type) . 1000))
 meta meta-2-dta-str1
 data-structure-for program1
 descriptors ((array . 1000))
 represents ((flag1 . 1000))}

{meta-2-dta-str1
 level 3
 β meta-dta-str1
 element-type
 (((R represents RED) . 900)
  ((B represents BLUE) . 900)
  ((W represents WHITE) . 900))
 rankings
 ((type . 500)
  (name . 550)
  (element-type . 400)
  (dimension . 525)
  (markers . 425)
  (length . 950)
  (first-element  . 300)
  (last-element . 300)
  (base  . 525))|
```

These units represents an array, which is used to represent the flag in the program. The name of the base unit is DTA-STR1, and is a zero-based, one-dimensional array, of length n, with three array markers into the array, with elements selected from the set {RWB}. An array marker is simply an index into the array which is used as a place keeper.

The meta-unit is META-DTA-STR1, which states that DTA-STR1 is an array, with maximum confidence. Two user-defined slots indicate that this is a data structure that is used by PROGRAM1, and that the array represents FLAG1, which is the flag in the problem. The ELEMENT-TYPE slot just says that the ELEMENT-TYPE slot in the base unit is an element-type.

In the meta-meta-unit the rankings indicate the importance of the slots. The range for the measures are: $-1000 \leq m \leq 1000$. The ELEMENT-TYPE entry is a statement about what the various entries represent, which is a statement about the entry and about the semantics.

The RANKINGS slot, in this case, is used to order certain modifiers in the generated text about this object. The view is that the slots in a unit define, to some extent, what is interesting about that object, and the higher the ranking, the more important that aspect is. Therefore, when being used as a source of adjectives, for instance, the highest ranked slot is closest to the noun in question.

3.5 Pattern Matching on Units

Unless the units are attached via a workspace entry in some procedural component of a unit, there must be some good expressive way of finding out the relationships between units and slots. In many aspects this unit structure is like a semantic net with name methodology rather than pointer methodology.

A pattern matcher is available for expressing a search for an existing network of relationships between slots and units. In the above example, suppose that one wanted to get a hold of the RANKINGS slot given the name of the original base unit. In particular, suppose that the measure for the LENGTH is desired. The appropriate call would be:

```
(netmatch '(dta-str1 meta ?meta)
          '(?meta meta ?meta2)
          '(?meta2 rankings
                   (pat (* (length . ?amount) *))))
```

This states that the chain indicated should be followed until the RANKINGS slot is found, in which case the secondary tree structure pattern involving the length can be found. The match-variables ?meta, ?meta2, and ?amount are bound upon success.

Both constant and variable data can be used in the match specification, using either LISP atoms or match-variables. All three entries, the unit name, the slot name, and the slot value, can be match variables, except that the first unit name as a previously, temporarily bound variable cannot be used. This means that if the user wants to search the data base of units, this must be done explicitly, whereas, the slots in a unit can be searched for an entry satisfying a general constraint.

Except for the meanings of ?-variables and *-variables, the semantics of the match-variables are exactly like that in the general pattern matcher that is used extensively in the system and is described in Chapter 5.

A ?-variable in a value position in the match specification means to either obtain the atomic value of the slot or the first value in a list of values. A *-value means to obtain the exact entry, atom or list.

The sequence: t (pat (* (length . ?amount) *)) means that the entry in this slot must match the pattern given, where the general pattern matcher (called UMATCH) is used.

If one did not know, then, the name of the slot containing the rankings, one could write:

```
(netmatch '(dta-str1 meta ?meta)
          '(?meta meta ?meta2)
          '(?meta2
            ? (pat (* (length . ?amount) *))))
```

3.6 Representation and Description

This completes the overview of the representation scheme used in Yh. In Chapter 5 the *descriptive* system will be discussed, which is the basis for some of the less structured sequencing and matching that occurs in Yh.

Chapter 4

The Text

This chapter discusses the underlying data structure or representation of text as well a some of the representations of the other interesting linguistic entities used by Yh. The basic structure is called the *stratified surface structure*, which is actually a cross product of a standard tree structure and a string-like structure.[1]

4.1 Stratified Surface Structure

Yh uses an internal representation of the text which nearly equally allows string-like and tree-like operations to be performed on it. This representation proves to be a good vehicle for the transformations that the system applies to pieces of text.

4.1.1 Motivation

Nearly all of the representation schemes for English outside of those used in text editors currently in use have been designed with the problem of parsing in mind. In parsing the problem is to take some object with a structure that is encoded linearly and to produce a second object with that structure explicit.

A major part of that structure is involved in assigning common contexts to objects playing certain roles. That is, there are usually a number of phrases and words which can perform certain functions based solely on their form, such as noun phrases, which can be agents, objects, or modifiers. Without any context there is often difficulty determining these ultimate roles. Once the roles are determined, a natural representation would be a simple tree structure, so that by observing the parental links, these roles become explicit.

The problem with this is that regaining the adjacency information that is very apparent in the original sentences is difficult. That is, exactly which words are next to which others can only be derived by a tree-walk with counters or markers.

However, since the process of parsing is that of assigning a structure to an object from which the meaning of the object can be derived, a great deal of detail of that structure must be made explicit in order to retrieve the facets of the structure. Whereas, if the meaning is also available, then the amount of detail can be reduced.

For example, in the sentence, *They are eating apples*, it can be made clear that the role of *eating* is one of a verbal form acting as a modifier in a traditional tree-structure representation by its having an ancestor in the tree which is identical to that of *apples*. Alternatively, if each word has a pointer to a semantic unit—one which 'represents' the meaning of the sentence in some standard form—then the information about the role

[1] Actually, the name, stratified surface structure, is misleading in that the actual data structure retains all of the traditional tree structure that we're used to in phrase structure grammars. The difference is mainly that of emphasis on the non-tree structure parts of it.

a word or phrase plays can be gathered with some retrievals or with some pattern match, for instance, on that unit. So, in this example, one could have *eating* point to part of a unit representing the event of eating used as an adjective attached to the unit for *apples*.

Thus, as in any representation, it is a matter of the *distinctions* that need to be made that decides whether a representation is adequate. In this case, many of the things that are necessary to be able to distinguish between come from either the deep semantic base (do phrases represent the *same* thing) or from the actual wording used (will a particular surface structure confuse the reader).

Since the semantic base is available and not hidden, the facts about the meaning of the utterance are directly open to inspection, not the product of deduction on a clever representation of possible parsings.

4.1.2 Initial Remarks and Disclaimer

Now that the stage is set, how is the text represented?

The data structure is a constant depth tree (currently the depth is 5), with the nodes at each level available as strings. Thus, it is trivial to access the next node to the right of one at that same level. Embedded clauses are represented as pointers to other such structures.

Here is an example of how the simple sentence, 'I like iguanas' is represented:

SENT		
SUBJ	PRED	
NP	VP	NP
NOUN	VERB	NOUN
I	like	iguanas

What is missing in this particular example is the fact that each node in the tree also points to a unit that contains a full description of what that node contains. The SENT node (topmost) contains a pointer to a unit that states that the sentence is a simple declarative, has an animate subject and direct object, and also contains a copy of the *situation description* at the time the sentence was created initially. In the next few chapters the exact role of the situation description will be given, but suffice it to say that it represents a description of what the system was attempting to do when this sentence was generated.

Therefore, even though the data structure above doesn't appear to have enough 'languagey' information—not all of the full range of linguistic categories for words and phrases are used—that information is really kept behind the scenes in a form that is more like that which is used throughout the rest of the system.

Moreover, functional entries (such as SUBJ and PRED) appear on some levels while categorical entries (such as NP) are on others. The second level is the exception to the categorical rule, always containing either SUBJ or PRED. This text representation is described for completeness and in order to give some feeling for how the system operates. The linguistic knowledge in Yh is quite primitive, and the point is that such a primitive knowledge base coupled with a strong organization can result in reasonable linguistic behavior.

Each node can either be an atom (symbol) or a list of symbols which describe some surface feature of the entry. So if a noun phrase modifies another noun phrase in the style of an appositive, the level may look like:

NP	(NP APPOSITIVE)

In deciding, when two noun phrases are adjacent, which modifies which, the additional marker (here, *appositive*) in the list tells the relevant story.

4.2 Features of the Representation

This section will deal with some of the things that can be done with the representation. In this section and in the chapter with the main example of the thesis the adequacy and usefulness of this representation will be shown. This section is also quite detailed and can be skipped without much problem.

In general, the data structure consists of *nodes* with four pointers: a back pointer, a forward pointer, an up pointer, and a downlist of pointers. These allow the movement up and down the tree, as well as left and right along levels. There are two other pointers, the pointer to the head of the structure (the leftmost nodes in the tree), and the pointer to the *current location*.

One thing to note is that, except that relative and embedded clauses have one extra branch to follow, which is invisible, this representation is exactly a tree structure with added pointers upwards to parent nodes and sideways to sibling nodes (nodes at the same depth).

4.2.1 Here

The data structure contains a pointer to the current location, which is a *slice* through the structure. It is a pointer to a set of nodes such that one is on the lowest, another is its parent, and so on until the uppermost level is reached. This is an example

<table>
<tr><td colspan="4">here</td></tr>
<tr><td>SUBJ</td><td colspan="3">here</td></tr>
<tr><td>NP</td><td>here</td><td colspan="2">NP</td></tr>
<tr><td>NOUN</td><td>here</td><td>DET</td><td>NOUN</td></tr>
<tr><td>this</td><td>here</td><td>an</td><td>example</td></tr>
</table>

where *here* represents such a location pointer.

4.2.2 Creating

There are several ways to create such a data structure. In the system, the data structure is called a *tower*.

The main way is to create a tower from a standard LISP representation of the tree (*create-tower-tree, create multi-tower-tree*). For example, the sentence above, "I like iguanas" came from the tree:

```
(sent (subj (np (noun (I))))
      (pred (vp (verb (like)))
            (np (noun (iguanas)))))
```

The second standard way is to create an empty tower (*create-tower*) and to then add things to the right (*add-right*) or to the left (*add-left*). When adding things in either direction one specifies a *partial slice* of the tower, which is added to the bottom of the tower with missing upper levels of the tower shared in the appropriate direction. For example, from the tower:

<table>
<tr><td colspan="2">SENT</td></tr>
<tr><td>SUBJ</td><td>PRED</td></tr>
<tr><td>NP</td><td>VP</td></tr>
<tr><td>NOUN</td><td>VERB</td></tr>
<tr><td>I</td><td>like</td></tr>
</table>

adding `(np noun iguanas)` to the right of the last slice (assuming the pointer is positioned there) yields the above full sentence.

4.2.3 Moving

One can move the current position pointer to the right or to the left (*move-right, move-left*) on any level. That is, the levels, viewed as strings, are available for linear scanning. In addition, one can move all the way to the left or right on a tower, all the way left or right on a level, or to any entry name on a level (*move-full-left, move-full-right, move-full-right-level, move-full-left-level, move-to-real-entry*). Also, given a location pointer, one can move to that position (*set-position*).

4.2.4 Deleting

An entry on any level at the current position can be deleted (*delete*).

4.2.5 Appending and Merging

Two towers can be appended (*append*) or two adjacent entries can have their daughters merged, with the new entry being either one of the adjacent ones (*merge*). Additional towers, in the form of trees, can be added to the right or the left (*add-tree-right, add-tree-left, add-multi-tree-right, add-multi-tree-left*).

4.2.6 Setting and Getting Entries

All entries and pointers to semantic units (which contain the semantics of the entry in a standard form) are available for inspection and changing (*get-entry, get-semantics, set-entry, set-semantics*).

4.2.7 Getting and Setting the Current Position

The current position can be retrieved (copied) and set (*get-position, set-position*).

4.2.8 Matching

The most important function is to be able to pattern match on a tower and to modify or create towers using variables which appear in the pattern that is successfully matched. This is the basis of the transformational system that will discussed shortly.

The pattern matcher is a backtracking tree matcher with fragments (*-variables). Special syntax exists to allow the user to specify the pattern in terms of mainly row patterns (patterns on a single level in the tree). This matcher is very similar to the Conniver matcher [Sussman 1972].

The motivation for using a pattern matcher with this representation is the same as the motivation for using a pattern matcher in any situation, which is that a pattern matcher allows one to use a descriptive language that refers to complex objects via context and with a fair amount of brevity. The style of transformations done are generally of the form: Antecedent $\rightarrow$ Consequent, where both sides are patterns.

The pattern matcher (*tower-match*) takes a pattern in a non-tower form and a tower, and then it decides if the pattern matches the tower. The pattern is in two parts, the *pure-pattern* and the *restriction-pattern*.

The pure pattern is much like a pattern in most pattern match languages. First, it is a list of patterns, one for each row. In the current system, there are five rows, and so the pure-patterns have 5 elements in the list. Each element is a list of pattern elements for that row. A pattern element can be an atom, in which the pattern element and the corresponding tower entry must be EQ. The pattern element can be a ?-variable (of the form ?<var>), in which case the only restriction on that variable is that all bindings of that variable to tower entries must be consistent. A pattern variable can also be ?, in which case it matches anything.

A pattern variable can be also be a *-variable, in which case it can match any number (even 0) of tower entries, as long as the binding is consistent, as in the ?-variable case. A pattern variable of * matches any number of entries, but does not do any binding.

In a match, the first consistent match is the answer—though one can obtain the next consistent one with a special call to the matcher.

Arbitrary restrictions can be applied to ?-variables and *-variables, in the latter case, either to the entire list of (potentially) matching elements or to each element as it is selected.

The exact meaning of these variables and restrictions, as well as their syntax, is identical to that in the two-way matcher, UMATCH, which is discussed in Chapter 6.

The pure-pattern

```
((sent1)
 (subj pred)
 ((restrict ? npp) vp np)
 (noun verb noun)
 (? ? ?))
```

matches the tower:

SENT		
SUBJ	PRED	
NP	VP	NP
NOUN	VERB	NOUN
Raoul	loves	iguanas

However, pure-patterns are not enough to fully specify a match since each row is treated as a string independently, which means that the tree-like nature of the representation is ignored. To fix this, one can specify the location(s) of a pattern element(s) with a WITHIN clause, which can appear either in the pure-pattern part of the pattern or in the restriction-pattern part—the normal place for them. The restriction-pattern part of the pattern is where one specifies additional constraints on the tree structure of the stratified surface structure in the form of WITHIN clauses. In other words, the matcher is explicitly a tree matcher with the WITHIN clauses specifying the tree structure.

A WITHIN clause is of the form `(within <pat-var> <pattern>)`, for example `(within pred (vp np))`, which states that in the above match example, the pattern, `(vp np)` must be within the `pred` part of the tower. This captures the fact that as a tree, `pred` is the parent node of `vp` and `np`. The locations of the various elements in the pure-pattern are kept on a (global) alist, even constant data, so that the locations can be derived easily. Of course, then, the sample WITHIN clause above is wrong (!) since the variable `np` appears twice, and there isn't any well-defined way to distinguish them. Here is the real pure-pattern for the above tower:

```
(((sent1)
  ((within sent1 (subj pred)))
  ((within subj ((restrict ?np1 npp)))
   (within pred ((restrict ?vp vpp)
                 (restrict ?np2 npp))))
  ((within ?np1 ((restrict ?noun1 nounp)))
   (within ?vp  ((restrict ?verb verbp)))
   (within ?np2 ((restrict ?noun2 nounp))))
  ((within ?noun1 ?word1)
   (within ?verb ?word2)
   (within ?noun2 ?word3))))
```

which is much more involved than the naïve first attempt.

The restriction-pattern can also contain arbitrary predicates that must be true; also there are built-in routines that access the internal alist so that the values of the variables in the match can examined. A full pattern is of the form: `(pure-pattern restriction-pattern)`, but often the restriction-pattern is empty. An alternative way of writing the above full pattern is:

```
(((sent1)
  (subj pred)
  ((restrict ?np1 npp)(restrict ?vp vpp)
   (restrict ?np2 npp))
  ((restrict ?noun1 nounp)
   (restrict ?verb verbp)
   (restrict ?noun2 nounp))
 (?word1 ?word2 ?word3))
 ((within sent1 (subj pred))
  (within subj ?np1)
  (within pred (?vp ?np2 npp))
  (within ?np1 ?noun1)
  (within ?vp  ?verb)
  (within ?np2 ?noun2)
  (within ?noun1 ?word1)
  (within ?verb ?word2)
  (within ?noun2 ?word3)))
```

4.2.9 Building from Matches

Once it is possible to pattern match nicely, the next natural thing to do is to build towers from the variables that have been so carefully set up by the matcher. This is also desirable since such an activity can form the basis of a transformational system on top of this representation.

The idea is that one can use a pattern almost exactly like that which the matcher does, except that it is instantiated rather than used in a match. The only difference between these patterns and the match style pattern is that the pure-pattern part shouldn't have any RESTRICTION clauses—just the variables that would have been in them—and the restriction-pattern cannot contain any predicates.

4.2.10 Replacement

A more interesting ability is to be able to replace part of a tower with something else (*tower-replace*). There are several options with this routine, which is used extensively by the transformations in Yh. Two towers

are supplied to the routine plus a specification of where the replacement takes place.

The result of a match in which either ?- or *-variables are involved is that these variables are bound to what they match. More than that, the location in the tower is associated with the variable as well, so that such a variable can be used to specify locations in the tower.

If a ?- or a *-variable is given to *tower-replace*, then that entry (or all of the entries) and all daughters are replaced by the tower provided. If it is a number, it is taken as a row number for the current position, which is then replaced. If it is a dotted pair of numbers, $(n_1 \quad . \quad n_2)$ then n_1 is the row number as above and n_2 is the number of elements starting at that position to replace, à la *-variables.

4.2.11 More Match-based Operations

There are several other operations that can be performed on towers based on the pattern matcher as well as some further complications on the basic matcher that are used to increase the expressive power of the patterns.

The first of these takes a ?- or a *-variable, a pattern, and a tower performs the match, and then positions the current position over the location (or the first location) that matched the variable (*match-position*).

Patterns can also be used to match and position on individual levels as in the full tower version (*level-match, level-match-position*). A matcher that can match the leafs of the tower as if it were a sentence—takes the relative clause escapes into account—is useful for making some simple word changes (*string-match, string-transform*). The string transformer only works for strings of equal length (word substitutions) and takes two patterns, a match pattern and an instantiation pattern.

Levels can be transformed also (*level-transform*) in which a pattern for a row and a transformed pattern for that row is supplied. The routine then constructs a full tower pattern from the row patterns, matches, and then builds the new tower.

The main matcher is able to match only parts of the full tower without needing to even trivially consider the extraneous parts. This is a useful feature since an entire paragraph is a single tower, and that can mean many sentences and entries. First, a pattern does not need to include every row, but if n rows are supplied, then that refers to the bottom n rows. Additionally, if an alist (left over from another match or constructed out of thin air) is supplied, then the WITHIN construction can greatly reduce the area considered in the match. Finally, there is the `(here <var>)` construction, which makes use of the current position in the tower. If some form appears such as `(here <var>)`, then that is taken to mean the current location on the level specified (specification through implication from the number of row patterns supplied), and the variable `<var>` will then be used to refer to this spot.

4.2.12 Listifying Things

Often it is useful to turn various parts of a tower into a simple list structure for the purpose of reasoning about it or finding out if any units in the system can perform special functions on that structure.

Basically, any level can be listified (*listify-level*) as well as any level starting at the current position (*listify-level-from-here*). The level below (with nodes with more than one daughter appearing as sublists) the one specified can be listified either entirely or starting from the current position (*listify-level-below, listify-level-below-from-here*).

4.2.13 Mapping

Any function can be applied to every element on a level, starting at the current position, and proceeding in either direction, stopping at the ends or on any predicate, and returning any value. That is, this is a basic full MAP ((*MAP tower level direction function stop-predicate return-function*)).

4.3 Relative Clauses

Relative clauses, or other clauses which are at a depth greater than that allowed, are represented by pointers to other stratified surface structures. The idea behind this is that relative clauses can usually be treated on the whole as a general qualifier (something that is added later as mandatory or optional information about the object it qualifies), and it is almost never the case that the pure string parts of sentences have major interactions over subclause boundaries. Furthermore, the existence of relative clauses is viewed as a result of higher level stratifications making large modifications to existing simple sentences.

4.4 Dire Implementation

The implementation is very complex, involving forward pointers, back pointers, up-pointers, down-pointers, header-pointers, position pointers, and HUNKS for the various pieces for efficiency. A HUNK is a MacLisp specific data structure which is a short list. For lists of length 4 to length 8 or 10, the expense of CDRing down to the average point is intolerable, and the overhead of using arrays (with garbage collectable headers of the same size as the data cells) of this size is often prohibitive. Thus, the HUNK is a short, power of 2 sized object that lives in a particular space (depending on the power of 2) in MacLisp, which is indexed as arrays, but since the sizes are known, the array headers are trivial.

There are special routines to print out towers and sentences (*print-tower, print-sentence*) since the structure is circular and unprintable with standard routines.[2]

4.5 Transformations

One of the most interesting parts of the linguistic component of Yh is the transformation system.

That is, given that deliberate writing is being modelled, it is natural to assume that changes to existing text must be made. An obvious way to represent these modifications is through transformations, although it may be advantageous to do some other kind of processing to determine whether a transformation is desirable or possible. This processing is likely to be semantic since it may be the case that the text is initially satisfactory and that some other purposes are being served by performing these changes.

It is nowhere assumed that simple sentences—only those generated from a context-free grammar—are initially produced with only transformations able to turn these into fuller sentences. In fact, fairly complex and generally overly complex sentences are produced first and the transformations are a simplifying influence.

The reason that complex sentences are produced is that the various generators that are in the system, mainly noun phrase generators, are careful to produce very exact, full noun phrases that are as unambiguous

[2] Unfortunately, the usefulness of this data structure, which is quite high when it works, is depreciated by the fact that once a bug in a pattern or in some instantiation specification occurs it can be very difficult to find due to the essential difficulty of printing the structure without special routines. Several routines to help verify the correctness of a tower structure have been written, but every eventuality is hard to anticipate. I'm not sure that this implementation is satisfactory, but the ability to scan levels of the tower have been very useful for many of the reading-like operations that Yh does.

as possible. Hence, it is often the case that there is a lot of repetition among noun phrases for the same or related things, and the transformations are used to move some of these repeated items around.

4.5.1 Transformations in Detail

A transformation is simply a unit (actually a shallow stratification) with a transformation part in the base-unit and a description part in the meta-unit. The description points out some of the interesting features of the transformation, such as what it does, what it assumes to be true at the beginning, and what it makes true at the end. Additionally, it tells why the transformation might be made in the first place. Some decrease the wordiness of a sentence, others collapse sentences, thereby destroying some parallel constructions, others make certain parts of it more emphatic, etc.

As with any other stratification (*stratification* with no adjective implies *shallow stratification*), when it is invoked the METHOD is evaluated, and, in the case of transformations, the method part of the meta-transformation applies the transformation. The transformation is a match pattern and an instantiation pattern as described above. Almost all transformations are applied in place (rather than copying the entire tower), and there is also specified in the transformation the match variable that is changed. These slots in the transformation unit are ANTE-TRANSFORM, CONSEQ-TRANSFORM, and TRANSFORM-VARIABLE, resp.

The following is an example of a tranformation that changes an occurrence of "iguana" into "green reptile."

```
{iguana-to-green-reptile
 ante-transform
 ((
   ((here '?sentence) *)              ;Row1: The current sentence
   ((within ?sentence (*types)))      ;Row2
   ((within *types (*ph1 np *ph2)))   ;Row3: Finds the noun phrase
   ((within *ph1 (*c))                ;Row4
    (within np (*d noun *e))
    (within *ph2 (*f)))
   ((within *c (*g))(within *d (*h)) ;Row5
    (within noun (iguana))            ;that contains `iguana'
    (within *e (*i))(within *f (*j)))))

 transform-variable noun
 conseq-transform
 ((
   (adj noun)        ;Row4
   (green reptile)) ;Row5
  (within adj (green)) ;Tree structuring instructions
  (within noun (reptile)))}
```

Notice that only those things that must be changed are actually changed, since the only relation between ANTE-TRANSFORM and CONSEQ-TRANSFORM is that the former is used to set variables and positions in the original tower while the second, along with TRANSFORM-VARIABLE modify the tower where needed. The routine that does the modification is destructive.

4.5.2 Problems

However, from what has been said so far, certain types of useful transformations are difficult or impossible. For instance, suppose that one uses *-variables to mean 'whatever is there.' Then there is no good way to

specify what is within what else. So if a pattern looks like:

```
(...(...(within *x (*y))...)...)
```

later saying the same WITHIN clause in the instantiation pattern may not result in the same tower fragment, since the building routine can only guess where things go (and it does). In short, what can happen is that a writer of transformations will use some *-variable to mean to assemble whatever entries are at some level for some portion of the level, and this writer may only specify that a second set of *-variables are to with "within" these. Later, when a structure is being built up out of these variables, the person may want the same relationships to hold between the *-variables that held in the original tower. The current WITHIN mechanism does not do that.

The solution is to use the OWITHIN clause in place of the WITHIN clause in the instantiation pattern, which causes the building routine to look at the old tower at that point to see where things went, and tries to put them in the right place.

For instance, given the (odd) tower (written as several towers; imagine they are simply appended together):

SENT		
SUBJ		
NP		
DET	ADJ	NOUN
The	odd	boy

SENT	
SUBJ	
NP2	
DET	NOUN
a	nut

SENT	
PRED	
VP	NP
VERB	NOUN
likes	iguanas

one can match it to the pattern:

```
((
  (?sent)
  ((within ?sent (subj pred))) ;subj and pred below that
  ((within subj (*nps))        ;some noun phrases in the subj
   (within pred (vp np)))
  ((within *nps (*classes))    ;some classes there
   (within vp (verb))(within np (noun)))
  ((within *classes (*words))  ;some words there
   (within verb (likes))(within noun (iguanas))))))
```

successfully, and then produce the same tower with the instantiation pattern:

```
((
  (?sent)              ;the rows of the new tower
  (subj pred)
  (*nps vp np)
  (*classes verb noun)
  (*words likes iguanas))

 (within ?sent (subj pred))
 (within subj (*nps))
 (within pred (vp np))
 (owithin *nps (*classes))  ;same classes in the same nps
 (owithin *classes (*words));same words in those classes
 (within vp (verb))
 (within verb (likes))
 (within np (noun))
 (within noun (iguanas))))
```

In addition, there may be some further processing that needs to be done between the match and the instantiation. Consider the passive transformation, which takes an active voice sentence and makes it passive voice. After the match has taken place and the verb phrase is available for inspection, the full power of the system may need to be applied to the task of finding out how to make that verb phrase passive in the manner desired. Perhaps it isn't as dramatic as that, but certainly the form of the passive verb phrase needs to be computed.

To do this there are two other fixed slots in the base-unit of the transformation, INTERNAL-METHOD and INTERNAL-BINDINGS. The meaning of the INTERNAL-BINDINGS slot is that it contains the names of pattern variables that will be made available to INTERNAL-METHOD, which can then set those variables before the instantiator looks at them.

In the specific case of the passive transformation, the internal method finds the unit that represents the main verb in the verb phrase and essentially reads[3] the unit in order to find out what form it should take. This makes the passive transformation unit an observer of the lexicon units—this is what the dictionary is made up of—and the usefulness, then, of the lexicon is a function of the distinctions that the observers, such as the passive transformation, can make in/on/with those units.

When the internal method finishes, the instantiation portion of the transformation continues, unless the global variable, `-abort-transformation-`, has been set, which means that the internal method has discovered that the transformation is not going to work the way it was planned.

4.5.3 More Problems

This still does not cover the full range of things that need to be done. To depend on the matcher and instantiator to be able to perform every function needed is to set a limit on the power of a transformation that may be linguistically unjustified or computationally unwarranted.

For example, consider the case of transformations that cover more than one sentence. Such a transformation is the one that collapses two adjacent sentences with the same subject. This transformation involves

[3]The way this is done is to have the searching matcher, discussed in Chapters 5, 6, and 7, find the right unit, which then is searched with the unit structure pattern matcher [Chapter 3] for the parts that know about the right word forms. This can only work because the units that make up the dictionary are in a fairly standard format, although not entirely fixed. However, the knowledge of the verb forms is always found in a patterned entry. The correct way is to use an intermediary unit that is *meta* to the lexicon entries for verbs.

erasing one of the sentences once the transformation has been made. In this case the first sentence is transformed to include the second, and then the second is deleted. The procedural part of the meta-unit is called upon to apply the normal match-based part of the transformation and then the special deletion is done.

Furthermore, since there is considerable power associated with being able to scan left and right through the text, some transformations have no match-based parts at all.

4.5.4 Range of Transformations

Currently there are transformations for all of the cohesion techniques explained in Chapter 2, and they work basically as described there. That means that there are approximately 20 such transformations. At the moment the task of writing these transformations is somewhat difficult and no one but the author has ever written one.

4.6 A Big Example Transformation

The following is a transformation that takes two sentence with the same agent and action and collapses them if they are not far apart in terms of intermediate sentences.

```
{decl-decl-same-agent-same-vp-collapse
 meta meta-decl-decl-same-agent-same-vp-collapse
 level 2
 internal-bindings (*sentences *g1 ?rp1 ?rp2)
 internal-method
 (cond
  ((> (length *sentences) 3)
   ;;; quit if more than 3 sentences between them
   (setq -abort-transformation- t)
   (format msgfiles
    '|S1 and S2 are too far apart;
      aborting transformation.|
    ?left-sentence ?right-sentence))
  (t
   (cond ((= 0                ;if adjacent
           (length *sentences))
          (setq ?rp1 ?rp2)))  ;the punctuation is the
                              ;punctuation of the right
                              ;sentence; otherwise the left.
   (if-success                ;if the OPTIMAL-CALL-GOALS
    (setq *g1                 ;succeeds (a plural is found)
     (optimal-call-goals
      `(,*g1
        (phrase-knowledge)
        (make plural tense for verb given string))
      pairs))
    t                         ;then quit happily
    (setq                     ;otherwise quit
     -abort-transformation- t))))
```

```
ante-transform
((
  ;;; First find the left and right sentences
  ;;; (passed as args)
  (* <∀(x)(eq x ?left-sentence) ?sentence1>
     *sentences  ;as well as the intermediate ones
     <∀(x)(eq x ?right-sentence) ?sentence2> *)
  ((within ?sentence1 (<subjp ?subj1>
   ;;; now flesh things out
   ;;; <pred var> is a restriction
   ;;; of var in the match language
                       <predp ?pred1><cstopp ?cstop1>))
   (within *sentences (*clauses))
   (within ?sentence2 (<subjp ?subj2>
                       <predp ?pred2><cstopp ?cstop2>)))
  ((within ?subj1 (<npp ?np1>))
   (within ?pred1 (<vpp ?vp1> <objpp ?objp1>))
   (within ?cstop1 (<pstopp ?pstop1>))
   (within *clauses (*phrases))
   (within ?subj2 (<npp ?np2>))
   (within ?pred2 (<vpp ?vp2> <objpp ?objp2>))
   (within ?cstop2 (<pstopp ?pstop2>)))

  ((within ?np1 (*c1))
   (within ?vp1 (*d1))
   (within ?objp1 (*e1))
   (within ?pstop1 (?punc1))
   (within *phrases (*classes))
   (within ?np2 (*c2))
   (within ?vp2 (*d2))
   (within ?objp2 (*e2))
   (within ?pstop2 (?punc2)))
  ((within *c1 (*f1))
   (within *d1 (*g1))
   (within *e1 (*h1))
   (within ?punc1 (?rp1))
   (within *classes (*words))
   (within *c2 (*f2))
   (within *d2 (*g2))
   (within *e2 (*h2))
   (within ?punc2 (?rp2)))))
```

```
 transform-variable ?sentence1
 conseq-transform
 ((
   (?sentence1)
   (?subj1 ?pred1 ?cstop1)
   (?np1 pconj ?np2 ?vp1 ?objp1 ?pstop1)
   (*c1 conj *c2 *d1 *e1 ?punc1)
   (*f1 and *f2 *g1  *h1 ?rp1))
  (within ?sentence1 (?subj1 ?pred1 ?cstop1))
  (within ?subj1 (?np1 pconj ?np2))
  (within ?pred1 (?vp1 ?objp1))
  (within ?cstop1 (?pstop1))
  (within ?np1 (*c1))
  (within pconj (conj))(within ?np2 (*c2))
  (within ?vp1 (*d1))(within ?objp1 (*e1))
  (within ?pstop1 (?punc1))
  (within *c1 (*f1))(within *c2 (*f2))
  (within conj (and))(within *d1 (*g1))
  (within *e1 (*h1))
  (within ?punc1 (?rp1)))}

{meta-decl-decl-same-agent-same-vp-collapse
  level 3
  purpose  ((merge sentence ? with ?
             based on same-vp and same-objp))

  goals (((merge sentence ?left-sentence
           with ?right-sentence
           based on same-vp and same-objp) . 600))

  bindings (?left-sentence ?right-sentence)
  constraints ((boundp t
                (and (m-boundp '?left-sentence)
                     (m-boundp '?right-sentence))))
  β decl-decl-same-agent-same-vp-collapse
  global-workspaces '(* output-buffer attention)
  method
  (lambda (pairs)
   (set-bindings-from-pairs pairs)
```

```
;;; We must see if both sentences exist!
(cond
 ((and
   (sentence-exists ?left-sentence)
   (sentence-exists ?right-sentence))

   ;;; Apply the transformation, move right,
   ;;; delete that sentence, move back,
   ;;; and set the current-sentence to
   ;;; be correct. Note that we cannot
   ;;; replace the tower in place
   ;;; and delete in 1 fell swoop,
   ;;; hence the malarky.
   (let sent u1 u2 ← () () ()
    do
    (cond
     ((apply-transformation-in-place β)
      (move-to-sentence ?right-sentence)
      (aset' u2 (get-semantics output-buffer
                 -sentence-level-))
      (aset' sent (get-entry output-buffer
                   -sentence-level-))
      (delete output-buffer -sentence-level- 'left)
      (move-to-sentence ?left-sentence)
      (setq current-sentence
       (get-entry output-buffer -sentence-level-))
      (aset' u1
       (get-semantics output-buffer -sentence-level-))
      (merge-semantic-units u1 u2)
      (format msgfiles
       '|Collapsing A1 and A2 due to
         same subject and verb phrase.|
       ?left-sentence ?right-sentence))))))
(return t))}
```

The first thing to notice is that the description of what this transformation does is very uninteresting, although it does do a fairly complex operation.

It takes two sentences, ?sentence1 and ?sentence2, and finds them in the current paragraph, which is what these lines in the ANTE-TRANFORM do:

```
(* <∀(x)(eq x ?left-sentence) ?sentence1> *sentences
   <∀(x)(eq x ?right-sentence) ?sentence2> *)
```

If there are more than three sentences between the two, the transformation is aborted. Otherwise, the plural of the verb phrase is figured out (failure here aborts the operation also) and the first sentence is mingled with the second. The second sentence is deleted and its semantic contents are merged with that of the first sentence.

The idea is to take a sentence pair like:

The array contains numbers...The list contains numbers.

and turn them into:

The array and the list contain numbers.

4.7 The Lexicon

To a large degree the lexicon represents the spirit of the system more than most other parts. This is because in word choice there is a fair amount of choice available, and these choices depend on many things besides the "intended" meaning of the word to be selected.

The lexicon contains the richest descriptions in terms of the internal description language. The richer the description attached to a unit the better a judgment the system can make. The transformations have fairly straightforward descriptions, as do the sentence schema. In general, the more structured the task, the poorer the descriptions are of the methods for performing that task.

In word choice, one is often able to subtly change the tenor of a conversation or to make other suggestions and hint at hidden meanings. With the choices available a clever speaker can hold several conversations at once with a variety of people while still maintaining a serious, intentional dialogue with one of the listeners.

This matter of choice is an example of collocation as mentioned in Chapter 2, in which the words chosen follow a similar vein, often adding to some overall impression of the meaning of the text rather than pointing exactly at it as the straightforward syntactic part does.

Poets often use this technique, and in some ways, poetry would be a fine place to do some work on the creativity in writing. For in poetry the shackles of syntax and grammar are loosened a bit.

InkWell

Starting in 2013 I did a pile of work in this area—a system called "InkWell." Check my website, `Dreamsongs.com`, for some papers about that work.

In "A Grief Ago" by Dylan Thomas [Thomas 1938], one finds the phrase 'boxed into love.' By plan ambiguous, with either meaning 'boxed' as in 'put in a coffin' or as in 'punched,' one can imagine that the concurrent themes of fighting to survive and giving in to death run throughout a piece of writing, and when faced with the choice of how to say 'pushed into love' or 'coerced into love,' it is necessary to have an index into the word 'boxed' as a synonym for 'coerce,' which is an unlikely synonym a priori, but highly likely given the other two nuances floating around.

The lexicon, then, is just another set of stratifications, with a descriptive and a procedural part. The procedural part simply puts the word(s) that are selected into the right place (normally the current position). The description part uses the extensive descriptive facility to be discussed in Chapters 5 and 6.

Therefore, there is encouragment by the system design to have very extensive and comprehensive descriptions of each word and the full context that is necessary for the alternative meanings. Multiple meta-units can be used to separate concerns, although this may not be necessary. Consider the example of the word, sleep. Not only does this word have the strict meaning of *sleep*, but it, and its adjectival form, sleepy, has connotations such as, dead, inattentive, unconcerned, unresponsive, drugged, unconscious, etc. In fact, the existence of thesauri lends credence that there is more to word choice than lookup and indexing, that judgment has a way of creeping into the picture.

These nuances can be captured with the descriptive system, with the word 'sleepy' having a possible description:

```
((sleepy . 950)
 (dead . 210)
 (unresponsive . 220)...)
```

and so on in the obvious way. The other influences and goals in the system at any time can then cause this word to be chosen over the more obvious ones.

4.7.1 Details of the Lexicon

Here is an example noun from the lexicon:

```
{program-word
 level 2
 goals ((program . 900.)
        (noun . 700)
        (verb . -700)
        ((number ?number) . 20))
 bindings (?number)
 pairing-function structured-full-pairs
 purpose (program)
 method (lambda (pairs (tower ()))
         (lexicon-single-word-noun-entry
          pairs tower 'program-word 'program 'programs)
         (return t))}
```

The description says that this can be used to express the concept of a program, and not 'to program.' The NUMBER entry means that one can specify `(number plural)` to get the plural of the word.

The call to LEXICON-SINGLE-WORD-NOUN-ENTRY is a speedup to compiled code. It figures out where to put the word and then performs the recent usage degradation operation, which puts a negative preference on the word that is decreased until a threshold is reached. The relaxing of the negative preference takes place over the course of the history of the system at fairly regular intervals (each agenda selection and each word selection). Thus, the normal course of events is that if a constant request for a word is made, the synonyms get cycled through in an order that depends on the recency of use and its other characteristics (see Chapter 6 on the exact meaning of *preference*).

Here is an example verb entry for initializing something:

```
{initial-value
 level 2
 goals ((initial-value . 910.)
        ((tense ?tense) . 300.))
 bindings (?tense)
 meta meta-initial-value
 purpose (initial-value)
 method (lambda (pairs (tower ()))
         (lexicon-multiple-word-verb-entry-with-templates
          pairs tower
          'initial-value
          '((is initialized to) (aux () prep))
          '((was initialized to) ((aux past) () prep))
          '((will be initialized to)
            ((aux future)(aux future) () prep))
          '((is being initialized to)
            ((aux progressive)(aux progressive)
             () prep)))
         (return t))}
```

The first part of the stratification, the base-unit, tells how build the verb phrase for 'to initialize to,' which is an idiomatic verb. This is similar to the noun example so far except that the entry, `(tense ?tense)`

means present, past, future, etc. The multiple entries in the call to LEXICON-MULTIPLE-WORD-VERB-ENTRY-WITH-TEMPLATES refers to the fact that the class level, as well as the word level, is set up by this routine. So, for the present tense, what is gotten is:

(VERB MULT AUX)	(VERB MULT)	(VERB MULT PREP)
is	initialized	to

The meta-unit looks like:

```
{meta-initial-value
 β initial-value
 level 3
 purpose ((phrase knowledge)
          (* initialized to *)
          (make plural tense for verb given string))
 pairing-function structured-full-pairs
 goals (((phrase knowledge ) . 500)
        ((make plural tense for verb given string) . 500)
        ((*pre initialized to *post)  . 500))
 constraints ((boundp t
               (and (m-boundp '*pre)
                    (m-boundp '*post))))
 method
 (lambda(pairs)
  (cond
   ((match '(*a is) *pre)
    (return
     (%instantiate '(*a are initialized to *post))))
   ((match '(*a was) *pre)
    (return (instantiate
             '(*a were initialized to *post))))
   ((match '(*a is being)
           *pre)
    (return (instantiate
             '(*a were being initialized to *post))))
   (t (return (instantiate
               '(*pre initialized to *post))))))}
```

This unit simply makes a plural out of a singular instance of this idiom. This type of meta unit takes care of so-called *phrase knowledge*, which means that it operates on the uninterpreted phrase in question, which is usually some idiom or other common phrase. This unit simply does some pattern matching on the phrase to return the correct plural.

4.8 Denouement

The lexicon, then, contains not just words indexed by their meanings, suitably represented, but words, phrases, and the basic building blocks of language in a network of interrelationships, multiple descriptions, and powerful phrase techniques that can turn one phrase into a similar or derived one. The lexicon is structured more like a thesaurus than a dictionary, and is able to demonstrate decaying usage behavior.

One can even include descriptions of other things which simply *remind* the system of the word or phrase, which will be retrieved if the current situation is similar to that one.

The text is not represented in a full parsed tree, but as a simplified stratified surface structure, which, by virtue of the semantics associated with each part of it, is able to make the distinctions that are necessary without recourse to standard detailed tree structures.

The structure of the text and the transformational nature of the system seem adequate to the task, and the operations that can be performed on the text are powerful enough that programming this particular machine is fairly straightforward and modular to the extent that adding knowledge requires only a minimal amount of information about the rest of the system, although an understanding of the design methodology of the rest of the system is necessary to be able to phrase the advertisement of the transformations and lexicon entries well.[4]

InkWell does what the footnote talks about better

Even though I had long forgotten about this footnote, the work I did on InkWell is mostly about word choice, subtext, moods, and lots of creative-writing type influences on writing choices.

[4] A regret: since 'completing' this research, I have come to believe that, although this approach allows for both top-down and bottom-up methods of expression to be used, the main creational act is word choice, where word choice is generalized to include idiomatic (to each speaker) phrases. The grammar, then, is used as 'glue' to piece together the parts so assembled, where needed. My feeling has always been that word choice should be an important contributor to text structure, but how strong a contributor, I fear I have underestimated. Regrettably, this system tends to be more top-down than this scenario.

Chapter 5

Planning

In any system one of the main concerns is to decide in which order things are to be done; in Yh there are several techniques for doing this, and these can be categorized along the dimension of refinement or the grain of the information examined.

In many AI systems involving planning, the general idea is to make a *plan* of action, based on a sequence of activities which, when performed lead to the goal at hand. There may be conditionals on these activities and alternatives available, but the entire problem and set of resources is examined in the planning process. This view is often characterized as locating a sequence of islands that can be reached one from the other with a single operation, which leads from the initial state to the desired state. This is an attractive metaphor and it will be used, but with some modifications.

The modifications arise, in the case of Yh, when one remembers that it is natural language generation that is going on. Here, the initial state is simply whatever state the generation program is in when the request for a generation is made, and the 'goal' state is the state in which the listener has understood what has been said, and what has been said is reasonably accurate. In the exact case of Yh, this is if the reader has understood the explanation of a simple program and the important points have been made in a fairly unambiguous way.[1]

5.1 Planning in Generation

Planning, in the AI literature generally, is determining a sequence of actions that will take an agent from some initial state to some final state. Both initial and final states are only loosely known in a generation problem. Planning in this domain means 1), examining the request for a generation to see what the points that must be stated are and 2), coming up with a sequence for saying these things.

In explaining programs, this plan is derived in a fairly straight-forward way from the program to be explained as follows. A program (in the limited case assumed here) is a set of data structures, some algorithm which is well laid out, and some miscellaneous routines to aid the algorithm. First, the data structures are explained in order of decreasing importance (assumed to be known from examining the program with the importances explicit—or implicit in the order they appear), then the algorithm is explained step-by-step in rough execution order (actually in lexical scan order), and finally the miscellaneous routines are explained.

This is a *knowledge-based* plan in that it depends on specific knowledge to the task of programming, and that knowledge is applied to fix the order of explaining the important points as well as locating those points.

[1] I don't think that the generation process can be specified much more than this, and we will always remain with words like 'important' and 'fairly unambiguous,' rather than with some easily pinned down specific goals.

How to say these points is a different task altogether, and one which cannot be planned at this stage in the operation. This is because there is the mission of saying things in a fairly unambiguous way, which can mean in a non-repetitive or non-puzzling way; this means using good style, which depends on what has been said and *how it has been said* up to the point of utterance. Moreover, this depends on the initial state, which is a very complex state, derived from the history of the generator. Therefore, the final planning must occur up to the context of *word choice*.

A further reason for this approach is that if a plan is made that calls for some set of things to have been said by a certain time, and if the planning does not go to word level, the situation could arise that some statements which render some planned reference unambiguous at that certain time may not have been said, because the words available were inadequate. In addition, it is desirable to make adjectival modifiers span various references to an object, such as:

The one dimensional array represents a stack. The array marker, which is the stack pointer, is initialized to 0; when this pointer becomes greater than 9 an error is signalled.

Here the details of the array have been introduced over a long period of time, giving the information that the array is one dimensional, zero-based, and has length ten. Of course this could have been said in one noun phrase, but the there may have been the admonition to be less wordy with noun phrases.

Again, the point that must be made is that it is not a high performance explanation program that is the research goal, but a program that can demonstrate a variety of linguistic behavior.

Therefore there is a very general planning level, in which the order of saying things is decided to a fairly coarse grain. The details of implementation are left until later. In the analogy of the islands, Yh identifies large, widely spaced islands, which are then locally planned. These islands are in the form of descriptions of the state of the generation, what is desired to happen on this island, what influences and soft-constraints apply, and, optionally, when to decide that the island has been completed. In addition, any failures are carried on to the next island. So if some goal is left unfulfilled, then that goal appears on the next island.

5.2 Agenda

The rough plan is represented on an *agenda*, which is an ordered list of things to do. As pointed out in Chapter 3, the agenda is a special unit in Yh. Here the exact structure of the agenda will be discussed.

As a unit, the agenda looks like:

```
{agenda
 level 1
 meta meta-agenda
 old-system-goals-name ()
 current-name ()
 current-form ()
 current-predicate ()
 current-priority 0
 current-highest-priority 0
 current-lowest-priority 0
 current-task ()
 in-use t
 items ()}
```

Each element in the list ITEMS is an entry on the agenda; it is of the form:

```
(...(name priority predicate type item)...)
```

where NAME is a pattern acting as an index into the agenda; PRIORITY is an integer, the higher the number, the higher the priority; PREDICATE is a predicate in the underlying language that is evaluated to decide if this item can be run at this time. TYPE is one of (PERFORM, GOAL-SELECT, SUICIDE-GOAL-SELECT).

If TYPE is PERFORM then ITEM is a form to be evaluated in the underlying language; if TYPE is one of the other two choices, then the *reactive* component is called on the value of ITEM, which is assumed to be a description of what to do (an island). If TYPE is SUICIDE-GOAL-SELECT, then the reactive component is called exactly once (PREDICATE is set to NIL). Otherwise the entry is liable to be called again.

Of special interest are the META and IN-USE slots, which are the mechanisms for the hierarchical agenda. The scheduler looks up the chain of meta-units each time the IN-USE slot of the meta-unit is T, and the highest level agenda with this property is used as the agenda. The idea is that the meta-agenda examines the base-agenda and decides what to. By using the IN-USE slot, Yh can control the agendas pretty well.

The use of the agenda in Yh, though, is to primarily be the vehicle for expressing the plan as outlined above, and thus most entries on the agenda are of the GOAL-SELECT variety, being a large island in the planning space. Normally, also, the items actually alternate between a GOAL-SELECT and a PERFORM where each PERFORM just adds the next island to the ongoing description, thus carrying over the unfinished work of each island.

Unlike some research current today, there is no great amount of expressive power claimed about the agenda mechanism here. That is, there is no attendant claim that many problems are solved by this technique, although the utility of representing the rough outline of the initial plan is a worthwhile feature. Although there is a stratified structure to the agenda, very little is done with it at the moment except to provide a means for garbage collecting the agenda. Presumably as more sophisticated things are asked of the generator, the meta-agenda will be used to observe which things are placed on the base-agenda and to detect interdependencies. This might stimulate rash claims about the existing methodology, but none are forthcoming at the moment.

5.3 Description Versus Representation

In Yh there are two distinct, but related, ways that correspondents to objects (physical and conceptual objects) are used and/or referred to. One is with *representations* and the other with *descriptions*. In Chapter 3 the representation scheme that Yh uses was discussed, and is based on units with slots and fillers, organized into meta-unit/base-unit towers with certain properties, able to support various types of inheritance, both implicitly and explicitly, and so on.

The objects in Yh that are instances of the representation—units—correspond to objects in the world or other objects in Yh. In building these objects it is hoped that there are operations in the system such that applying these operations on the objects corresponds to making inferences or observations about the world, and, in particular, about the objects in the world the representations[2] correspond to.

There is a second kind of thing that can be used along side of a representation system, which is a descriptive system. Here there are *descriptions*, which are very much like linguistic objects to the system. A description is a set of descriptors which can be reasoned about, compared, constructed, and taken apart. A description is an object which presents the features of a representation and their relative importance. It is a monolithic structure in order to be easy to manipulate by a matcher.

[2]I will use the word *representations* in the discussion in this section to refer to the objects built up with the tools of the representation system. So if there are some actual units in a system, then they are the representations. What they correspond to in the real world (or in the world of the system, if units correspond to those sorts of things) will be called the *objects.*

The representations in any system constitute the *world view* of the system. 'Thinking' about the world means manipulating these representations and nothing more; creating new representations is either expanding one's world view or becoming demented, depending on the reliability of the new representations in handling the world.

On the other hand, one can create and manipulate descriptions freely because they constitute the explicit beliefs about the nature of things. A description is a way of noting to the system that some belief is held, or some facets of a representation are being considered.

The description is used in Yh as a linguistic entity. Creating and manipulating descriptions is simply ruminating to itself much in the manner that people sometimes 'talk' to themselves. By allowing the free use of descriptions without requiring the system to make commitments about its long-term beliefs and world view, a great deal of freedom is possible.

The different uses to which the two are put determine some features of their structure: the representation is used for reasoning and thinking about the world, doing information retrievals, determining defaults when new objects are being created, and other structured activities. The description is used for passing around beliefs, matching, partial matching, changing relative importances, ignoring parts of a description, speculating about descriptions of things, having other influences alter the way the description is viewed, and other unstructured activities. Therefore, the representation is structured to accomodate structured actions, while the description is unstructured to facilitate unstructured actions.

For example, when a program is being described by Yh this is a structured part of the process in that the parts of the program that need to be discussed are explicitly there and in some order that is specified within the unit that represents the program. This structured representation results in the structured overall plan for discussing the program—talk about: the first array, the second array, the list, the main program, the subroutine, the first macro, and then the last macro. This is a representation driven process because the structure in the representation maps straightforwardly into the structure of the actions of the system (making the plan).

When the generation process moves to word and phrase choice, though, the safe grounds of a representation are left and the choice of action is left up to the descriptive system, which finds units (representations) that are appropriate within the abilities of the system to model the situation. The decision of what to do at the point where the system can only do the best to achieve some description is a *judgment* about appropriateness.

So, Yh manipulates descriptions in a description language. When a generation task is presented to it, that task is in the form of some representations about the objects that text will be generated about—the program—and a description of the generation task. Yh responds to the description and not the representations. Adding the representations informs Yh about a new world view—now there are new objects. If the request is in the form of a representation, then that would mean that the request is something Yh knows about, much as it knows about the new program, but that will not cause it to *want* to talk about the program; the description of the request will.

Another way of seeing the difference between descriptions and representations comes from viewing each representation in the system as an individual with special abilities. For instance, an individual may stand for some object in the world or it may be able to perform certain activities within the system. In Yh both of these functions are realized: there are units (representations) that are the meaning of a generic program, an input, a transformation, a word, and there are units that perform transformations, move things in and out of the agenda, decide what form a sentence will have.

A representation is then *appropriate* in a given situation or it is *inappropriate*. For instance, a unit representing a chair is appropriate when a chair is being recognized or a unit to insert a specific word is appropriate when a word with that given definition is needed.

Specifying when a representation is appropriate is the role of a description. So, associated with most representations is a description of when that representation is useful. This description is then used for matching with a situation to determine the degree of appropriateness.

One of the reasons for a descriptive system is that the choice of action may depend on longer range goals that need to be expressed. That is, there may be some background condition (such as not using words with certain connotations) that influence the activities over a long period of time, but not in a dramatic way. This is accomplished through the descriptive system while decoupling it from the representation system.

Finally, one may want to decouple the activities described above because there is a desire to make each unit diamond-like, representing a narrow but precise amount of information, and any stretching of that amount should not tamper with the structure of that unit.

Associated with units is a certain stiffness, in that a unit is a solid point or individual within the entire system. Through linking of descriptions with units, the stiffness is reduced because the decision of appropriateness is made through a matching operation that is decoupled from the representation. If this matching process allows one to stretch the applicability of a representation to a situation then there is less need to anticipate every eventuality with specific units: other units can make do in the unexpected situation.

The rest of this chapter will talk about the descriptive system a little bit; there is a global description of the current state of beliefs and desires, and there are descriptions attached to many representations. Many of the things that Yh does involves changing its behavior based on the description of what is important at the moment.

An example of the system behaving at the behest of its descriptive system will occupy much of the rest of this chapter. The point of this example will be to show how the descriptive system, which is not required to be consistent with anything,[3] is able to solve some simple problems. Some of these problems can only be done easily because the descriptive system is decoupled from the representation system as much as it is.

The central idea is that the global description is a set of beliefs, among other things, and the tide of belief can ebb and flow when the reactive component[4] is allowed to press forward. Once something is in a representation it is fixed, so that any such ideas of changing importance over a short period of time must be expressed through the descriptive system.

It is interesting to note that, empirically, the structured planning and control of the system is invariably the result of representation driven activities, the representation being structured, and hence the action; similarly, the unstructured planning is the result of the description driven activities, descriptions being unstructured.

5.4 System-goals

SYSTEM-GOALS is the special unit that is the holder of the island of planning mentioned above. As a unit it is of the form:

[3] Unlike the representation system, which is consistent to the extent that a world view must be consistent.

[4] The component that lets the descriptive system control sequencing.

```
{system-goals
 level 1
 influences ()
 goal-list ()
 soft-constraints ()
 plan-list (() . ())
 reply-method
 <some code>
meta meta-system-goals}

{meta-system-goals
 β system-goals
 method <a garbage collector call>
 plan-list ()
 purpose ((name (system goal unit)))}
```

The interesting entries are GOAL-LIST, INFLUENCES, SOFT-CONSTRAINTS, and PLAN-LIST. The exact usage of these will be explained later, but the general idea will be presented here.

Essentially, SYSTEM-GOALS contains a description of the state of belief of Yh at this time, along with it goals. A large number of meta-units contain descriptions of what they do in a form that can be matched against this unit. The result of the match is not simply a T or NIL, but a numeric measure of the strength of that match. As a numeric measure, there is the possibility of considering the best match rather than the first match. Additionally, there is a mechanism for making the matching process more sensitive to certain situations at different times by affecting the measure of a match. Thus, it is possible to talk about *influencing* a match rather than merely negating it.

5.5 Summary of the Matching Process

This matching mechanism is also a sequencing technique in that the unit with the best match is selected to be invoked next when the reactive component is operating. Finding such a unit, though, can also take the form of a search if no unit satisfies any threshold conditions imposed on the match.

Finally, the matching mechanism is also a simple problem-solver since, by maximizing the match, a kind of hill-climbing ability is provided. If there are variables which match non-constant data, the matcher can then bind those variables in the 'optimal' way for the situation as part of its maximizing behavior.

And, furthermore, by assigning values to these variables, if present, a parameter passing mechanism is provided over and above any others that exist.

This matching technique is called *hybrid matching* or *influence-based matching*.

The intuitive meaning of GOAL-LIST is that it is a set of patterns and measures of the desirability and accomplishment for the 'facts' those patterns represent. The exact format is:

```
(...(<pattern>
    (D_b . D_d)
    (A_b . A_d) . <rest>)...)
```

where <pattern> is some pattern in the pattern language to be discussed in Chapter 6, D_b is the measure of desirability of <pattern> being accomplished, D_d is the measure of undesirability of it being accomplished (there may be a difference of opinion expressed on this point by parts of Yh), A_b is the measure of accomplishment of <pattern>, and A_d is the measure of disaccomplishment (different parts of Yh may be working against each other). <rest> are internal comments, reflecting mainly who originally posted this

entry, although if the second comment is "flush-when-satisfied," a certain type of garbage collection can remove this entry. The structure of <rest> will be discussed in Chapter 6.

The INFLUENCES slot contains entries which affect the choice of action by the reactive component when considering the GOAL-LIST. This slot is of the form:

```
(...<pattern> . <amount>)...)
```

where <pattern> is in the pattern language and <amount> is such that $-1000 \leq amount \leq 1000$. If <pattern> is something that a unit claims to have something to do with, <amount> influences the measure of the evaluation of invoking the unit accordingly.

SOFT-CONSTRAINTS is a list of the form:

```
(...(<predicate> . <amount>)...)
```

which is used to affect the measure of a match by adding <amount> to it if <predicate> is non-NIL.

5.6 Example of a System Description

The following is an example of a system-goal unit that was used to explore the power of the reactive component without benefit of full planning. The reactive component is called that because it reacts to the current situation by matching abilities against it. As a simple problem solving technique, one would expect that some reasonable behavior is possible.

The main point of the example is to demonstrate the syntax of the descriptive language and show how the reactive component does some simple problem solving through the hill-climbing matcher. The descriptions shown constitute the form of the *solution* to the problem stated below and does not constitute the process of determining the solution from the problem statement.

The example is similar to the Blind Robot problem [Michie 1974], [Sacerdoti 1977], which was once proposed by Michie as the exemplar planning problem—if some program could solve this problem (easily) then the planning problem would be solved. Though in the form of a puzzle, and though it has a definite goal that is recognizable, there are aspects of this problem that are remarkable for a benchmark emerging from the problem-solving era.[5]

A robot is in a room with two boxes, a table, some red objects, and a door. The red objects are initially at the door. In this version it is assumed that there are four keys and an indefinite number of red objects. The keys are in the two boxes, but it is not known how many are in each box, and it is not known which key opens the door. There are no other objects. The robot has no perceptual abilities, does not know where it is, and does not know whether it has anything in its hand. The table is empty. The idea is to take a red object outside the room (through the door). Bringing the key that opens the door to the door opens it right away; taking that key away closes the door.

The main point of the problem is to be able to describe the fact that nothing is known for sure, and there is no possibility for ever finding out anything except by making it true. Thus it is not known where the keys are, and they all must be brought to the door in order to guarantee that it will open. But since the red object is already at the door, if the keys are brought there, and the robot does not get rid of the red object from the door, when the door is opened and the robot goes to pick up the red thing, it might grab the key and lock the door. So the red thing must be disposed of, and the matter of the possible object in the hand has to be dealt with.

[5]This problem is not exactly the one Michie stated because I was working from an old memory when I programmed it up. This version differs from Michie's in that the number of keys is known, but only one of the keys opens the door. The original problem had an indefinite number of keys, any of which opened the door, as well as an indefinite number of other objects. I think that given the way the problem Yh did is described, Michie's version is as easy or easier.

The fact that beliefs are described with measures means that the solution of this problem is fairly simple, though it may not be the case that a full, elegant solution can be attained with simply the reactive component. Recall that the reactive component simply does maximal pattern matches. The real solution of the problem may take some actual planning and self-observation.[6]

The location of the keys, then, is described with some confidence, and the activities of the robot are centered on pouring its confidence in where keys are from the boxes to the door, and making sure things don't get mixed up in the process. The reactive component is monitored by the rest of the system—the system that does this problem—although the observation is performed in a manner not consistent with the methodology outlined earlier.

Nevertheless, here is the description used:

```
{system-goals
level 1
influences
  (((confuse (restrict ? redp) (restrict ? keyp)) . -200)
   ((confuse (restrict ? keyp) (restrict ? redp)) . -200))

goal-list
  (((location door) (0 . 0) (50 . 0) ())
   ((at (key1 key2 key3 key4) box1) (0 . 0)(1000 . 0) ())
   ((at (key1 key2 key3 key4) box2) (0 . 0)(1000 . 0) ())
   ((at () outside)(0 . 0)(1000 . 0) ())
   ((at () table) (0 . 0)(1000 . 0) ())
   ((at (red) door) (0 . 0)(1000 . 0) ())
   ((in-hand (smthng))(0 . 0)(100 . 0) ())

   (empty-hand (0 . 0) (100 . 0) ())
   ((known-at key1 box1) (0 . 0) (230 . 0) ())
   ((known-at key1 box2) (0 . 0) (230 . 0) ())
   ((known-at key2 box1) (0 . 0) (230 . 0) ())
   ((known-at key2 box2) (0 . 0) (230 . 0) ())
   ((known-at key3 box1) (0 . 0) (230 . 0) ())
   ((known-at key3 box2) (0 . 0) (230 . 0) ())
   ((known-at key4 box1) (0 . 0) (230 . 0) ())
   ((known-at key4 box2) (0 . 0) (230 . 0) ())
   ((known-at red  door) (0 . 0) (205 . 0) ())
   ((disposal-site box1) (400 . 0) (0 . 0) ())
   ((disposal-site box2) (400 . 0) (0 . 0) ())
   ((disposal-site table) (600 . 0) (0 . 0) ())
   ((export red) (1000 . 0) (0 . 0) ()))
plan-list (())
meta meta-s-g}
```

[6] I want to make certain that there is no question about what this example is intended to show. First, there is a distinction between *description* and *representation*; this example shows some of the power of making a system act on the basis of a description rather than on the basis of explicit instructions. The exact details of this example are tuned to make the actions of the system appear to be carefully reasoned. As such, some parameters are well chosen. Therefore, I do not intend this to reflect a solution to the Blind Robot Problem, but a model for what the solution can look like if the descriptive power of the system is above a certain level. The real solution may lie in being able to determine these parameters well enough (they can be selected less optimally and still allow the system to perform adequate actions), or it may lie in being able to discover the need for parameterizations of a certain type. Yh does not solve the Blind Robot Problem, it provides a mechanism for expressing a reasonable solution.

The highlights of this are that the confidences of the locations are rigged so that it takes four attempts to move something from one place to another, which is the inverse of the probability that the object has been grasped. Any larger numbers here simply cause the system to make redundant steps, which is acceptable. The point is that the optimal solution can be expressed. The entries that are of the form `((at <set> <location>)...)` describe the totality of objects that may or may not be at a location at a given time. The GRASP and UNGRASP routines observe the total state and modify these into non-existence as the KNOWN-AT values decrease.

The cutoff for certainty is 200, so that anything below that is basically assumed false, but values $0 < x < 200$ are uncertain enough to cause confusion. So that SMTHNG in the hand is fairly ambiguous.

The DISPOSAL-SITEs are simply places that the robot can put things that are not the door.

Another interesting point is that there are influences that discourage the robot from confusing red things and keys. The method for a better system doing this problem would involve deducing that this should be the case, and doing a good job of observing the operation of the system as it performs these activities. The following is a truncated transcript of the actions of Yh:

```
Door    Box1    Box2    Table   Hand    Outside Unknown
----    ----    ----    -----   ----    ------- -------
RED     KEY4    KEY4            SMTHNG             ROBOT
        KEY3    KEY3
        KEY2    KEY2
        KEY1    KEY1

Trying: Export RED
Key not at door
Failed!

Trying: Transfer KEY1 from BOX1 to DOOR
Cannot transfer KEY1 to DOOR
       because that would confuse KEY1 with (RED)

Trying: Dispose (RED) from DOOR to TABLE
Hand not empty
Trying: Dispose (SMTHNG) from SOMEWHERE to TABLE
Moving to TABLE
Emptying hand: Hand contains (SMTHNG)

Disposing  (SMTHNG) from SOMEWHERE.

Trying: Transfer (RED) from DOOR to TABLE
Moving to DOOR
Grasping object
Grabbing something at DOOR

Moving to TABLE
Emptying hand: Hand contains (RED)

Disposing  (RED) from DOOR.
```

```
Door    Box1    Box2    Table   Hand    Outside
----    ----    ----    -----   ----    -------
        KEY4    KEY4    SMTHNG  EMPTY
        KEY3    KEY3    RED
        KEY2    KEY2    ROBOT
        KEY1    KEY1

Trying: Transfer KEY1 from BOX1 to DOOR
Moving to BOX1
Grasping object
Grabbing something at BOX1

Moving to DOOR
Emptying hand: Hand contains (KEY1 KEY2 KEY3 KEY4)

Door    Box1    Box2    Table   Hand    Outside
----    ----    ----    -----   ----    -------
KEY1    KEY4    KEY4    SMTHNG  EMPTY
KEY2    KEY3    KEY3    RED
KEY3    KEY2    KEY2
KEY4    KEY1    KEY1
ROBOT

Trying: Transfer KEY1 from BOX1 to DOOR
Moving to BOX1
Grasping object
Grabbing something at BOX1

Moving to DOOR
Emptying hand: Hand contains
                (KEY1 KEY2 KEY3 KEY4)

Trying: Transfer KEY1 from BOX2 to DOOR
Moving to BOX2
Grasping object
Grabbing something at BOX2

Moving to DOOR
Emptying hand: Hand contains
                (KEY1 KEY2 KEY3 KEY4)
```

etc...

```
Door    Box1    Box2    Table   Hand    Outside
----    ----    ----    -----   ----    -------
KEY1                    SMTHNG  EMPTY
KEY2                    RED
KEY3
KEY4
ROBOT
```

```
Trying: Export RED
Trying: Transfer RED from TABLE to OUTSIDE
Moving to TABLE
Grasping object
Grabbing something at TABLE

Moving to OUTSIDE
Emptying hand.
Hand contains (RED SMTHNG)

Door    Box1    Box2    Table   Hand    Outside
----    ----    ----    -----   ----    -------
KEY1                    SMTHNG  EMPTY   RED
KEY2                    RED             SMTHNG
KEY3                                    ROBOT
KEY4

Trying: Transfer RED from TABLE to OUTSIDE
Moving to TABLE
Grasping object
Grabbing something at TABLE

Moving to OUTSIDE
Emptying hand: Hand contains (RED SMTHNG)

Door    Box1    Box2    Table   Hand    Outside
----    ----    ----    -----   ----    -------
KEY1                            EMPTY   RED
KEY2                                    SMTHNG
KEY3                                    ROBOT
KEY4

Exporting  RED.
```

In other words, the robot moves to the table to dispose what's there so as to not confuse it with anything, then moves the red thing to the table because that doesn't confuse it with the keys. Then it makes a large number of trips to the boxes (four trips to each box) and brings them to the door. Then it gets something at the table and takes it outside. Finally, it does that again.

The actual descriptions of the units that do the processing are not that interesting. Mainly they are obvious given what they are for: `(export ?object)`, `(dispose ?object from ?location)`, `(at ?object ?location)`, `(transport ?object from ?location1 to ?location2)`, `empty-hand`, `(move-to ?location)`, and `grasp`.

They have the obvious pre-conditions, which they check. Disposal sites (where things are put temporarily) are chosen entirely by the maximizing matcher.

In the real solution to this problem, there would be a planner that would simply be able to reason about the task of exporting an item, though it might not be able to foresee the problem with confusing things.[7] In the solution here, the unit for transporting an object has the description:

[7] I observed someone who is not an expert puzzle-solver attempt to come up with a plan for solving it in the form given here. This person went through the steps of moving things around until it was realized that it was a bad plan to confuse the red thing with the keys. Then the red object was removed to the table right away. In a proposed system, I would think that it would go

```
goals (((transport ?a from ?b to ?c) . 750)
       ((at ?set ?c) . 10)
       ((confuse ?a ?set) . 900))
pre-conditions ((empty-hand . 900))
```

which means that as soon as the export unit posts the transport goal, the system sees the problem, and makes sure nothing can be confused.

5.6.1 The Real Blind Robot Problem

The real problem can be solved with the existing mechanism using the initial description:

```
{system-goals
level 1
influences
  (((confuse (restrict ? redp) (restrict ? keyp)) . -200)
   ((confuse (restrict ? keyp) (restrict ? redp)) . -200))

goal-list
  (((location door) (0 . 0) (50 . 0) ())
   ((at (key1 key2 key3 key4 something) box1)
    (0 . 0)(1000 . 0) ())
   ((at (key1 key2 key3 key4 something) box2)
    (0 . 0)(1000 . 0) ())
   ((at () outside)(0 . 0)(1000 . 0) ())
   ((at () table) (0 . 0)(1000 . 0) ())
   ((at (red) door) (0 . 0)(1000 . 0) ())
   ((in-hand (smthng))(0 . 0)(100 . 0) ())
   (empty-hand (0 . 0) (100 . 0) ())
   ((known-at key1 box1) (0 . 0) (200 . 0) ())
   ((known-at key1 box2) (0 . 0) (200 . 0) ())
   ((known-at key2 box1) (0 . 0) (200 . 0) ())
   ((known-at key2 box2) (0 . 0) (200 . 0) ())
   ((known-at key3 box1) (0 . 0) (200 . 0) ())
   ((known-at key3 box2) (0 . 0) (200 . 0) ())
   ((known-at key4 box1) (0 . 0) (200 . 0) ())
   ((known-at key4 box2) (0 . 0) (200 . 0) ())
   ((known-at something box1 (0 . 0)(500 . 0)())
   ((known-at something box2 (0 . 0)(500 . 0)())
   ((known-at red  door) (0 . 0) (205 . 0) ())
   ((disposal-site box1) (400 . 0) (0 . 0) ())
   ((disposal-site box2) (400 . 0) (0 . 0) ())
   ((disposal-site table) (600 . 0) (0 . 0) ())
   ((export red) (1000 . 0) (0 . 0) ()))
plan-list (())
meta meta-s-g}
```

Here it is expected that one trip to each box will do the trick.

through the steps, realize that confusing the red thing and the keys is bad and post that as an influence. Then the full description of the `transport` unit would see this right away – not too far off what the person did.

5.7 Summary of Examples

The examples were mainly intended to show the descriptive system describing some beliefs that a robot might have. In addition, by letting the system react to this situation with the abilities pointed out (of being able to grasp, ungrasp, etc), and with the units that perform these actions being able to observe preconditions, and modify the beliefs of the system (change SYSTEM-GOALS), the system was able to successfully 'solve' the puzzle by getting something red outside the door. By 'solve' is meant that the system is able to perform the description of the actions as given by the initial situation description. The real solution is to derive this description and the attendant 'pieces of advice.' About how this description is derived Yh has absolutely nothing to contribute.

The system accomplishes this by basically posting desires for moving the keys to the door, observing that the red thing at the door should not be confused with the keys, and then just slowly building its confidence in the keys being at the door, by making repeated trips. The pattern matcher finds a good location for the red thing—on the table.

There is nothing special about the 'planning' done here; the point is just the use of the descriptive system to solve the problem of modelling lack of knowledge.

5.8 Planning and Sequencing

Another way to view the various types of things that are going on in Yh is by a shift in control methodology. Simply stated, there has been a tendency away from traditional control structures towards agenda-like mechanisms. The reactive component of Yh is the next step along the same dimension.

Consider traditional sequencing in which control proceeds from statement to statement. The only exception to this is the conditional; by setting flags or variables earlier in the execution of a program, one can cause control to move in one direction or another later on, but essentially, the flow of control—or at least the possibilities for control—is set by the programmer very early on.

In an agenda system different agenda items are placed on an agenda along with priorities and predicates, or some other ordering primitives. The system must be attuned to handling the items placed on the agenda, of course, but the decisions about what and when to do them are left more uncertain and can be manipulated by the system itself. On the other hand, one often finds in practice that the agenda is simply used to cluster small groups of tasks that can be done in any order, but each group one after the other.

The reactive component builds on the agenda by making each agenda item a rich description of the things that need to be accomplished at that level, but with no admonition or advice about the decomposition within those tasks. This decomposition is left to the reactive component, which locates methods, by matching, that accomplish subsets of the tasks at hand. The tasks left over—not accomplished—are given to the next agenda item or island to handle.

Thus more and more of the decomposition of the methods for accomplishing things is left up to the system and not to the programmer.

5.9 The Lesson

The lesson learned here is that a descriptive system that allows a fair amount of uncertainty in what the situation is can sometimes help solve problems that are difficult to solve using traditional techniques. Being able to have some belief in the 'fact' that the keys were in two places at once—that is, that the system had some reason to believe the keys were in each place—allowed it to make some initial headway in the problem.

The other lesson is that it is the fullness of the description of `transport` that allowed the system to 'see' early that there would be confusion that would be disastrous later if the keys were moved to where the red object was. This *advice* was a given in this solution, but it is not obvious how this advice could be derived automatically.

In Yh it became abundantly clear that austere descriptions were a poor way to operate in a rich environment.

Chapter 6

Influence-based Matching

This chapter discusses the most complex part of Yh, the unit selection, or the hybrid matching process. This process has been loosely described in other parts of this thesis and is the box labeled MATCHER in the figure in Chapter 1.

6.1 Planning Versus Action or Participation

It often seems that in everyday life planning is rarely more than a matter of deciding what the best thing to do is without making an explicit plan. By this it is at least meant that there are many partial or whole plans that sit around waiting to be used with elaborate enabling descriptions of them (here a mechanistic terminology is being imposed on a non-mechanistic example) which are triggered by some deliberation on the matching of these descriptions to the current situation. This is essentially the view taken by Schank [Schank 1977] and many others. This is contrasted with the view that these plans are constructed on the fly each time.

Though building plans is a worthwhile and necessary type of activity, it is an activity like any other that must occur in some framework, which includes the ability to make the kinds of decisions about what to do next outlined above.

Maybe it is helpful to think about the difference between natural language generation as a problem for AI and 'formal' problem solving. In the latter case, problems are presented like those found in the Sunday newspapers or in puzzle books where there is a situation, some rules, and a goal in mind. An 'intelligent' person figures out how to apply the rules in order to obtain the goal. In the former case, one has something to communicate, some methods for accomplishing the 'goal' of expressing that something, and one does that until one is satisfied. Of course, there is a natural enough sounding comparison or analogy between these tasks. But the major difference is that there are, usually, a huge number of ways to accomplish the goal of expressing the situation, and that a large number of them—nearly all?—can work. Furthermore, there are a number of other factors which can contribute to how that process is carried out, and therefore, to what the final result will be. Thus the problem becomes one of how to decide what to do next.

It becomes apparent that what is needed is a mechanism for looking at the current situation as it is 'known' to the program and to be able to assess the best course of action from that examination.

This is called the *reactive* component of Yh, and has been discussed in the previous chapter to some extent. This is what happens on the large islands that the planning process provides. These islands are islands of description of some desired state.

6.2 Hybrid Descriptions

The main thrust of these description entries is to have the system, that is, its individuals, interact with each other and with these descriptions of some of the things it knows in such a way as to produce some interesting behavior, both external, observable behavior, and internal, trace behavior.

Roughly speaking, there are several ways to view the activity of an interesting object, especially artificial objects with purposeful behavior. The first is to think of the system as waiting for some set of goals to satisfy (work on), at which point powerful techniques are brought to bear so that the capabilities of the system bring about the completion of the goals. A second way is that the system simply exists and is always acting. Sometimes this activity is quiescent (that is, the activity is uninteresting at the moment), and sometimes it is perturbed by something which causes a change in behavior which is once more interesting. In a language generation system, the system is perturbed by planting a desire to express some situation, which will be expressed to some satisfaction. A better example is a timesharing or other basically interrupt driven process.[1]

These two views, of course, are not incompatible, but the latter has at least one worthwhile implication that is more apparent from the outset—namely, that it is nowhere mentioned that the quiescent state, if it exists, is the same as all other quiescent states. The state of the system is a product or natural result of its past. One should not expect that there would be a large number of equal values for this function. So a perturbation results in a different quiescent state, and it is expected that if a similar request was made of the system, it might not respond the same way: the same way people are expected to change as more experiences are gained. This is to be distinguished from learning, in which by some external measure progress is made or better responses are given. Here there is no learning, but success is measured by the fact that the system stops reacting in some time and survives in a new state to react on other occasions.

The philosophy of this system is that it exists and reacts to influences brought to bear both by itself and by the 'external world' in the guise of externally introduced entries on the above-mentioned list of descriptions. This list is called the 'system-goals,' somewhat of a misnomer in that it contains descriptions of the state of things, counter-goals (or anti-goals) as well as the things normally thought of as goals.

In having an object, in this case an artifact, act in an interesting, though possibly not intelligent way, there seems to be a need for a number of mechanisms that interact in a complex manner. It doesn't seem that one should be able to find a half dozen mechanisms that have the richness to exhibit the kinds of things one would expect to see in a full natural language generation system. Certainly a human mind is more complex than an operating system, and perhaps one should not be able to see any sort of truly appropriate behavior in anything less complex. So far it has been hinted that there are several mechanisms that are present in Yh: units in a meta-style relationship, communication to both be used as traditional communication and as the basis for self observation, levels of activity to monitor behavior, and finally what is called 'influence-based matching.' This last item has also been called 'goal choice' or 'planning' herein, and now it will also be called *searching* in order to throw the full weight of its aspects onto the reader.

To some extent, behaving in an interesting way is no more than reacting to a situation without much thought or standard planning.[2] Presented with a situation or with a what-is, and given a set of possibilities, one is selected which is appropriate to the what-is. As with anything else the game is to formalize the what-is, the potentialities, and turn the selection into an algorithm.

[1] Many of these ideas originate from the work of Maturana and his colleagues [Maturana 1970], [Varela 1979]

[2] My view is that in day to day activities a person acts routinely, as if encountered situations are familiar enough that standard plans with some slight modifications can be applied, rather than elaborate new plans constructed. Speaking is such an activity in that people can generally ramble on without really stopping to think for tremendous amounts of time. There are many things that people do that argue for recognition as the main mental process rather than hard core thinking. This is what I mean by reaction, recognizing a familiar situation and applying a related plan.

6.3 A Review of the Situation

Meta-units contain entries labeled: PURPOSE, GOALS, PRE-CONDITIONS, CONSTRAINTS, PREFERENCE, ADDED-GOALS, SOFT-CONSTRAINTS, PAIRING-FUNCTION, INFLUENCES, and COUNTERGOALS. These entries are used by the goal choice process to 'match' a unit's advertised behavior against the needs of the system as measured by the system goals. This matching process is quite complex, and reflects, in some ways, the fact that the architecture of digital computers may not be ideal for AI, though probably components of it are. However, if the converse is taken to be true, namely that only those things which are relatively simple and efficient to do on current digital equipment are relevant to AI, then perhaps the problem with progress is that the natural complexity giving rise to interesting behavior will be left out of these systems.

The system-goals 'workspace,' which is a unit like any other, but one whose primary purpose is to store information as a kind of description sink/source, contains a list of entries and a list of 'influences,' described below. These individual entries are then considered together as a description of the relevant parts of the situation for formulating what to do next, though it may not be the entire description in absolute terms.

The game is then to find the unit which best 'matches' this situation in some sense of the word. To accomplish this, it is necessary to bring the description of the unit into contact with the system-goals so that the points of comparison can be made; in the end, it is desirable so obtain a measure of the strength of the comparison and thereby choose the best one. Entries in the system-goals unit are paired with entries in the unit under consideration—in particular, with entries in the GOALS slot. Since later it is important to know the points of comparison that led to a rating, a record will be kept of the pairings of items. So the interesting parts of this method are to produce a pairing and then to measure the contribution to the strength of match by each pair in the pairing.

In other words, this is very much like a plausible move evaluator for a one move look-ahead tree search algorithm: there are a number of units that can be invoked next, based on having a description that resembles the desired state. However, some resemblances are better than others, so a numeric measure of the resemblance is calculated.

In any case, let us turn to the method by which an individual is selected to operate on the current situation.

6.4 Overall Goals of Matching

Initially there is a set of descriptions which describes the current beliefs the system has. The important feature of this description is that it is a combination (unordered) of patterns which tell something of the situation. For example, a pattern might be:

```
(known-at red table)
```

which states that there is something red at the table. Of course, this syntactic object does not 'state' that, but rather this is the form which causes the system to behave in such a way that one could say that its belief is that it knows that something red is at the table.

The existence of the pattern in the set of patterns representing the description can not and should not be taken as an absolute avowal of a belief in that object. Recall that in the unit system talked about earlier, one can have a unit that says something while its meta-unit can deny it. Similarly the description of the situation has the same quality without recourse to a lot of meta-type overhead.

Thus, there is a measure of the faith in the description; this measure is, again, strictly meaningless without the behavioral impact it has on the system. Think of this measure as affecting the strength of the belief, the attention that gets paid to it, or the importance it has in the description of the situation. This

idea is only the next reasonable step beyond the all or nothing style of description used in some other AI systems. The claim is that there are some interesting behaviors associated with this scheme.

It is also important to represent in a uniform way the desires, goals, and predilections that the system has. Since a desire can become a 'fact,' a second measure of this desire is added to the items that constitute the situation description. So far, then, as part of each item there is a pattern consisting of a formal description of the item, a measure of the interest that the system has in that item being or becoming a fact, and a measure of the extent of belief the system has about the item.

An example of why the measure is associated with the description is by considering matching two descriptions, one of a paradigmatic object and the other an instance. By letting the important descriptive elements of the object have a higher weight. The unimportant details, which are useful for distinguishing the actual individual described, could have a lower weight and would only be considered once the important attributes have been accounted for.

So when one is trying to tell if an object is a table, the important table-features will dominate the match, while once it is ascertained that there is a table, the lesser weighted features, like color, dominate. Finer distinctions rise to the fore once the larger, measurably dominant ones are equal.

The problem that this approach solves is that of stating in a uniform way the fact that some features are to be considered above others in certain cases. If all features are given a yes-no status, then one can end up in the bind of making decisions about the identity of objects based mainly on their color.

Of course, an all or nothing scheme can be made to work by making buckets of importance in the descriptions, so that there is a discrete gradation of the importance. The possible advantage of a numeric system would be 'continuity' and the ability to also obtain easily a measure of the strength of a match.

Up to this point there is a place for descriptions of the desires and beliefs the system has. In addition there are descriptions—advertisements—associated with many units; a reasonable thing to do is to try to match one of these descriptions against the current situation description. Since the situation description can contain things that are not of interest, and since one does not want to get a strict yes or no answer (in all cases) this matching must take into account the measures just talked about. The descriptions attached to units just contain an absolute measure of importance, unlike the ones in the situation description since the former just measure strength, while the latter also measure degree of belief and degree of desirability.

If, for some units, there are descriptions of what each of those units can do in certain cases, and if there is a description of the current situation, one might want to find the unit whose action description matches best the situation description. There are two reasons to want to make such a match, and therefore two distinct types of matching. The first type matches the advertised purpose of a unit against the situation description viewed as a statement of the desired situation in order to decide which unit to activate; the second type matches the identity description of the unit viewed as a description of the object further specified by the unit against the situation description viewed as a description of an object.

Unit selection matching will be the focus of the rest of the discussion in this chapter.

6.5 Unit Selection

The matching process allows for different kinds of 'facts' to exert their 'influence' on the situation. Several ways that these influences can be expressed and the ways in which these expressions alter the outcome of the matching process will be presented.

The matching process will be presented with motivations interspersed with the explanation.

[Note: the rest of this chapter deals with the details of the matching method, which is derived from a method of numeric weighting used in [Shortliffe 1974]. The rest of this chapter can and should be skipped by those who do not need to see these details.]

6.5.1 Single Descriptors

First consider the case where a unit has as its description a single entry, which is a pattern and a measure. The situation description[3] is then matched against this single entry. To do this, the situation description is scanned until an entry in the situation description has a pattern that matches the pattern in the unit description. Some computation is then carried out to see if the measures are in accord. In this case of goal choice matching, it is the 'desirability' that is compared with the measure in the unit description.

Remember that each 'fact' in the situation description looks like:

```
(<pattern1> <desirability> <accomplishment>
            <comment><minor action> .
            <more comments>)
```

and that the entry in the unit looks like:

```
(<pattern2> . <amount>)
```

The first requirement is that <pattern1> and <pattern2> *match* under some definition of *match*. This match is done by a pattern matcher that is described later in this chapter.

If both <desirability> and <amount>have the same sign, then they are in accord and the measure (in any case) is given by:

$$\texttt{<desirability>}\frac{\texttt{<amount>}}{1000}$$

Notice that without the measures, the matching so far is much like that found in MICRO-PLANNER, in that each 'thing' has a single pattern that is matched against the 'goal,' in the MICRO-PLANNER case, once more, a single pattern.

In all cases, the range of measures is $-1000 \leq m \leq 1000$, which is to be viewed as strengths $-1.0 \leq p \leq 1.0$ multiplied by 1000 and then using integer instead of floating arithmetic for efficiency.

In this simplest of cases, when a selection is occurring, for each pairing of the unit descriptor with a situation description descriptor, the measure is computed. The pairing, which is a correspondence between the descriptor in a particular unit and the description in the situation description, with the highest measure is selected. Here, this is simply no more than finding a method for the most pressing desire.

Once the numeric scheme is used to pick which unit to apply in a situation, there must be ways to ensure that a number of different units which advertise the same abilities get used from time to time—or else, why have them at all?

Of course, there are conventions, methods, and mechanisms to allow a unit with a lesser measure to be applied. There is the method MICRO-PLANNER uses. Namely, the ordered list of candidates is available during execution, so that this list can be traversed or examined (actually more in the vein of CONNIVER). Also, there is a convention built in that adds a bit of preference to those units which display more variety over the history of the system, most things being somewhat equal; this could allow some of these underdogs a chance. There is the possibility of actually modifying the measure of the entry to change the behavior of the system. If done much in practice, though, this tends to lose the original values, which may have been chosen in order to encourage certain tendencies. Next one can add, in another slot in the unit, a preference, which acts as an additive factor in computing the measure, so that one can change the value of a measure without having to forget the original. There is an added convenience of being able to decay this preference automatically and have the preference disappear at a certain point. The preference entry actually is a list of preferences with predicates attached so that the preference modification can be added only in certain cases.

[3] Recall the *situation description* is a synonym for the entries in the GOAL-LIST slot of the SYSTEM-GOALS unit.

The last method used to similar to one that solves a related problem, which is one that comes up as the unit is used in practice and it is seen that it really doesn't work well in a particular case or set of cases.

Consider a group of units for interacting with a simple blocks world with blocks, pyramids, a table, a hand, and possibly some other simple objects; assume there is a unit that states simply that it can do this:

```
(move <blockp ?a> onto <objectp ?b>)
```

If it happens that the unit to do this works fine except when ?b is a pyramid (an obvious problem) one may want to add a proviso that this case should not be tried. One way is to make that entry look like:

```
(move <blockp ?a>
 onto <objectp ∧ ∀(x)(not (pyramidp x)) ?b>)
```

On the other hand, it might be the case that the effectiveness is in doubt and the goal is to only dissuade the unit from being used in this case unless it is the only candidate. The above solution actually forbids the unit from being considered because the predicate, `<objectp ∧ ∀(x)(not (pyramidp x)) ?b>` is absolute. One might want to dissuade the unit from being called in some cases only because it is inefficient: for example, the above unit might try to physically alter the pyramid by chopping off its top or by cementing on some platform, so if the idea is to minimize alterations, dissuasion is the proper technique.

To do this there is yet another mechanism in the COUNTERGOALS slot on the meta-unit in question, which is of the form:

```
(...(<pattern> . <amount>)...)
```

so that it is possible to say in this slot:

```
(...((move <blockp ?a>
           onto <pyramidp ?b>) . -600)...)
```

in order to dissuade this from happening, where -600 is some arbitrary[4] dissuasion factor.

The mechanism of the dissuasion factor is to scan the instantiated patterns in the GOAL-LIST slot in the system-goals unit, and if they match, the dissuasion factor is 'added in,' for some definition of 'added in' to be precisely given later.

Another situation where it is reasonable to have other outside things to influence the strength of a match is when there are some general principles or pieces of advice that the system should have. In the Blind Robot Problem, there was the admonition: "don't confuse a key with anything." So there is mechanism for that too, though, again, it is possible to embed the same effect in the situation description by fooling with the entries. This mechanism is the INFLUENCES slot in the situation description (SYSTEM-GOALS). This is the same format as the COUNTERGOALS entry and is matched against every goal in the unit being matched that goes by in order to modify the measure of the match. In fact, each meta-unit with an advertisement also has a INFLUENCES slot itself, which is used to modify the numeric score of the match by scanning the situation description entries.

Each participant in the match can bring to bear SOFT-CONSTRAINTS, which are predicates with an associated numeric value, and if the predicate is true, that value is 'added in' to the final score.

A related further influence on a match are the PREFERENCES, which are predicates and measures exactly like SOFT-CONSTRAINTS, but with names associated with each one and some mechanisms for decaying the measures over time automatically.

[4]For the purposes of this discussion.

6.5.2 Decisions and Thresholds

One sets a level of the strength of a match, above which it is considered that the match succeeds whereas below it fails. Therefore, one can say that in practice there is a mapping that turns the whole process into a standard yes-no situation. Unless there is some way that this strength of match is usable in an interesting way, then it is just another superfluous and possibly only incidentally interesting object.

First of all, one can rate the possible matches by comparing the measures of strength. Thus of any two matches it is possible to state which is stronger and, hence, more plausible.

Second, one can have other 'facts' exert their influence over the match in a uniform, controlled way. So the fact that, for some reason, two patterns don't match so well in this case can be expressed, without facing the dilemma of deciding whether that means they definitely do or do not match.

Specializations of a match can be dissuaded from occurring, by having specializations of the patterns negatively influence a match. Thus, one can start out with a general unit and, though its abilities don't change, its description does until it is well tuned. Descriptions of a unit can be changed either by adding new descriptors or by adjusting the measures of existing descriptors. This is the speculative area of 'learning' or 'evolution' in a limited sense. Although it's a fun topic, very little will be said about it because the exact mechanisms haven't been explored much.

6.5.3 Multiple Descriptors

The really interesting behavior occurs in the case where a unit has a description with more than one entry and/or there are pre-conditions or added-goals. In this cases, it is necessary to establish pairings between entries of the situation description and the goals of the unit.

Suppose there is such a unit and a situation description with at least as many entries. There may be more than one way to pair the entries in the unit description with the entries in the situation description. So, if the unit description is:

```
(((on <blockp ?a><blockp ?b>) . 500)
 ((on <blockp ?b><surfacep ?c>) . 500))
```

the situation description might be:

```
(((on blocka blockb) (0 . 0)(600 . 0)...)
 ((on blockx blocky) (0 . 0)(500 . 0)...)
 ((on blockb table)  (0 . 0)(700 . 0)...)
 ((on blocky table)  (0 . 0)(400 . 0)...)...)
```

If these two descriptions are matched (using not the goal choice matching process, but the identification matching process for simplicity[5]) the following results. If the assignments for ?a is BLOCKA, ?b is BLOCKB, and ?c is TABLE, the measure is 545; if it is ?a is BLOCKX, ?b is BLOCKY, and ?c is TABLE, then the measure is 400. The actual result looks like this:

[5]In order to ease any confusion at this point, the only real difference between the goal choice matching process and the pure matching process is that the former uses the desirability of the entry on SYSTEM-GOALS to measure appropriateness, and the pure matching process uses the accomplishment entry. Also, the IDENTITY slot from the unit instead of the GOALS slot is used as the unit description. There are also variants that match units against units and SYSTEM-GOALS against SYSTEM-GOALS.

```
(((545 . 0)
  (((((on ?a ?b) . 500)
      (on blocka blockb) (0 . 0) (600 . 0) nil)
    (((on ?b ?c) . 500)
      (on blockb table) (0 . 0) (700 . 0) nil))
   ((?a . blocka) (?b . blockb) (?c . table))))
 ((400 . 0)
  (((((on ?a ?b) . 500)
      (on blockx blocky) (0 . 0) (500 . 0) nil)
    (((on ?b ?c) . 500)
      (on blocky table) (0 . 0) (400 . 0) nil))
   ((?a . blockx) (?b . blocky) (?c . table)))))
```

In this match, there was a constraint, namely that the variables ?a, ?b, and ?c must be bound after the pairing, to eliminate degenerate matches. There are cases when the degenerate matches are acceptable and have been planned for, which will be discussed soon.

What happened is that the matcher was effectively looking for the best possible assignment of variables that reflected the situation of a short stack (?a on ?b on ?c). The situation description, though, did not have absolute knowledge of the real situation: there was some uncertainty in its 'mind,' in the form of not being sure what was on what.

Once ?a is assigned, there is only one consistent further match possible: namely, if ?a is assigned to be BLOCKA then ?b must be BLOCKB and ?c must be the table; if ?a is assigned to be BLOCKX, then ?b must be BLOCKY and ?c must be the TABLE. The idea is that there is a measure associated with each such choice—the choice of assignments—and the matcher attempts to assign the variables in such a way as to maximize the measure. Therefore, the matching process can be looked upon as an attempt to find the optimal assignment of variables in an uncertain data base.

The fact that there is such an optimization going on implies that the basic process is in fact the node generating stage of a search in the space of possible pairings of submatches. As hinted above, this search is subject to a number of constraints, or influences, that enter at various points and represent the expression of a type of effect that is wished on the behavior of the system.

The major constraint imposed on the multiple entry case is that the pairs must all have pattern portions that match according to the 2-way matcher (unifier). In general it is stipulated that any pattern might match the empty pattern at the toplevel.[6] This means that it is possible for two descriptions to match even though not all of the descriptors in unit description have been paired. Suppose there is the situation description:

```
(((on blocka blockb)(0 . 0)(1000 . 0)...)
 ((on blockb table) (0 . 0)(1000 . 0)...)
 ((on blocka table)(700 . 0)(0 . 0)...)...)
```

and the unit description:

```
((on <blockp ?a> table) . 600)
 ((blue ?a) . 500))
```

these would match, though `(blue ?a)` would not be paired with anything. The situation description, by the way, states that the system believes with total certainty that BLOCKA is on BLOCKB and that BLOCKB is

[6]This stipulation is actually subject to the whim of the pairing function that is used, and this is subject to the whim of the user setting the default for this function. At this point several are used frequently. Some pairing functions allow a null pattern to match any pattern, but the most efficient pairing functions attempt to pair maximally, matching in the order of appearance of the entries in the unit, where it is assumed that earlier matches constrain the possible other matches. This, for example, is true in the short stack case, where one assignment determines all of the others.

on the TABLE. It desires fairly strongly that BLOCKA be put on the TABLE, but this is not now the case. The unit description describes a unit that will put a block on the table and turn it blue. To 'understand' this unit, one could imagine that the action of moving the block would be done with a manipulator having as a hand a pair of paint brushes that are covered with wet, blue paint.[7]

The measure of that match is 420, with ?a bound to BLOCKA. What makes this interesting is that one can describe fully the actions of a unit, and if any of the incidental actions are not relevant, then that unit can be applied even though it has an odd side effect. Suppose there was an influence that didn't want BLOCKA to be blue with strength -400. Then the rating would be 220.[8]

6.6 The Exact Process

The exact process involves finding all units that match, and finding the match with the best or most appropriate score. The following is the exact process of evaluating the unit once it is decided to try it.[9]

Each 'fact' in SYSTEM-GOALS looks like:

```
(<pattern><desirability><accomplishment>
          <comment><minor action> .
          <more comments>)
```

The <comment> field is generally used to indicate which individual last modified this entry. These entries are kept in a simple list structure, since accessing them is done as part of computations which dominate the access time.

The fields. <desirability> and <accomplishment> are of the form:

$$(m_b \quad . \quad m_d)$$

where $-1000 \leq m_d \leq 0 \leq m_b \leq 1000$. m_b stands for the 'measure of belief' of the item, and m_d stands for the 'measure of disbelief.' The intent is that the belief in an item is the sum, $m_b + m_d$, and the reason that the positive and negative portions are separated is that this way it can be seen whether there were any negative or positive contributing factors in the final value.

A typical entry in the GOALS slot in a unit looks like:

```
(<pattern> . n)
```

where $-1000 \leq n \leq 1000$.

The first step in the process is to pair items from the GOALS slot of the unit with entries from the SYSTEM-GOALS unit. This process is controlled by one of several algorithms for doing the pairing, and the choice of algorithm is entirely under user control. At the moment, there are two basic versions, with two variants each to these versions.

Essentially, the idea is, given two sets of items, A and B, make a pairing, $(\ldots(x_i, y_i)\ldots)$ such that $\forall i, x_i \in A, y_i \in B$ and $P(x_i, y_i)$ is true for some specified predicate P.

The predicate, P, in most of these cases is the matching predicate UMATCH, which is a two-way matcher of tree structures.

[7] This cryptic passage occurred in the original manuscript without explanation from the author. Given his obviously unstable mental state, and his well-known violent temperament, we have chosen to leave it—The Editors.

[8] The method calculating these final measures will be presented later in this chapter, but suffice it to say that it is NOT simply addition and subtraction.

[9] The rest of this chapter is fairly technical and dense, describing in detail the matching process and the calculation of final measures for a match within the context of selecting the unit to invoke given a situation description.

6.6.1 The Basic Matcher

<pattern> is a LISP tree structure consisting of atoms (constants) and so-called 'match variables.' Two patterns match iff there is a correspondence between the nodes in each tree that reflects similar structure as trees and whose nodes match. Constants match iff they are EQ in the LISP sense. Match variables can have any of the following forms: ?, *, ?<var>, *<var>, =?<var>, =*<var>, and the special forms:

($r ? $pred_1$ ... $pred_n$)
($r ?<var> $pred_1$&...$pred_n$)
($r * $pred_1$ ... $pred_n$)
($r *<var> $pred_1$ ... $pred_n$)
($ir * $pred_1$ ... $pred_n$)
($ir *<var> $pred_1$ ... $pred_n$)

These match variables match any datum in the data pattern if they are of the ?-variety, or match any number of data if they are of the *-variety. Thus (foo * bar) matches

(foo is a funny word and so is bar)

and (foo ?) matches (foo bazola). In addition, variables which begin with these characters match the same way, but they may or may not, according to user specification, set the LISP variables as well—the so-called 'assign variable' mode. So, if (foo ?b) is matched to (foo bar) ?b has the value 'bar' after the match. Moreover, if some variable occurs in one place in a pattern, each occurrence must match the same thing. Thus the assignments must be consistent.

One can also demand that these variables can match anything that satisfies some given predicates, which is the purpose of the 'restriction' variables—those that are a list starting with $r or $ir. For example, (foo ($r ? numberp)) matches (foo 7) but not (foo bar). The variable in the $r syntax can be any of the types mentioned so far, and in the case that it is a *-variable, the predicate must be true of the list of things tentatively assigned to that variable. So one can say: (foo ($r * (lambda (x)(= (length x) 7)))) which means that there must be a list whose first element is foo and which has 8 elements. If one wants to have some predicate true of each element of a *-variable, one uses the syntax ($ir ...); for example: (foo ($ir * numberp)) matches a list whose first element is foo followed by any number of numbers. Also, if the *-variable is assigned to a null list, the $ir predicate is assumed to be true.

The matcher is a true tree-matcher so that correct backtracking to consistently assign variables is done; since there are *-variables, which can match any number of items, the choice of how many will allow a consistent assignment involves backtracking. Moreover, the matcher is restartable if that is specified at the first match; that is, one can obtain the next consistent match for two objects if, during the original match, the user specifies that a restart might occur. This is because the matcher can explicitly save the state of the match and re-compute that state in order to force an artificial failure, causing a backtrack, and hence making the matcher believe it could not find a consistent match with the current situation. The reason a state needs to be saved is that the variable, *, does not remember what it matched as a variable of the form, *<var>. Note that the matcher cannot be restarted unless it is in 'assign *-variable' mode.

Finally, the matcher is a two-way matcher, so that variables can be in either the 'pattern' or the 'data.' There are two modes for restriction patterns—($r ...)—interacting with ?-variables and *-variables, one which allows the pattern variables to be examined by restriction clauses, and one which assumes that all restrictions are true of match variables. The first mode is default. There are various ways to specify that the values of match variables are to be assumed from another source, either an alist or the old (LISP) value (the

'=<match variable>' syntax) So if the variable, =?a, appears in a pattern, LISP evaluates the symbol and plugs in that value in place of the variable.

The pattern matcher is internally tail recursive and continuation-based, using the same technique that McCarthy eventually used to show that the Samefringe Problem [McCarthy 1977] could be easily solved without co-routines.[10] Suppose that there is some recursion to occur on a tree-structure in normal CAR-CDR recursive order, and suppose that the decisions are being made at various points in the execution of the function. Once the CAR recursion is completed, there is no way for failure in the CDR recursion to back up into CAR recursion, because these recursions are at the same level. The idea, then is to make the CDR recursion be a sub-computation of the CAR recursion, which is accomplished through continuation passing, in which the CDR is passed as an explicit argument to the routine, which then calls itself on the CDR explicitly within the CAR call. McCarthy calls this the *gopher* technique[11]

This matcher forms the base level for most of the interesting actions taken by Yh, and the chooser. It is used with the same elan that EQ is used by most mortals. In the Blind Robot problem, it is called 29665 times; the total execution time for that problem was on the order of 50 cpu seconds, so that a large number of early failures occur in the matcher, which is a desirable situation.

6.6.2 The Combinatorial Pairing Function

The pairing function produces pairs of entries from the SYSTEM-GOALS and the GOALS slot of the unit in question. In the first of the two versions (V_1), there is no assumption made that any of the entries from the unit are interdependent, and it assumes that is is alright to pair a pattern from the unit with no entry from the situation description. A variant on this assumes that the only reason to allow an entry from the unit to be paired with nothing is that it does not have a matching entry. So in the first case, the predicate is

$$P(x, y) = T \quad \text{iff} \quad (\text{umatch} \quad x \quad y) = T$$

whereas in the second case, the predicate is

$$P(x, y) = T \quad \text{iff} \quad (y = NIL) \vee ((\text{umatch} \quad x \quad y) = T)$$

The second version (V_2) assumes that there is a strong interdependence between the entries in the GOALS slot in the unit, and the predicate is performed in series. This second version uses the fact that the matcher, UMATCH has the side-effect of assigning variables from a pattern, and the =<match-variable> syntax is used to make the interrelationship explicit. In addition, the order of appearance of the entries in the slot is very important. The variants on this are the same as above: whether or not a nil-pair is allowed.

V_1 is a more powerful pairing function in that it also allows one to specify the set of elements from the second set which must appear in a pair in a pairing. This is in conjunction with the nil-pairs option.

In each version, it is not allowed for a pairing (set of pairs) to contain two pairs (x_1, y_1) and (x_2, y_2) such that $x_1 = x_2$ or $y_1 = y_2$, where each element in the sets are distinguished.

The choice of pairing function is mandated by the entry in the PAIRING-FUNCTION slot in the unit first, the default pairing function (a global user variable), or V_1, nil-pairs variant, in that order depending on which is non-NIL.

[10] The matcher, by the way, is a 2500 line, single function which is explicitly tail recursive. The code is highly optimized and is therefore unreadable. The single direction matcher is heavily used by the SAIL MacLisp system.

[11] These separate inventions were made independently, with this continuation pre-dating the McCarthy Samefringe solution. In my case, I was unaware, basically, of the continuation literature, and have since learned that this technique has been standard for many years. In my case, the continuation is simply an explicit data structure that is passed, representing the rest of the pattern and data to be matched. McCarthy does the same thing, but if you optimize his data struture, you obtain a tree rotation (to the casual observer) rather than a continuation.

6.7 Numeric Matching Continued

The fields <desirability> and <accomplishment> are dotted pairs:

$$(m_b \quad . \quad m_d), -1000 \leq m_d \leq 0 \leq m_b \leq 1000$$

They are similar to the 'confidence level' used in medical diagnosis with MYCIN-like systems [Shortliffe 1974], [Tracy 1979]. In fact, these computations are taken directly from MYCIN and adapted where necessary. m_b is the 'measure of belief' or the amount of desirability there is to accomplish this item when it appears in the <desirability> field, and it is the degree of accomplishment when it is in the <accomplishment> field. m_d is the 'measure of disbelief' or the amount of undesirability, or the amount of disaccomplishment or anti-accomplishment, in the respective cases. These numbers roughly represent the percentage (times 10) of desirability or accomplishment in the obvious way. The *confidence level* is defined as the sum of m_b and m_d:

$$\text{Cf}((m_b \quad . \quad m_d)) = m_b + m_d$$

There is some level at which it is decided that the confidence is 'acceptable,' and this is stored in the global variable 'acceptance-level,' currently 200. There is also a superior level ('superior-level'), which is used in some cases, and is currently 800.

The meaning of the acceptance level is that it is the threshold above which the system is willing to concede that a fact is basically true. The superior level is that level above which the system believes a fact with good confidence. The precise technical usages will be shown later in this chapter.

The first comment field, as noted, is usually the description of the individual that modified the entry, the next field is the <minor action> field, which is used to help later individuals or the controlling process to decide what to do when certain conditions arise with respect to this entry. For example—the only example to date—there is an entry that can go here that says as soon as this goal is accomplished (above the acceptance level), remove this entry from the list of entries. This normally is used to set up temporary goals (which can be ignored) but which disappear when they have been accomplished since it was only for some other side effect that they were done in the first place. More will be said about the techniques for programming in this odd language later. As will be seen, there are a large number of different ways that this system can be affected, and a number of hacks can be performed in order to imitate some of the typical control structures found in other languages. In ways it is unfair to really use these techniques, so they are avoided in any real applications.

The <more comments> means first that the length of the entries is not important. Secondly, these comments are generally used for passing other parameters around to other individuals that might be accessing them. This technique is not a particularly recommended one, and is not used currently.

The GOALS entry on units is made up out of dotted pairs of patterns and numbers such as (<pattern> . n) where pattern is a statement in the pattern language above of what goal this individual purports to further and n is a measure of the degree to which that goal is furthered. So if that number is 750, then the individual goes a long way towards satisfying that goal. The GOALS entry in the individual is a list of such things, since there is no reason to suppose that only one goal can be satisfied by one unit. In general the order in the list does not matter, so it is actually a set, unless the pairing function V_2 is used.

The IDENTITY slot on units is exactly the same format as GOALS and is used in the matcher exactly as GOALS, but is used in the context of matching the identity of a unit rather than matching abilities. In the rest of the discussion on matching, the case of matching abilities to desires (goals) will be assumed.

The entry PRE-CONDITIONS is similarly formatted and each item represents a pre-condition that needs to be satisfied before the unit can apply. As usual, the number represents the degree to which the pre-condition

needs to be satisfied before the unit can work, but this number can be small or negative, which means that the pre-condition is unimportant or needs to not be true before the system proceeds. As will be seen soon, the pre-conditions play a funny role in that a unit can be invoked without all of its pre-conditions being satisfied. It is up to the unit to make a final decision about how important these things are, while the goal choice mechanism only measures the appropriateness of a unit to a situation.

The entry ADDED-GOALS is formatted the same way as GOALS, and each item represents the goals that are changed, in addition, to the items in GOALS. These are used in the matching and evaluation process to see if these side-effects conflict with other items in the situation description. Additionally, if the actions of the unit are simulated the pre-conditions, goals, and added goals are used to manipulate the situation description.

6.7.1 Pairing

The first thing that happens is that all possible pairings between the goals specified in the meta-unit's description and the goals in the system-goals is generated. This generation is combinatorily terrible, but is reduced slightly by the application of some constraints, as indicated by the 'constraints' entry and the basic filter, which is the matcher described above. The constraints are of the form:

((<pattern> <condition> <e_1> ... <e_n>) ...)

The <pattern> is simply for indexing on these entries. If <condition> is true, then all of the e_i must be as well. These constraints are applied each time a possible pairing is made. The filter applies for each element of the pairing. For a concrete example, look at an actual set of goals from the blind robot domain:

```
{bindings (?a ?b ?site ?set)
  purpose ((dispose ?a from ?b))
  pre-conditions ((empty-hand . 800))
  soft-constraints (((boundp '?site) . 20))
  constraints ((bound-disposal-site
                t
                (boundp '?a)
                (boundp '?b)
                (boundp '?set)))
  goals  ((dispose ?a from ?b) . 750)
         ((disposal-site ?site) . 300)
         ((at ?set ?site) . 150)
         ((confuse ?a ?set) . 200))...}
```

Here there are a few things to point out. For instance, there are a fair number of goals listed, only one pre-condition, a purpose and a bindings entry, and several constraints in different forms. What this unit purports to do is to take something, ?a, and remove it from someplace, ?b. In order to do so it must locate some disposal site, ?site, and look at the things already there, ?set. There are several kinds of constraints here: the constraints on CONSTRAINTS and the soft (influential) constraints on SOFT-CONSTRAINTS. The hard constraints demand that the variables, ?a, ?b, and ?set be bound by the match, and the soft constraint says that ?site should be bound, but not necessarily.

The pairing, using V_1, is done with isolated matches between entries, and at the end, the matches are re-done from top to bottom (left to right) to verify that a consistent matching has been made. This double

matching is unfortunate, but unavoidable.[12]

The meaning of the goals is that a disposal site needs to be found that contains nothing that will be confused with ?a, if that is a concern. The BOUNDP constraint is so that a site will be selected. Thus the goals will do some 'problem-solving' albeit on a primitive level. Nevertheless, the important feature, the numeric weighting, has not been discussed yet.

Initially there is an empty confidence entry, $(0 \quad . \quad 0)$, which will represent the score of the match, and amounts are added to it to determine the rating for this pairing of goals in the unit with entries in the system goals. The computation is quite simple and is based to a large extent on the method in [Shortliffe 1974] and [Tracy 1979].

6.7.2 Notation

First, some notation and operations are defined.

A *score* is of the form:

$$(m_b \quad . \quad m_d)$$

as in the case of the entries in the situation description. In order to evaluate a pairing associated with a unit and a situation description, one starts out with a zero score, $(0 \quad . \quad 0)$, which is then modified. The basic modification formula is:

$$\begin{aligned} \text{Ac}(n, (m_b \quad . \quad m_d)) = \text{If} \quad & n \geq 0 \\ & \text{Then} \quad (m_b + \frac{(1000 - m_b)n}{1000} \quad . \quad m_d) \\ & \text{Else} \quad (m_b \quad . \quad m_d + \frac{(1000 + m_d)n}{1000}) \end{aligned} \qquad (1)$$

The motivation is that if n is the measure of the change that some goal has on the belief or the disbelief in a score, then this formula increases (or decreases) the belief (or disbelief) by the percentage, $\frac{n}{10}$, of the distance from where the measure of belief (or disbelief) to certainty. So, suppose the score is $(600 \quad . \quad -200)$, and n is 500, then the new score is given by:

$$\text{Ac}(500 \quad , \quad (600 \quad . \quad -200)) = (800 \quad . \quad -200)$$

because 200 is half the distance (50%) from 600 to 1000. If n were -500 in the above case, the score would be $(600 \quad . \quad -600)$.

Now define the *confidence level* of a score as:

$$\text{Cf}((m_b \quad . \quad m_d)) = m_b + m_d \qquad (2)$$

Some other operations are:

$$\text{Bm}(x, y) = \frac{xy}{1000} \qquad (3)$$

Standing for Belief-Modify, this treats x as a percentage and takes that percentage of y.

[12] Even in the V_2 case this is done because the match variables need to be set, and the pairing functions are not assumed in general to perform consistent pairings on this level. V_2 could skip this step by using some flags, but the loss of efficiency checking for this is possibly not worth the effort.

$$\text{Ta}(x) = min(1000, \text{acceptance-level} - x) \qquad (4)$$

Standing for To-Accomplish, this computes how much work is needed to accomplish something with measure x.

$$\text{Chop}(x) = max(-1000, min(1000, x)) \qquad (5)$$

This ensures that $-1000 \leq x \leq 1000$.

$$\text{Bm3}(x, y, z) = \text{Chop}(\text{Bm}(z, \text{Bm}(x, y))) \qquad (6)$$

This extends Bm to three values.

$$\text{Nn}(x) = min(1000, 1000 - x) \qquad (7)$$

Standing for Numeric-Not, this computes the amount of work needed to get something with measure x up to total certainty.

$$\text{Nnn}(x) = min(1000, 1000 + x) \qquad (8)$$

Standing for Negative-Numeric-Not, this computes the amount of work needed to get something with measure x up to total uncertainty.

6.7.3 Computing the Evaluation for a Pairing

Suppose that there is a pairing of entries in the GOALS slot of the unit and items in the GOAL-LIST slot of the situation description, $(\ldots (u_i, sd_i) \ldots)$, and

$$\begin{aligned} u_i &= (\texttt{<pattern1>} \quad . \quad c), \\ sd_i &= (\texttt{<pattern2>} \quad (d1 \quad . \quad d2) \quad (ac1 \quad . \quad ac2) \ldots) \end{aligned}$$

and <pattern1> matches <pattern2>. Let Des be

$$\text{Cf}(\texttt{<desirability>})$$

and Acc be

$$\text{Cf}(\texttt{<accomplishment>}).$$

If $Acc > 0$ then the score is updated with:

$$\text{Bm3}(c, \text{Nn}(Acc), Des)$$

otherwise it is:

$$\text{Bm3}(c, min(1000, Acc + 1000), Des)$$

In this second case, $Acc + 1000$ is the difference between the absolute minimum for Acc, which is -1000 and Acc; this function is simply Nn with -1000 substituted for 1000. This is the amount of work needed to dis-accomplish this goal, which is what Des indicates is supposed to happen;

Additionally, let PATTERN1 be the result of substituting the values of pattern variables for those pattern variables in <pattern1>, and let

$$U = \{(p \quad . \quad n) \in \text{COUNTERGOALS} \quad | \quad (\text{umatch} \quad \text{PATTERN1} \quad p)\}$$

which simply collects all of the items in COUNTERGOALS with first element matching <pattern1>. Then the score is modified by:

$$\forall (p \quad . \quad n) \in U \quad \text{do} \quad \text{Ac}(n, score) \qquad (9)$$

This has the effect of adding in all of the countergoal influences onto the score.

The above computation is done for every pair (u_i, sd_i) in the pairing $(\ldots (u_i, sd_i) \ldots)$.

6.7.4 Pre-conditions

Next the influence of the pre-conditions are added. For each pre-condition, $(pat \quad . \quad c)$ in PRE-CONDITIONS, let

$$U = \{(p \quad Des \quad Acc) \in \text{SYSTEM-GOALS} \quad | \quad (\text{umatch} \quad \texttt{pat} \quad p)\}$$

with both Des and Acc in the familiar format, $(m_b \quad . \quad m_d)$. Then, for each element in U perform: If $c > 0$ then

$$\text{Ac}(\text{Bm}(-c, max(0, \text{Ta}(\text{Cf}(Acc)))), score) \qquad (10)$$

And if $c \leq 0$ then

$$\text{Ac}(\text{Bm}(c, max(0, \text{Ta}(\text{Cf}(Acc)))), score) \qquad (11)$$

Thus, pre-conditions add negatively proportional to how much work needs to be done to accomplish them.

6.7.5 Added-goals

Added goals are the goals that are added to the situation description in the case that the unit is invoked.

For each added-goal, $(pat \quad . \quad c)$ in ADDED-GOALS, let

$$U = \{(p \quad Des \quad Acc) \in \text{SYSTEM-GOALS} \quad | \quad (\text{umatch} \quad \texttt{pat} \quad p)\}$$

with Des and Acc in the familiar format. Then, for each element in U perform: If $Des < 0$ and $c < 0$ or if $Des > 0$ and $c > 0$ then the score is unaffected. This is because added goals are simply descriptors that are added to the situation description by the actions of the unit described, and so there is no added advantage for a unit to add further desire to something already desired (or subtract from something undesired), but there is a negative value associated with trying to add conflicting goals. If $Des > 0$ the score is updated by:

$$\text{Ac}(\text{Bm3}(c, \text{Nn}(\text{Cf}(Acc)), Des), score) \qquad (12)$$

Otherwise it is given by:

$$\text{Ac}(\text{Bm3}(c, \text{Nnn}(\text{Cf}(Acc)), Des), score) \qquad (13)$$

Again, adding a goal that conflicts with the interests of the system reduces the score.

6.7.6 System Influences

The INFLUENCES slot on the SYSTEM-GOALS unit contains a list of the same format as the goals on the GOALS slot of the unit being measured; that is, $(\ldots(\texttt{<pattern>} \quad . \quad amount)\ldots)$ Let UINFLUENCES be the entries on the INFLUENCES slot on the unit in question, then for each influence, $(pat \quad . \quad c)$, let

$$U = \{(p \quad . \quad n) \in \text{GOALS} \cup \text{ADDED-GOALS} \cup \text{UINFLUENCES} \quad | \quad (\text{umatch} \quad \texttt{pat} \quad p)\}$$

Then for all elements in U the score is modified by:

$$\text{Ac}(\text{Bm}(c, n), score) \qquad (14)$$

6.7.7 Unit Influences

The unit influences interact with the GOAL-LIST and INFLUENCES slots on the SYSTEM-GOALS unit. These influences are the same format as the system influences. For each influence, $(pat \quad . \quad c)$, let

$$U = \{(p \quad . \quad n) \in \text{INFLUENCES} \quad | \quad (\text{umatch} \quad \texttt{pat} \quad p)\}$$

For each element in U the score is modified by:

$$\text{Ac}(\text{Bm}(c, n), score) \qquad (15)$$

Additionally, for each influence, (pat, c), let

$$U = \{(p \quad Des \quad Acc) \in \text{SYSTEM-GOALS} \quad | \quad (\text{umatch} \quad \texttt{pat} \quad p)\}$$

For each element of U the score is modified by:

$$\text{Ac}(\text{Bm}(c, \text{Cf}(Des)), score) \qquad (16)$$

This ends the modification by objects that involve pattern matching, the rest are predicates that are evaluated in order to obtain the enabling condition for modification.

6.7.8 Preferences

Each unit can have a list of preferences, each of the form $(name \quad predicate \quad amount)$. For each such, the score is modified if the predicate evaluates to non-NIL by:

$$\text{Ac}(amount, score) \qquad (17)$$

In addition, there is a queue of the form

$$(\ldots((unit \quad name) \quad factor \quad threshold)\ldots)$$

such that at regular intervals, if $unit$ contains a preference named $name$, if $\langle amount \rangle < \langle threshold \rangle$ then that preference is deleted. Otherwise it is replaced by:

$$(name \quad predicate \quad \text{Ac}(amount, \text{Bm}(factor, amount)))$$

6.7.9 Soft Constraints

Both the unit and the system goal unit have soft-constraint entries. Let U_{sg} be the entries on the SOFT-CONSTRAINTS slot on SYSTEM-GOALS and let U_u be the entries on the SOFT-CONSTRAINTS slot on the unit in question. Then let $U = U_{sg} \cup U_u$. For each element, $(predicate \quad . \quad amount) \in U$, if the predicate evaluates to non-NIL, the score is modified by:

$$\text{Ac}(amount, score) \qquad (19)$$

6.7.10 Level Bonus

Finally, the level of the unit affects the score by the amount `-level-bonus-`, which is typically small:

$$\text{Ac}(\text{-level-bonus-}, score) \qquad (20)$$

6.7.11 Exceptions

As mentioned earlier (Chapter 3) the measures for all of the entries on the unit can be functional rather than constant. In these cases, there is a function rather than a number, and the evaluation is exactly as above, but the function is EVALed rather than the variable being referenced. So, expression 14 becomes:

$$\text{Ac}(\text{Bm}(c, \text{EVAL}(n)), score) \qquad (14')$$

for instance.

6.8 The Full Process

The full selection process starts out with a SYSTEM-GOALS unit and a number of units *in situ*. The basic idea is to obtain all units and pairings that match the situation description, choose the 'best' unit and pairing, and then to invoke that unit with those pairings. In the section on sequencing, it was noted that one could then create a situation description and repeatedly *react* to it by invoking best matching units until some condition of success or diminishment occurs.

The PURPOSE slot on a unit contains entries that are used to index the community of units. Suppose the situation description contains the entry:

```
((on blocka blockb) (700 . 120)(0 . 0) bazola)
```

which states that the goal `(on blocka blockb)` ought to be done at some point, the key indexing information is the pattern, `(on blocka blockb)`. Suppose a unit has an entry on its GOALS slot which is:

```
((on ?a ?b) . 723)
```

then it is customary for the PURPOSE slot to contain `(on ? ?)`, which says that this unit is appropriate to evaluate in this case.

Of course, there is no demand to list everything in the GOALS slot in the PURPOSE slot, and also none to not list other things as well. The idea behind not listing an entry is that that entry is not important or occurs in many unrelated contexts. The reasoning behind listing other things is that an entry in SYSTEM-GOALS may be used to trigger an otherwise common description. I.e., if a description of a unit would match many situations, a PURPOSE entry can be used to index that match in only specific cases.

Once a list of units is produced, the selection process will then produce an evaluation for each pairing of the entries in the situation description and the unit's description according to the above described process. Recall the example in an earlier section of this chapter in which the unit description was:

```
(((on <blockp ?a><blockp ?b>) . 500)
 ((on <blockp ?b><surfacep ?c>) . 500))
```

and the situation description was:

```
(((on blocka blockb) (600 . 0)(0 . 0)...)
 ((on blockx blocky) (500 . 0)(0 . 0)...)
 ((on blockb table)  (700 . 0)(0 . 0))...)
 ((on blocky table)  (400 . 0)(0 . 0)...)...)
```

this time phrased as a goal rather than as a fact.

The result of evaluating this unit description against the situation description is a measure for each pairing that is possible. This evaluation is:

```
(((545 . 0)
  (((((on ?a ?b) . 500)
      (on blocka blockb) (600 . 0) (0 . 0) nil)
    (((on ?b ?c) . 500)
      (on blockb table)  (700 . 0) (0 . 0) nil))
   ((?a . blocka) (?b . blockb) (?c . table))))
 ((400 . 0)
  (((((on ?a ?b) . 500)
      (on blockx blocky) (500 . 0) (0 . 0) nil)
    (((on ?b ?c) . 500)
      (on blocky table)  (400 . 0) (0 . 0) nil))
   ((?a . blockx) (?b . blocky) (?c . table)))))
```

There are two possible pairings of the unit description and the situation description, resulting in the two measures, 545 and 400. The pairings that gives rise to these evaluations are also given.

Returning to the general selection procedure, for each unit a list of such evaluations is returned, and the entire list will be called P, for *possibilities*. This list is then filtered for all entries, $(ev \quad . \quad \texttt{<pairings>}) \in P$ such that Cf$(ev) >$ `superior-level`. If this set is non-empty, there are a number of superior choices, and the best of them is selected.

Each list of possibilities is initially sorted according to the following obscure predicate, Cf_g, which, if given two 5-tuples, $S_1 = (u_1, p_1, e_1, l_1, n_1)$ and $S_2 = (u_1, p_2, e_2, l_2, n_2)$, where u_i is the unit name, p_i is a pairing, e_i is its evaluation, l_i is the level (LEVEL slot) of u_i, and n_i is the number of pairs in p_i, decides:

$$\begin{aligned} \text{If } \texttt{acceptance-level} < e_1 \quad & \& \\ \texttt{acceptance-level} < e_2 \quad & \& \\ \langle e_1 - e_2 \rangle < \texttt{equal-level}(= 75.) & \end{aligned} \tag{21}$$

then:

$$n_1 < n_2 \Longrightarrow S_1 < S_2;$$
$$n_1 > n_2 \Longrightarrow S_1 > S_2;$$

If $n_1 = n_2$ then:

$$l_1 < l_2 \Longrightarrow S_1 < S_2;$$
$$l_1 > l_2 \Longrightarrow S_1 > S_2;$$
$$l_1 = l_2 \Longrightarrow [e_1 < e_2 \Longleftrightarrow S_1 < S_2]$$

Otherwise:

$$e_1 < e_2 \Longrightarrow S_1 < S_2;$$
$$e_1 > e_2 \Longrightarrow S_1 > S_2;$$
$$e_1 = e_2 \Longrightarrow [n_1 < n_2 \Longleftrightarrow S_1 < S_2]$$

This odd predicate has a fairly straightforward meaning which renders it easy to understand (!). If both e_1 and e_2 are acceptable, and if they are within the equality tolerance, then other factors dominate the decision in the order: number of pairs in the pairing, the level number (the higher, the better), and finally the evaluation. If they are not close (the normal case), or not acceptable, the evaluation dominates, unless they are equal, when the number of pairs dominates. This is the *value preference ordering*.

There is a second relation that is sometimes used, called *variety ordering*.

In the case of variety preference, the predicate, Cf_{g_1}, takes two 6-tuples $S_1 = (u_1, p_1, e_1, l_1, n_1, o_1)$ and $S_2 = (u_1, p_2, e_2, l_2, n_2, o_2)$, where everything is as above, but o_i is the number of occurrences of u_i in the plan list (the PLAN-LIST slot on SYSTEM-GOALS).

$$\begin{aligned} \text{If } \texttt{acceptance-level} < e_1 \quad & \& \\ \texttt{acceptance-level} < e_2 \quad & \& \\ \langle e_1 - e_2 \rangle < \texttt{equal-level}(= 150.) & \end{aligned} \tag{22}$$

then:

$$n_1 < n_2 \Longrightarrow S_1 < S_2;$$
$$n_1 > n_2 \Longrightarrow S_1 > S_2;$$

If $n_1 = n_2$ then:

$$o_1 < o_2 \Longrightarrow S_1 < S_2;$$
$$o_1 > o_2 \Longrightarrow S_1 > S_2;$$

If $o_1 = o_2$ then:

$$l_1 < l_2 \Longrightarrow S_1 < S_2;$$
$$l_1 > l_2 \Longrightarrow S_1 > S_2;$$
$$l_1 = l_2 \Longrightarrow [e_1 < e_2 \Longleftrightarrow S_1 < S_2]$$

Otherwise:

$$e_1 < e_2 \Longrightarrow S_1 < S_2;$$
$$e_1 > e_2 \Longrightarrow S_1 > S_2;$$
$$e_1 = e_2 \Longrightarrow [n_1 < n_2 \Longleftrightarrow S_1 < S_2]$$

If there are no unavoidable (evaluation $>$ `superior-level`) ones, the list is searched for evaluations that are higher than another threshold, `acceptance-level`, in which case the best one is returned. There is an option in this process which allows for variety of unit choice to have some precedence over evaluation. In this option, this last step is preceded by one in which the units which do not display more variety than the constant, `acceptable-variety` are eliminated.

6.8.1 What If Nothing is Selected by Now?

The process of selecting a unit and a pairing is not defeated if nothing is selected up to this point. The next step, however, is highly counterintuitive and Chapter 7 discusses it. There will be a fair amount of justifications given for the next step, and this chapter is already somewhat lengthy and not complete.

6.9 Anomalies and Examples

There is yet another wrinkle in this complex story, the COMPLEX, which is used to group entries on the system goals unit. A COMPLEX is simply an alternative entry on SYSTEM-GOALS, and is of the form:

```
(COMPLEX (<pattern> <des> <acc>...)...)
```

where the CDR of the entry is a list of normal entries on SYSTEM-GOALS. The complex acts entirely transparently with respect to the extra grouping, except that units can only be indexed into based on the first entry in the complex. Recall that the units are indexed by the pattern parts of the entries of the situation description. For a complex, the only pattern used is the pattern of the first entry in the complex.[13]

[13] An obvious use for complexes is to be able to have all the things in them go together somehow. So, for instance, if some unit used some things from a complex, it should use everything from it that it can. This turned out to be hard to do and not that useful anyway, since in order to make some things available means to make everything available from a complex, and what if you *need* something from another complex.... Anyway, there are several much better ways (easier to express, faster to run) of doing what you want, and that will be the topic of the next section of this interminable chapter.

Here is an example of the syntax and some interesting features of the language:

```
{system-goals
 level 1
 soft-constraints ((ztesch . 900))
 goal-list
 (((floor a) (0 . 0)(900 . 0) ())
  (complex ((on a b) (500 . 0)(0 . 0) ())
           ((on b c) (500 . 0)(0 . 0) ())
           ((on c d) (500 . 0)(0 . 0) ()))
  ((on d e) (500 . 0)(0 . 0) ())
  (complex ((on e f) (500 . 0)(0 . 0) ())
           ((on f g) (500 . 0)(0 . 0) ())
           ((on g h) (500 . 0)(0 . 0) ()))
  ((red h) (0 . 0)(900 . 0)()))
 plan-list (())
 meta meta-s-g}
```

Notice the soft constraint with predicate ZTESCH, which is just a variable evaluation.
Here is a unit that will be matched against the situation description above:

```
{meta-bazola
 level 1
 bindings (?a ?b ?c ?d ?e)
 pairing-function structured-full-pairs
 purpose ((fool-around ?))
 influences (((on a b) . (baz)))
```

where `structured-full-pairs` is the internal name for the V_2 pairing function.

```
 soft-constraints
 (((member-accomplished-goal `(floor ,?a)) . (floor))
  ((member-accomplished-goal `(red ,?e)) . (red)))
```

where `member-accomplished-goal` tests whether its argument is the pattern of an entry in the system goals unit with an accomplishment measure greater than `acceptance-level`. `floor` and `red` are functions that return integers.

```
 goals (((on ?a ?b) . 700)
        ((on =?b ?c) . 700)
        ((on =?c ?d) . 800)
        ((on =?d ?e) . 800))
```

The syntax, '=?b' means to use the LISP value of the variable, '?b,' which is set by the pairing of the entry, `(on ?a ?b)`, on the previous line (remember that V_2 matches in order). =?b is used in order speed up the match, by V_2, and is part of the UMATCH syntax.

```
 constraints
 ((bazola t (all-bindings-boundp 'meta-bazola)))}
```

This says that the variables in the slot, BINDINGS, for this unit, named `meta-bazola`, must be bound by the pairing process.

```
(setq -functional-measure- t)
(defun floor () 200)
(defun red () 100)
(defun baz () 0)
(setq ztesch ())
```

The above tell the system to expect functions in measure locations of entries on the unit, define the simple functions, and set the variable for the soft constraint. The following is the result of the evaluation of this unit on that situation description, with the pairings noted:

```
(((862 . 0)
  (((((ON ?A ?B) . 700) (ON D E) (500 . 0) (0 . 0) NIL)
    (((ON =?B ?C) . 700) (ON E F) (500 . 0) (0 . 0) NIL)
    (((ON =?C ?D) . 800) (ON F G) (500 . 0) (0 . 0) NIL)
    (((ON =?D ?E) . 800) (ON G H) (500 . 0) (0 . 0) NIL))
   ((?A . D) (?B . E) (?C . F) (?D . G) (?E . H)))
  4)
```

Notice that the rating is slightly higher because of the soft constraint concerning the color red.

```
((847 . 0)
 (((((ON ?A ?B) . 700) (ON C D) (500 . 0) (0 . 0) NIL)
   (((ON =?B ?C) . 700) (ON D E) (500 . 0) (0 . 0) NIL)
   (((ON =?C ?D) . 800) (ON E F) (500 . 0) (0 . 0) NIL)
   (((ON =?D ?E) . 800) (ON F G) (500 . 0) (0 . 0) NIL))
  ((?A . C) (?B . D) (?C . E) (?D . F) (?E . G)))
 4)

((847 . 0)
 (((((ON ?A ?B) . 700) (ON B C) (500 . 0) (0 . 0) NIL)
   (((ON =?B ?C) . 700) (ON C D) (500 . 0) (0 . 0) NIL)
   (((ON =?C ?D) . 800) (ON D E) (500 . 0) (0 . 0) NIL)
   (((ON =?D ?E) . 800) (ON E F) (500 . 0) (0 . 0) NIL))
  ((?A . B) (?B . C) (?C . D) (?D . E) (?E . F)))
 4)

((877 . 0)
 (((((ON ?A ?B) . 700) (ON A B) (500 . 0) (0 . 0) NIL)
   (((ON =?B ?C) . 700) (ON B C) (500 . 0) (0 . 0) NIL)
   (((ON =?C ?D) . 800) (ON C D) (500 . 0) (0 . 0) NIL)
   (((ON =?D ?E) . 800) (ON D E) (500 . 0) (0 . 0) NIL))
  ((?A . A) (?B . B) (?C . C) (?D . D) (?E . E)))
 4))
```

And this last pairing has a higher rating because A is the floor.

6.9.1 Non-Constant Patterns

In the units presented so far, the GOALS slot has been constant. There would seem to be a need for being able to specify a schemata for the entries. This, it turns out, is an option that the user specified pairing function can do. Such a pairing function was written to demonstrate this capability. It is call

structured-full-pairs-source, and is a variant on V_2. If an entry looks like, (source <pattern>), it generates as many patterns based on <pattern> as the traffic can bear. An example is best here:

```
{system-goals
level 1
goal-list (((on a b) (500 . 0)(0 . 0) ())
           ((on b c) (500 . 0)(0 . 0) ())
           ((on c d) (500 . 0)(0 . 0) ())
           ((on d e) (500 . 0)(0 . 0) ()))

plan-list (())
meta meta-s-g}
```

This unit represents that it is desired to create a stack of things. If there was a unit that could build any stack of things, no matter how tall, the description of that unit might look like:

```
{baz
 level 1
 bindings (?a ?b)
 pairing-function structured-full-pairs-source
 purpose ((fool-around ?))

 goals (((on ?a ?b). 700)
        ((source (on =?b ?b)) . 700))
 constraints ((bazola t (all-bindings-boundp 'baz)))}
```

The notation (source (on =?b ?b)) is a a generator of patterns, which generates the sequence, (on =?b ?b1), (on =?b1 ?b2), (on =?b2 ?b3),... What this description will do is to select a pattern of the form (on ?a ?b) and then locate the highest tower above it. So, for each of the four possible ways of selecting items from the situation description, there is a pairing. The pairings and evaluations produced by this are:

```
(((353 . 0)
  (((((ON ?A ?B) . 700) (ON D E) (500 . 0) (0 . 0) NIL))
   ((?A . D) (?B . E)))
  1)

 ((579 . 0)
  (((((ON ?A ?B) . 700) (ON C D) (500 . 0) (0 . 0) NIL)
    (((ON =?B ?B1) . 700) (ON D E) (500 . 0) (0 . 0) NIL))
   ((?A . C) (?B . D)))
  2)

 ((726 . 0)
  (((((ON ?A ?B) . 700) (ON B C) (500 . 0) (0 . 0) NIL)
    (((ON =?B ?B1) . 700) (ON C D) (500 . 0) (0 . 0) NIL)
    (((ON =?B1 ?B2) . 700) (ON D E) (500 . 0) (0 . 0) NIL))
   ((?A . B) (?B . C)))
  3)
```

```
((821 . 0)
 (((((ON ?A ?B) . 700) (ON A B) (500 . 0) (0 . 0) NIL)
   (((ON =?B ?B1) . 700) (ON B C) (500 . 0) (0 . 0) NIL)
   (((ON =?B1 ?B2) . 700) (ON C D) (500 . 0) (0 . 0) NIL)
   (((ON =?B2 ?B3) . 700) (ON D E) (500 . 0) (0 . 0) NIL))
  ((?A . A) (?B . B)))
 4))
```

It is also possible to vary the measure of strength in the patterns generated, so that one can, for instance, make large towers less preferable.

6.9.2 The Other Variants

There are a number of other situations in which this basic matching and/or selection framework is used. The main variant is using the accomplishment measure rather than the desirability measure for matching, which has the effect of matching situations rather than capabilities and desires.

All other variants are matching objects that are not covered exactly in the framework so far, such as situation descriptions against other situation descriptions and units against units. Invariably these are done by turning a situation description into a unit description or vice versa.

The following is an example of matching two situations to determine if one reminds Yh of the other:

```
{system-goals
 level 1
 goal-list
 (((bright green female iguana) (0 . 0)(700 . 0)())
  ((dull green female iguana) (0 . 0) (400 . 0)())
  ((large metal cattle prod) (0 . 0)(800 . 0)())
  ((startled room service clerk)(0 . 0)(900 . 0)())
  ((screaming women)(0 . 0)(250 . 0)())
  ((throbbing headache)(0 . 0)(650 . 0)())
  ((dull pensive mood) (0 . 0)(270 . 0)()))
 influences ((pensive . 150))}

{system-goals2
 level 1
 pairing-function structured-full-pairs-source
 goal-list
 (
  ((startled room service clerk)(0 . 0)(300 . 0)())
  ((screaming women)(0 . 0)(650 . 0)())
  ((throbbing headache)(0 . 0)(650 . 0)())
  ((dull pensive mood) (0 . 0)(970 . 0)()))
 influences ((pensive . 150))}
```

```
(((739 . 0)
  (((((STARTLED ROOM SERVICE CLERK) . 300)
     (STARTLED ROOM SERVICE CLERK)
     (0 . 0)
     (900 . 0)
     NIL)
    (((SCREAMING WOMEN) . 650)
     (SCREAMING WOMEN)
     (0 . 0)
     (250 . 0)
     NIL)
    (((THROBBING HEADACHE) . 650)
     (THROBBING HEADACHE)
     (0 . 0)
     (650 . 0)
     NIL)
    (((DULL PENSIVE MOOD) . 970)
     (DULL PENSIVE MOOD)
     (0 . 0)
     (270 . 0)
     NIL))
   NIL)
  4))
```

6.10 Special Control Structures

The sequencing in Yh is either by agenda, by the reactive component (via the agenda), or by standard control structures. There is a need for a way to invoke units using the reactive component, but from standard control structures rather than through the agenda.

The interesting control activity at this point is to be able to say, "here is the description that I want to cause a change in control." Thus, this mechanism is 'invoke by specific description,' but the desirable thing is to be able to have whatever is on the system goals unit at the time influence the decision about what to call.

So, by a simple function call it is possible to invoke the entire reactive component. The main argument to the function will be a description of the situation desired. The mechanism of the function call is to make everything that is a normal entry on the GOALS slot of SYSTEM-GOALS become an influence and to call the unit selection process with a system goals unit where GOALS is provided by the description in the function call. There are several basic routines to do this operation, but they will not all be understandable until something about the nature of invoking a unit via the matching process is explained.

6.10.1 Passing Pairs as Arguments

The METHOD slot in a unit can have the equivalent of a LAMBDA expression for receiving parameters from the calling routine. Units which are expected to be called by this process normally have a lambda variable, called PAIRS for consistency, which receives the pairs that caused the successful evaluation of that unit. The first operation done in the unit is normally:

```
(set-bindings-from-pairs pairs)
```

which simply causes the bindings in the BINDINGS slot to be bound to what they matched in the unit selection process. So, if the unit were:

```
{bazola
  bindings (?a ?b)
  goals (((on ?a ?b) . 760))
  purpose ((on ? ?))
  method (lambda (pairs)
          (set-bindings-from-pairs pairs)
          (print (list ?a ?b))...)}
```

and the system goals contained:

```
...
((on blocka blockb) (500 . 0)(0 . 0)())
((on blockg blockq) (500 . 0)(0 . 0)())
...
```

then when the PRINT is performed, ?a is bound to `blockg` and ?b to `blockq`.

6.10.2 Optimal Calling of a Set of Goals

One can call a set of goals from any point in the normal execution, which causes a pseudo-SYSTEM-GOALS to be established. This is done by taking the set of entries (descriptors with associated desirabilities and accomplishments, etc) and converting them to influences, adding these to the INFLUENCES slot on the system goals unit. Then the descriptors passed to this function are converted to entries by providing $(1000 \quad . \quad 0)$ as the desirability for each descriptor and $(0 \quad . \quad 0)$ as the accomplishment. If each original entry in SYSTEM-GOALS is of the form:

```
(<pattern> <desirability> <accomplishment>
           <comment><minor action> .
           <more comments>)
```

then the corresponding influence, as created, is:

$$(<pattern> \quad . \quad \text{Cf}(\text{Ac}(\text{Cf}(\texttt{<desirability>}, \texttt{<accomplishment>})))) \qquad (23)$$

Here, a *goal* is simply the pattern part of a full entry on the SYSTEM-GOALS unit.

The full selection process is called on that constructed situation description, in the manner described above.

6.10.3 Optimal Calling of a Set of Entries

This is similar to the previous case, except that the full entries are supplied, not just the pattern descriptors.

An orthogonal set of options is whether the modified (constructed) situation description or the set of goals or entries is added to the current situation description unit. In one case, there is a change made to the SYSTEM-GOALS unit, and in the other, no change is made, but a selection of unit is made and an invocation done on a situation description that may not have caused that invocation.

What these are used for is to re-direct the flow of control, but influenced by the current situation description. Often, for example, one wants to insert a noun phrase or some adjectives at that point, but with the 'state of mind' that is current still in force.[14]

[14]This was the original intention of the COMPLEX, but this method turned out to be easier to implement and more often was in a form that reflected the desired semantics—it was more transparent.

6.10.4 Example

Suppose there was a situation description about the colors of some blocks on a table, plus some requirements about things to build as well as things that have been built. In order to put BLOCKA on BLOCKB on BLOCKC right now, one would say:

```
(optimal-call-goals
 '((on blocka blockb)
   (on blockb blockc)))
```

6.10.5 More Control

One can also specify the PURPOSE indices which will be used to select the unit to send control to. Normally, the patterns in the goals or entries determine automatically the patterns that index the units that are then evaluated. If specific patterns are supplied (as the last argument) then these are used in place of the default ones. This way, if there are goals or entries that are irrelevant in terms of selecting units, their patterns may be left out and they will act as secondary goals only. This is often used in certain word selection processes where the set of words that are chosen for consideration come from this index set, and then the decision is made on the basis of some other description, namely the one supplied to the function.

6.11 How is This a Good Idea?

There is some question about how and when this is a good idea. The whole idea of influences and tendencies is an attempt at making programming this system more like psychiatry than brain surgery. By this is meant that modifying a large system in standard programming languages involves changing the insides of the system. With this type of system it is conceivable that by changing the descriptions of things one can achieve the desired results.

By making the matching system sensitive to slight variations one hopes to achieve control by suggestion and not so much by demand. However, by putting variations in a situation description and having a sophisticated matcher, one cannot expect great results unless the unit descriptions are finely attuned and richly varied themselves. That is, unless the other parties in the matching process are sensitive to the nuances, there is no possibility for interesting behavior, and the mechanism is wasted.

Thus, there is an inherent interest in making every description in the system as full as possible and to not hesitate to multiply describe things, to have many ways of doing the same thing, etc, so that a refined selection can be made.

Invocation by matching of this type allows one to describe the task, and even the hoped-for method so that a good knowledge of the exact nature of the system is irrelevant.

Furthermore, by adding sensitivity to the system in the way that has been done, not only are choices of units affected, but the binding mechanism becomes a problem solving aid itself, its sensitivity also being increased. So the result is that the binding mechanism becomes more responsive to the situation, and interesting behavior is expected.

Chapter 7

Counterinduction

Simulated Annealing

Strange as it might seem, when I was working on Yh I didn't know about randomized algorithms and metaheuristics. In particular, I had never heard of *simulated annealing*. I did come across counterinduction while reading Feyerabend, and it could be said that my interpretation of counterinduction as a sort of resource-allocation strategy was a step on the path to (re)discovering ideas like simulated annealing.

Later around the early 2000s I discovered simulated annealing and made a hobby of seeing which puzzles (like Sudoku) could be solved using it. I created a framework to make these hobby horses easy to code up; and later, InkWell made use of simulated annealing (and genetic algorithms and various simple forms of machine learning). I attribute my fondness for these sorts of metaheuristics to my partnership with counterinduction.

In the previous chapter a selection process was presented for choosing a unit to invoke given a situation description. At the end of the process described it could be the case that nothing had yet been chosen. This means that no unit had yet had an evaluation better than `acceptance-level`, which is normally 200.

It must be kept in mind, though, that the normal method of evaluation is an estimate of how applicable the unit is or to what extent its capabilities match the needs of the system as reflected by the SYSTEM-GOALS unit's description of that. In some ways, a better evaluation method is to actually perform the actions and see how much improvement takes place.

Therefore, the fact that no unit displays a high enough measure may not mean that no unit is good enough, but that the estimation function has broken down—maybe. So, one might expect that a reasonable next course is to try some other evaluation method. The method is called *counterinduction* and has several interesting features, demanding a lot of justification for its existence.

7.1 The Situation

At this point the system is within the reactive component, attempting to find a unit with a description that would be appropriate to invoke on some situation description. The evaluation of all units has fallen short of the minimum required to proceed, and the most obvious course is to abandon the search. In short, the situation assumed to be the case at this point is hopeless. Unless some unusual action is derived then the only alternative is to give up, which means abandoning the situation description, going to another agenda item, or letting the meta situation description attempt to figure out what to do.

Any means of finding an alternative is preferable to giving up. Recall that the evaluation in the previous chapter is just an estimate of the effects of the unit being invoked. Moreover, the description in that unit may not be entirely accurate.

One thing that can be tried is to take a number of the candidates—the units that claim to have something to do with the situation description—and to do a search based on performing them. In this case, each unit's effects would somehow be simulated on the situation description, and some measure of that new situation would be made. This would be a different form of evaluation of a unit, and perhaps it would be a more accurate reflection of that unit's applicability. Then one could invoke some version of the selection process to see what could be done after this unit has been used. This is a traditional search, and it should probably be a breadth first search with limited branching factors and depth. The reason for that is that, after all, it is a bad situation and there is probably not much point in expending a large amount of time and computation in an expensive search.

In other places Winograd [Winograd 1980] has argued that a resource limited search (or match) is a good way to go about a number of things. To some degree, the matching process in the previous chapter is an example of that in that it, too, is an estimate of a complex process, much as a resource limited computation is in other contexts.

The descriptive system can be thought of as expressing in a uniform manner a monolithic version of the contents of the represented objects in the system, with the hybrid matching process as an approximation of some network traversing or searching procedure.

Resource limited searches simply provide a direction, rather than an answer, in an expensive environment. In no way are they an end-all solution, but just a foothold in an otherwise slippery and gigantic environment. It is nothing magic, and it is not a panacea, but it is something to try—since there is little point in spending too much time in a hopeless situation, a limited version is an obvious thing to do.

But a search of any kind is a way to find surprising results. This is true trivially since in any well understood domain, one can expect that a description matching process might very well be able to come up with most answers, since the questions are, to some extent, well expected. When a search is undertaken, though, this hints at deep trouble: an unexpected situation has occurred, and there is nothing better to do than to look at the alternatives and try to imagine what the alternatives would do to further the cause, or at least put the system in a place that it could get out of easily.

7.2 An Unexpected Consideration

Given that units have descriptions, though numerically measured, how can one expect that the system could try anything new outside of what is anticipated by the descriptions and the various complex ways of influencing the consideration of those descriptions? One cannot expect this, in general, without outside influence or without exploratory behavior. And exploratory behavior is exactly what a resource limited search is designed to be.

In some ways, it is fair to say that a set of descriptions about the abilities of the units represents the *theory* the system has about the world, since the way the system reacts to it is exactly a function of those descriptions, along with what those descriptions cause to happen when the system is perturbed by the outside world.

These descriptions are the way the world is interpreted, since the system cannot conceive easily anything that is not related in some descriptive way to what it knows about already. And its entire activity is dominated by these descriptions.

But this allows the problem that the descriptions can be inadequate, leading to puzzling situations—situations that cannot be confronted easily within the given framework. This is the situation assumed currently.

One can always give up, but this does not solve any problems that are confronted, except insofar as ignoring a problem is a solution in any context. To make progress, then, means to abandon the world view, since to not abandon it means to remain within it, and that way, progress is never made, since little changes.

So to progress within a theory of the world means occasionally giving up that theory, or postulating things that are outside its realm, or even contradictory to it. In an exact sense, any exploratory behavior on the part of a system—any searching—is an affirmation of that principle. In short, one is in the position of having every method that it known to be effective not working. To search is to say, "well, the conclusion that the system is stuck is wrong: assume it isn't stuck and that the action, x, which is known to not be good is used anyway; then what?"

7.3 A Minimal Violation of Rationality

The answer to the problem of what to do lies in the domain of the limited search, and, in particular, in the choice of resource allocation. Given a number of alternatives and evaluations of those alternatives, and given some budget of resources to expend on the search for which of these alternatives to try, the question becomes one of giving each alternative some amount of time to prove itself better than expected.

The obvious answer, when there are n alternatives that are to be tried, is to give each the same amount or to give the best one the most, etc. The problem is that this violates the newly formulated principle of letting exploration take its course when all else fails and its companion: no increase of knowledge is possible within a rigid framework.

When faced with the defeat of a system of thought or whatever, there is no point in partially defending that world view: an evaluation of the best alternatives so far leaves none acceptable. To then partially trust that evaluation *in this case* is not *trivially* justifiable.

The principle of *counterinduction* is, thus, adopted. Being a resource allocation strategy in the face of defeat, it allocates resources *inversely* proportional to the evaluation of the previous chapter. At the end of the resource limited search, the best unit is then selected.

An additional rationale is that if some unit and pairing comes in with a low initial evaluation, one expects that it will take more actions later on to make up for the bad score, hence that unit should be explored in greater detail. This is simply the reverse evaluation preference strategy.

7.4 The Exact Process

This section describes the exact process that is used. The exact situation is that there is a set of units, pairings, and evaluations, none of which is above `acceptance-level`. Denote that set by, $P = \{\ldots(\text{unit}, \text{pairing}, \text{eval})\ldots\}$.

First, this set is scanned for a default unit and pairing in case nothing turns out to be very good. This scan just grabs the best unit and pairing above a certain threshold (currently set at the no-op `acceptance-level`). Then the counterinduction process is started.

At this point it is worthwhile to note that there are two versions of this process, the *variety preference* and the *value preference*. Since the variety preference uses a poor measure of variety it will not be discussed except to say that the resource selection is different where noted and that the sorting predicate is Cf_{g_1} rather than Cf_g as defined in Chapter 6.

The set, P, is then skimmed to remove all but the first n, where n is a global amount called `-branching-factor-`, normally 3,[1] Then the resources are allocated as follows:

$$\text{Rs}(br, bf, ev) = \frac{br}{bf} \quad \frac{max(1000, 1000 - ev)}{1000} \qquad (1)$$

where br is `-basic-ration-` $= 6$, bf is the `-branching-factor-` $= 3$, and ev is the evaluation, $-1000 \leq ev \leq 1000$.

In the case of variety preference, the resources are given by:

$$\text{Rs1}(l, h, bf, br) = (\frac{br}{bf})(1 - \frac{h}{l}) \qquad (2)$$

where l is the length of the plan (history) so far and h is the number of occurrences of the unit in that plan (history).

The number returned by one or the other of these functions is the depth of the tree along the branch starting out with that unit and pairing. At no other point are resources allocated.

Next the small tree is expanded by simulating the actions of a unit using only the description of it. In some ways there is some danger involved in this assumption—that the description accurately reflects the actions of a unit—when the description is used in an advertisement style, which means that it is used to encourage or discourage use of a unit, not necessarily to be an accurate reflection on anything.

7.4.1 Rating the Situation Description

A situation description contains the goal-list and the influences, as given by the slots GOAL-LIST and INFLUENCES. Call these L and I, respectively.

The rating of a situation description is given by: If L is NIL, then the score is $(1000 \quad . \quad 0)$. Otherwise let Score $= (0 \quad . \quad 0)$ and m be the cardinality of U, where

$$U = \{\texttt{(<pat> <des> <acc>...)} \in \text{SYSTEM-GOALS} \quad | \quad \langle \text{Cf}(\texttt{<des>}) \rangle > \texttt{acceptance-level}\}$$

where `<des>` and `<acc>` are both of the form $(mb \quad . \quad .md)$.

Define:

$$\text{Sac}(m, (mb \quad . \quad md)) = \text{If} \quad m \geq 0 \quad \text{then} \quad (mb1 + m \quad . \quad md1) \quad \text{Otherwise} \quad (mb1 \quad . \quad md1 + m) \qquad (3)$$

Then the score is updated for elements of U, `(<pattern> <des> <acc>...)`, by:

$$\text{Sac}(\frac{\text{Bm}(\text{Cf}(\texttt{<des>}), \text{Cf}(\texttt{<acc>}))}{m}, score) \qquad (4)$$

where Bm is as defined in Chapter 6.

[1] Although many were explored, this seemed to give adequate performance without excessive runtime. The example shown later was run with many values for this and other parameters.

For the influences, I, the updating to $score$ proceeds for (<pat> . n) and (<pat> <des> <acc>...) with:

$$\text{Ac}(\text{Bm}(n, \text{Cf}(\texttt{<acc>}), score) \qquad (5)$$

This rating is, roughly, the average measure of accomplishment balanced against the average measure of desirability.

7.4.2 Example

Here is a simple situation description and its rating:

```
{system-goals
 goal-list ((baz (500 . 0)(500 . 0) ())
            (foo (750 . 0)(500 . 0)())
            (ztesch (0 . -500)(0 . -500)())
            (mumble (572 . -231)(782 . -23)()))
influences ((foo . -125))}
```

which has the rating $(281 \quad . \quad -62)$.

This process is called $\text{Rg}(L, I)$.

7.4.3 The Process Continues

Now that the daughters of the current (initial) node have been selected, the process runs by letting each sub-branch grow to that depth where each node may sprout only -branching-factor- daughters. Consider now what happens at each node in order to simulate and evaluate the actions of the unit and pairing for that node.

At each node there is given the 3-tuple, (U, L, P), where U is the unit that is selected to be simulated, L is the list of goals in the situation description (same as above notation), and P is the set of pairings. The system takes the pairings and looks at the unit's pre-conditions and added goals to simulate the action of the unit. The simulation is as follows.

7.4.4 The Simulation

Let Pr be the set of pre-conditions for the unit, from the slot PRE-CONDITIONS. $\forall p \in Pr$ the system checks if the pre-condition, p is true. Let n be the measure associated with p, then if $n > 0$, an entry in L must be found with an accomplishment field, <acc>, such that

$$\text{Cf}(\texttt{<acc>}) > \texttt{acceptance-level}.$$

If $n < 0$ then an entry with <acc> such that

$$\text{Cf}(\texttt{<acc>}) < -\texttt{acceptance-level}.$$

must be found. If there is no entry with a matching pattern, it is assumed the pre-condition is not satisfied.

If all pre-conditions are satisfied, then the added-goals are considered before the goals. For each added goal, (<pat> . n), the entry corresponding to <pat> is retrieved from the situation description, (<pat> <des> <acc>...). First <acc> is set to $(0 \quad . \quad 0)$. Then <acc> in the entry is set to $\text{Ac}(n, (0 \quad . \quad 0))$.

The reason that the field is initially zeroed is that the added goals represent what that entry will look like exactly rather than relatively. The goals represent relative changes.

For each goal, (<pat> . n), both the ones appearing in the pairing and those not, the corresponding entry, (<pat> <des> <acc>...), situation description is updated by Ac(n, <acc>).

If there are any unsatisfied pre-conditions, the search on that branch stops, otherwise the best -branching-factor- units and pairings are selected as daughters of this node. The resources are decremented for each downward arc in the tree, and when a node hits zero, it is terminated.

The ranking of the original parent node for the unit and pairing that started things off is then set to the rating, Rg, which was defined above.

7.4.5 Selecting the Winner

The unit, then, with the best rating of the searched nodes and of the originally chosen default is selected, but only if the finally chosen ranking is above acceptance-level. This cutoff is a global variable which can be set to any value, as can be superior-level, which was described in Chapter 6..

As can be seen at this point, it is never the case that a unit and pairing with a bad rating (below acceptance-level) is chosen. The method of searching, though, gives some precedence to units and pairings with poorer estimates. Since there are two different rating schemes used, counterinduction can be used to select an interesting path in a bad environment or it can simply provide an alternative rating that is slightly more realistic.

7.5 Tuning

The parameters -branching-factor- and -basic-ration- were experimentally set, at 3 and 6 resp. Note that the computations associated with simulating the actions of the system are quite high, so that wide branching and deep searching is prohibitive. Even at the current settings 10 seconds of KL-10 CPU time are required for the search. The example shown in the next section was repeated for values $3 \leq$-branching-factor-≤ 7 and $1 \leq$-basic-ration-≤ 10. Only for values $3 \leq$-basic-ration- did it succeed.

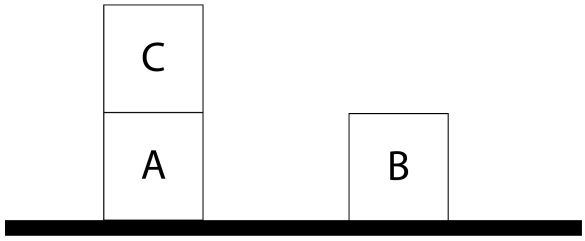
The timing mentioned above occured while running the generation example shown in the next chapter; it happened during adjective choice, in which Yh was considering whether or not an adjective should used instead of some other construction by proposing descriptions of paragraphs based on the two choices. The other construction was a second sentence with the adjective in the predicate adjective position.

7.6 An Example

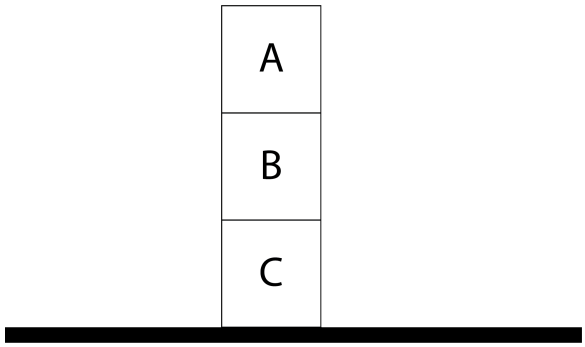
The following example is taken from Sussman's thesis [Sussman 1973] and is the famous *Anomalous Situation* which cannot be handled easily—or at least not naturally—in the HACKER framework. Since then many people have proposed planning systems that handle it easily [Sacerdoti 1977]. This example is not meant to show that Yh can handle the situation with ease, and, in fact, the description of the abilities of the simple robot will be wrong, intentionally.

The point, as usual, is not to claim that Yh solves this in any interesting way, but to demonstrate the mechanisms in a familiar setting and to show some of the tolerance the system has for bad descriptions.

The problem is simply stated a follows: there are three blocks on a table, labeled A, B, C, and TABLE. The original configuration is that C is on A and A is on the TABLE; B is on the TABLE. The goal state is A on B on C on the TABLE.



The goal state is:



The arm that is able to move things around can only move one block at a time, and that block must have a cleartop.

Here is the situation description:

```
{system-goals
 goal-list
 (
  ((cleartop b) (0 . -1000)(1000 . 0)())
  ((cleartop c) (0 . -1000)(1000 . 0)())
  ((cleartop table) (0 . 0)(1000 . 0)())
  ((cleartop a) (1000 . 0)(0 . -1000)())
  ((on a b) (1000 . 0)(0 . 0) ())
  ((on a table)(0 . 0)(1000 . 0) ())
  ((on b table)(0 . 0)(1000 . 0)())
  ((on b c) (1000 . 0)(0 . 0) ())
  ((on c a) (0 . 0) (1000 . 0)())
  ((on c table) (1000 . 0)(0 . 0)()))
 influences (((on b c) . -300))}
```

It states the facts as mentioned earlier and also that the table, C, and B have clear tops. Additionally, it states that the desired state is with A having a clear top. A problem with this description is that the desire for making A have a clear top and B and C NOT have cleartops is much too high: they are stated as absolutes rather than as strong desires. The measure reflects not just the final desire for the situation to exist, but the importance of maintaining the fact throughout. Notice that this is, in fact, a bug in the description: final goals and goal maintenance should not be confused if the system is to be run. However, the searching process will be able to overcome this problem.

Since a large table is assumed, the entry about the table having a cleartop should never change.

The influence that B not be put on C is a piece of advice that is assumed to have been given somehow. Notice this will lead to a problem when the system tries to put B on C as part of the legitimate activities associated with the request.

Now the description from the unit:

```
{p
 level 0
 purpose ((on ? ?))
 goals (((on ?a ?b) . 1000)
        ((on ?a ?c) . 0))
 pre-conditions (((cleartop ?a) . 900)
                 ((cleartop ?b) . 900))
 pairing-function structured-full-pairs
 added-goals (((on ?a ?c) . -1000)
              ((on ?a ?b) . 0)
              ((cleartop ?c) . 1000)
              ((cleartop (restrict ?b (lambda(x)
                                       (not (eq x 'table)))))
               . -1000)
              ((cleartop ?a) . 1000))
 bindings (?a ?b ?c)
 constraints
 ((boundp t (and (m-boundp '?a)(m-boundp '?b)(m-boundp '?c)
                 (member-accomplished-goal `(on ,?a ,?c)))))}
```

A few interesting things to notice: The entry `((on ?a ?c) . 0)` is meant to figure out what ?a is on when the unit is evaluated, which is deduced when the constraints, `(m-boundp '?c)` and `(member-accomplished-goal `(on ,?a ,?c))` are applied. Pre-conditions are that ?a and ?b have cleartops, which are well-stated, although perhaps too strongly, for, recall, the body of the unit may be able to do something like `(optimal-call-goals `(on ,?x table))`, where ?x is what is on ?a, for instance, at the time, thus doing backward chaining.

The added-goals are somewhat interesting as well. The entry `((on ?a ?c) . -1000)` means that ?a gets taken off of ?c. The mysterious entry `((on ?a ?b) . 0)` means that the entry, `((on ?a ?b) . 1000)`, will result in `((on ?a ?b) <des> (1000 . 0)...)` appearing in the situation description, since the added-goal modifies the entry by first zeroing the accomplishment portion (thus doing a *normalized* modification).

The entries, `((cleartop ?c) . 1000)` and `((cleartop ?a) . 1000)` simply assert the cleartops that are known at the time. The entry, `((cleartop (restrict ?b ...)) . -1000)` means that if ?a is put on ?b, that will result in ?b not having a cleartop, unless ?b is the table, which is big enough to never be cluttered.

The clever reader will guess that this isn't going to work in the straightforward way. Look at the initial evaluation of the unit, P, under all pairings of things to try:

```
((1000 . -900)
 (((((ON ?A ?B) . 1000) (ON A B) (1000 . 0) (0 . 0) NIL)
   (((ON ?A ?C) . 0) (ON A TABLE) (0 . 0) (1000 . 0) NIL))
  ((?A . A) (?B . B) (?C . TABLE)))
 2))

((1000 . -1000)
 (((((ON ?A ?B) . 1000) (ON B C) (1000 . 0) (0 . 0) NIL)
   (((ON ?A ?C) . 0) (ON B TABLE) (0 . 0) (1000 . 0) NIL))
  ((?A . B) (?B . C) (?C . TABLE)))
 2)
```

```
(((0 . -1000)
  (((((ON ?A ?B) . 1000) (ON C TABLE) (0 . 0) (0 . 0) NIL)
    (((ON ?A ?C) . 0) (ON C A) (0 . 0) (1000 . 0) NIL))
   ((?A . C) (?B . TABLE) (?C . A)))
  2)
```

The three choices are: 1) put A on B; 2) put B on C; and 3) put C on the TABLE. The first gets a rating of 100 $(= 1000 - 900)$, although the pre-conditions are not satisfied (A does not have a clear top). The second gets a rating of 0 because it does something that is desired eventually (puts B on C), to get a 1000, but it also gets a -1000 because it leaves A without a cleartop and makes B have one. Putting C on the TABLE is not something that is desired (hence a 0 positive rating), and, moreover, it leaves A without a cleartop and gives one to B, attaining a -1000 in the bargain.

It doesn't take much to see that things are set up very poorly in both the situation and unit descriptions, the possibilities of lossage being quite high. In general, the approach of simply having the above sort of description of what is desired with a minimal set of abilities is what is at fault more than the fact that even the implementation of the approach is bad. For instance, it is never stated that C must end up on the TABLE, though this is very much part of the human understanding of the problem. Given this fact—that the real goal state is `(and (on a b)(on b c)(on c table))`—HACKER could easily solve the puzzle. The reason for failing on this situation is that the TABLE is never distinguished as a special object, and there is no way of telling this with the predicates available *unless* the system is able to look at the ADDED-GOALS slot to notice that the TABLE is special.

For, how else is it to deduce that the TABLE is different? Even when made to reason as "if the goal situation was true, where would everything be?," this knowledge is specific to moving blocks (or things) in a gravity-filled world. Any way of improving the abilities of the system outside of added description (and hence with added *distinctions* given to the system), all that can be hoped is that blind search will work.

And that's just what Yh does in this case.

However, the rankings of the best things to try are `(on a b)`, `(on b c)`, and `(on c table)` in that order. If the strategy is to try them in order with resources proportional to the rating, `(on c table)` might not make the grade.[2] With counterinduction, `(on c table)` gets the most resources: with a `-basic-ration-` of 6 and a branching factor of 3 it gets 4.

The result, of course, is that putting C on the TABLE is the best idea, and it also provides the plan of action that succeeds:

```
((P (200 . -400)   ;alternative rating
    (((((ON ?A ?B) . 1000) (ON C TABLE) (0 . 0) (0 . 0) NIL)
      (((ON ?A ?C) . 0) (ON C A) (0 . 0) (1000 . 0) NIL))
     ((?A . C) (?B . TABLE) (?C . A)))
    2)

 (P (600 . -500)
    (((((ON ?A ?B) . 1000) (ON B C) (1000 . 0) (0 . 0) NIL)
      (((ON ?A ?C) . 0) (ON B TABLE) (0 . 0) (1000 . 0) NIL))
     ((?A . B) (?B . C) (?C . TABLE)))
    2)
```

[2] But you knew that already.

```
(P (1000 . -300)
   (((((ON ?A ?B) . 1000) (ON A B) (1000 . 0) (0 . 0) NIL)
     (((ON ?A ?C) . 0) (ON A TABLE) (0 . 0) (1000 . 0) NIL))
    ((?A . A) (?B . B) (?C . TABLE)))
   2))
```

The final rating is $(1000 \quad . \quad -300)$ since the admonition is still alive to not put B on C, but the plan of putting C on the TABLE, then B on C, and finally A on B is found.

Of course, whenever the unlikely candidate is the right thing to try one expects that schemes that favor that candidate will work. The main point of counterinduction is to balance the search in a domain in which the length of a plan is not a measure of its elegance.

7.7 Distinctions

As was complained about in the example, unless the system is given some basis of telling that the table is different than a block, it is unable to reason or manipulate those objects in a powerful fashion. Thus, a representation must allow *distinctions* to be made among the objects represented. In the example above, inadequate distinctions are made between the blocks and the table.

A representation must be able to let the system make distinctions between things that it is to reason about, and unless the system can "see" the same distinctions that the person can in a similar situation, one cannot hope that the system will act on the basis of those distinctions.

In an purely symbolic representation system without hierarchical networking, there is at most one distinction per symbol in a representation. That is, the symbol is present or it isn't, so that a symbol is equivalent to a bit in information theory (actually it is a bit plus a way of referring to that bit). One way of increasing the information content, and the distinction content, is to increase the number of bits or symbols.

With networking, it is possible to achieve added distinctions with higher level representations. In this system, the numeric measure, influences, soft, and hard constraints allow the system to talk about possible distinctions; with a large, powerful set of units with their related descriptions the distinctions provided by the mechanism can be realized.

7.8 Final Note

About the role of methodologies in the pursuit of scientific knowledge, the philosopher of science, Paul Feyerabend, said of the advice, "anything goes," in AGAINST METHOD [Feyerabend 1975]:

> *It is both reasonable and* ABSOLUTELY NECESSARY *for the growth of knowledge. More specifically, one can show the following: given any rule, however 'fundamental' or 'necessary' for science, there are always circumstances when it is advisable not only to ignore the rule, but to adopt its opposite. For example, there are circumstances when it is advisable to introduce, elaborate, and defend* AD HOC *hypotheses, or hypotheses which contradict well-established and generally accepted experimental results, or hypotheses whose content is smaller than the content of the existing and empirically adequate alternative, or self-inconsistent hypotheses, and so on.*
>
> Counterinduction does not go as far as this.

Chapter 8

Example of Generation

In this, the penultimate chapter, an extended example of the behavior of Yh on an interesting generation example is given. The example is a LISP program that does the Dutch National Flag problem.

Also in this chapter, some of the programming language specific knowledge that Yh has will be presented.

The knowledge sources are the language planning specific units, the lexicon, the programming language specific units, and the stylistic units (transformations).

The knowledge about programming languages (and the methods of talking about them) will be discussed as they come up in this chapter.

8.1 Context

The problem that Yh is given is to generate a simple explanation of the Dutch National Flag program as implemented in LISP. It is assumed that the program has been synthesized by the PSI automatic programming system [Green 1977] and is, therefore, annotated to a minimal degree and is a mystery to the user of the system.

That is, the program is represented, internally, by a semantic net-like structure, that is the result of deductive action by the synthesis system after a mixed initiative dialogue with the user. Thus, the details of the program produced and some of the data structures decided upon are a topic of speculation for that user. Yh attempts to explain the program in a way that increases the user's confidence that something resembling his expectations was synthesized. So, though the overall structure of the program is known, many of the obvious things an expert would get from reading the code are not apparent from the conversation.

Hence, what one should look for in the explanation is a medium level presentation of the data structures and the flow of control in simple terms.

8.2 Dutch National Flag

Given a sequence of colored objects in a row, each one of three different colors, the idea is to put all of the same colors next to each other in a row by only exchanging two objects at a time. The colors are normally red, white and blue, and those three colors in a row form the Dutch National Flag. The colors must also end up in a pre-determined place, say red on the left, white in the middle, and blue on the right.

So if one is given the initial row:

B	R	W	B	R	W	B

then one expects to get:

R	R	W	W	B	B	B

Thus, it is a form of sorting, and it can be done in linear time using an array and three markers into that array. The following is a simple MacLisp program that solves the problem where an array is used to the store the elements in the sequence:

```
;;; Dutch National Flag

(declare
 (array* (notype flag 1)) ;represents the Flag
                          ;can be r,b, or w. r = red
                          ;w = white b = blue
         (special n))     ;represents the length of the Array

(macrodef exchange (x y)   ;exchanges (flag x)  and (flag y)
 (let q ← (flag y) do
      (store (flag y) (flag x))
      (store (flag x) q)))

(macrodef redp (x) (eq (flag x) 'r))  ;tests if (flag x) is red

(macrodef bluep (x) (eq (flag x) 'b)) ;tests if (flag x) is blue

(macrodef whitep (x) (eq (flag x) 'w));tests if (flag x) is white

(macrodef incr (x) (setq x (1+ x)))   ;increments x by 1

(macrodef decr (x) (setq x (1- x)))   ;decrements x by 1

(defun dnf ()
 (let l m r ← 0 0 (1- n) do     ;initialize l,m, \& r
      (while
       (not (> m r)) do
       (cond ((redp m)
              (exchange l m)
              (incr l)(incr m))
             ((bluep m)
              (exchange m r)
              (decr r))
             (t (incr m))))
      t))
```

The flag is represented by a 1-dimensional, 0-based array of n elements, FLAG1. There are three array markers, L, M, and R, standing for Left, Middle, and Right, resp. L and M are initialized to 0; R is initialized to $n - 1$.

While M is not bigger than R the program does the following: If (flag m) is red, it exchanges (flag l) with (flag m), incrementing L and M by 1. If (flag m) is blue, it exchanges (flag r) and (flag m), decrementing R by 1. Otherwise, it increments M by 1.

In order to exchange (flag x) and (flag y), the program saves the value of (flag y), stores the value of (flag x) in (flag y), and then stores the value of the temporary in (flag x). An element of FLAG is red if it contains R, blue if it contains B, and white if it contains W.

The above three paragraphs are the explanation that Yh wrote[1] for this program, but from an internal representation which is annotated approximately as it is above.

In the rest of this chapter a little bit about this explanation will be discussed. As can be seen, the explanation isn't wonderful, and the writing style isn't likely to garner a Pulitzer prize, but it isn't too bad. The major drawback with explaining programs as a domain for natural language generation is that there are fairly stylized ways of talking about things, such as the actual algorithm part, which leave little room for invention.

8.3 The Input

The input to Yh consists of several stratifications (units with meta-units) which are derivable from the original semantic net that the acquisition phase of PSI produces. The data structures given below are the actual input, and is hand generated from representations of the program as plausibly acquired by PSI. The code shown above is a typical LISP implementation of the program described in the data structures below.

```
;;; Here's the program

{program1
 level 1
 meta meta-program1
 reply-method
 (lambda ()
  (np-unit-reply-method 'program1))
 code
 <This is exactly the code for DNF given above>

 data-structures
 ((dta-str1 . 1000)    ;The array
  (dta-str2 . 950)     ;Left marker
  (dta-str3 . 925)     ;Middle marker
  (dta-str4 . 900))    ;Right marker

 description (((a program) . 1000))}

{meta-program1
 level 2
 β program1
 meta meta-2-program1
 steps (((a setof (steps)) . 950))
```

[1]Well, not all at once it didn't, but each paragraph separately and at different times, with the history of the previous paragraphs and sentences saved and input for the next generation. The first paragraph took three minutes of KL-10 CPU time to produce, and Yh runs out of storage space when the full attempt is made.

```
code
(([lambda bind (initialize (l 0)(m 0)(r (1- n)))
                (step step1)]
  ([l variable (array-marker for FLAG)]
   [m variable (array-marker for FLAG)]
   [r variable (array-marker for FLAG)]) ;initialize l,m, and r
  ([while loop (step step2)
                (scan-array FLAG)]
   [(not (> m r))
    stop-predicate
    (stop-when (to-the-right-of m r))]
   [do keyword]

   ([cond conditional (case-statement with 3 cases)]
    ([(redp m) predicate (macro-call macro1)
                (tests (red (flag m)))
      (cf linguistic-entry in meta-macro1)]
     [(exchange l m) action (side-effects t)
                    (macro-call macro4)
      (array-operation (exchange (flag l) (flag m)))
      (cf linguistic-entry in meta-macro4)]
     [(incr l) action (macro-call macro5)
      (integer-operation (increment l))
      (cf linguistic-entry in meta-macro6)]
     [(incr m) action (macro-call macro5 with argument m)
      (integer-operation (increment m))
      (cf linguistic-entry in meta-macro5)])
    ([(bluep m) predicate (macro-call macro2)
                 (tests (blue (flag m)))
     (cf linguistic-entry in meta-macro2)]
     [(exchange m r) action (side-effects t)
                    (macro-call macro4)
      (array-operation (exchange (flag m)(flag r))
      (cf linguistic-entry in meta-macro4)]
     [(decr r) action (macro-call macro6)
                (integer-operation (decrement r))
      (cf linguistic-entry in meta-macro6)])
    ([t predicate true]
     [(incr m) action (macro-call macro5)
                (integer-operation (increment m))
      (cf linguistic-entry in meta-macro5)])))))
[0 value][0 value][(1- n) expression
                   (integer-operation (decrement n))])

data-structures
(((a (setof data-structures)) . 900))
purpose ((a program))
descriptors ((program . 800))}
```

```
{meta-2-program1
 level 3
 rankings ((steps . 900)(data-structures . 800))
 β meta-program1}
```

The description of the program contains pointers to units that describe the data structures and commented code. The commented code is nothing but the regular executable LISP code with some other entries associated with the steps. The format of a entry is [<real code><type of action> . <comments>] where an <real code> is nothing but a the actual code as represented by LISP. <Type of action> comes from a small set, including *action, predicate, bind, initialize, . . .* <Comments> are usually lists which have built in meaning to the programming knowledge expert part of Yh; these comments are intended to have the meaning that a person might guess using a priori knowledge of how people choose variable names.

One interesting part of the description is the set of data structure describing units along with their relative importances. This will prove of use in the order of generation of the description of the data structures.

```
;;; Here are the data-structures

{dta-str1
 level 1
 meta meta-dta-str1
 markers (dta-str2 dta-str3 dta-str4)
 type array
 element-type (one-of (R B W))
 dimension 1
 length N
 first-element 0
 last-element (1- N)
 name flag1
 base 0-based}
```

The above slots mean the obvious things.

```
{meta-dta-str1
 level 2
 β dta-str1
 meta meta-2-dta-str1
 data-structure-for program1
 descriptors ((array . 1000))   ;this points to another unit
                                ;that describes the base-unit
                                ;as a prototype
 represents ((flag1 . 1000))}

{meta-2-dta-str1
 level 3
 β meta-dta-str1
 meta meta-3-dta-str1
 element-type (((R represents RED) . 900)
               ((B represents BLUE) . 900)
               ((W represents WHITE) . 900))
```

```
 rankings
 ((type . 500)
  (name . 550)
  (element-type . 400)
  (dimension . 525)
  (markers . 425)
  (length . 950)
  (first-element  . 300)
  (last-element . 300)
  (base  . 525))}
```

The RANKINGS are the order of importance of the slots in the base-base-unit for describing this array. This is also used to order the adjectives when more than one is produced.

```
{meta-3-dta-str1
 level 4
 β meta-2-dta-str1
 element-type
 (((a list with element (pattern . number)) . 1000))}

{dta-str2
 level 1
 type array-marker
 meta meta-dta-str2
 name L
 value ()}

{meta-dta-str2
 level 2
 β dta-str2
 meta meta-2-dta-str2
 data-structure-for program1
 descriptors ((array-marker . 1000))

 name
 (multi
  (vp ((verb multiple present) (stands))
      ((verb multiple prep) (for)))
      (objp (noun (left))))
 value
 (((an (index into dta-str1)) . 900)
  ((an integer st 0 ≤ value < (LENGTH from dta-str1)) . 800))

 linguistic-entry
 (multi (det (the))(adj (left))(noun (marker)))}

{meta-2-dta-str2
 β meta-dta-str2
 level 3
 name (((a linguistic-entry st name is the subject) . 900))}
```

The slot, NAME, is a linguistic entry with a very limited meaning and is intended to explore the idea of using uninterpreted linguistic entries for the purpose of increasing the readability of a text that results from some interaction with a user. That is, if the user explains something some way which is unimportant to the program, the way that the object was referred to can them be used without having to understand the expression fully. In this case, the user said something like "L stands for left," which is actually a fairly sophisticated statement, but which has no relevance to the problem of synthesizing the Dutch National Flag program.

Thus the entries here just say that "<subj> stands for left" is a way of describing the unit DTA-STR2 where <subj> is some noun phrase referring to DTA-STR2.

The slot, LINGUISTIC-ENTRY is a way of referring to DTA-STR2 that is, again, a result of some comment by the user.

The other two array markers are similar to DTA-STR2, and are called DTA-STR3 and DTA-STR4.

```
{flag1
 level 4
 meta meta-flag1
 linguistic-entry (multi (det (the))(noun (flag)))}

{meta-flag1
 β flag1
 level 5
 description ((linguistic-entry for flag1) . 950)}
```

The flag is simply a unit representing whatever the user meant by *the flag* when it was stated. As far as the synthesis system is concerned, only the operationally relevant things about the flag (an array) are important, and *the flag* can then be used to refer to the data structure that was decided on to represent the operationally defined object, the flag. Again, Yh doesn't know the first thing about flags.

```
{macro1
 level 1
 meta meta-macro1
 stepof program1
 code (lambda (x)
       (eq (flag x) 'r))}

{meta-macro1
 level 2
 β macro1
 purpose ((a macro) (describes redp))
 descriptors ((macro . 900))
 step (((a macro) . 900))
 name redp
 type test
```

```
code ([lambda bind (bind (x argument))]
      ([x variable (an integer st 0 ≤ x < n)])
      ([eq predicate][(flag x) array-reference]
       ['r constant]))
linguistic-entry
(single (sent (subj (np (noun ((flag x)))))
              (pred (vp (present-verb (is)))
                    (np (noun (red))))))}
```

The above is the macro that tests whether an element in the array represents a red thing. The other macros for blue and white are called MACRO2 and MACRO3 resp.

```
{macro4
 level 1
 meta meta-macro4
 stepof program1
 code (lambda (x y)
       ((lambda (q)
         (store (flag y) (flag x))
         (store (flag x) q))
        (flag y)))}

{meta-macro4
 level 2
 β macro4
 purpose ((a macro)(describes exchange))
 descriptors ((macro . 900))
 name exchange
 step (((a macro) . 900))
 type action

 code
 ([lambda bind
   (initialize (x argument (an integer st 0 ≤ x < n))
               (y argument (an integer st 0 ≤ y < n)))]
  ([x variable argument]
   [y variable argument])
  (([lambda bind temporary (initialize (q (flag y)))]
    ([q variable temporary])
    ([store array-operation
      (moves (flag x) into (flag y))]
     [(flag y) array-reference]
     [(flag x) array-reference])
    ([store array-operation
      (moves the temporary q into (flag x))]
     [(flag x) array-reference]
     [q temporary reference]))
   [(flag y) array-reference]))
```

```
 linguistic-entry
 (single
  (sent (subj (np (noun ((()))))))
        (pred (vp (present-verb (exchanges)))
              (np (noun ((flag x)))
                  (conj (and))
                  (noun ((flag y))))))}
```

This description also contains some uninterpreted linguistic entries. Notice that these entries are parsed in the manner presented in Chapter 4.

```
{macro5
 level 1
 meta meta-macro5
 stepof program1
 code (lambda (x)
       (setq x (1+ x)))}

{meta-macro5
 level 2
 β macro5
 purpose ((a macro)(describes incr))
 descriptors ((macro . 900))
 name incr
 step (((a macro) . 900))
 instance-of incr-macro
 type action

 code
 ([lambda bind (initialize (argument (a variable)))]
   ([x variable])
   ([setq assignment (add 1 to x)]
     [x variable-reference]
     [(1+ x) integer-operation
      (integer-operation (increment x))]))
 linguistic-entry
 (single
  (sent (subj (np (noun ((()))))))
        (pred (vp (present-verb (increments)))
              (np (noun (x)))))}
```

```
{macro6
 level 1
 meta meta-macro6
 stepof program1
 code (lambda (x)
       (setq x (1- x)))}

{meta-macro6
 level 2
 β macro6
 purpose ((a macro) (describes decr))
 name decr
 descriptors ((macro . 900))
 step (((a macro) . 900))
 type simple-action

 code ([lambda bind (initialize (argument (a variable)))]
       ([x variable])
       ([setq assignment (subtract 1 from x)]
        [x variable-reference]
        [(1- x) integer-operation
         (integer-operation (decrement x))]))
 linguistic-entry
 (single
  (sent (subj (np (noun (()))))))
        (pred (vp (present-verb (decrements)))
              (np (noun (x)))))}
```

8.4 The Ball Starts Rolling

The fact of putting the above structures into Yh's memory does not start the process of producing an explanation. Some agenda items must be implanted.

Two such items are placed: one to print the final paragraph once it is written—this entry has a very low priority on the agenda so it is done after everything else is complete; one to start things going. This latter entry is simply a request on the situation description (SYSTEM-GOALS) that adds a desire to talk about PROGRAM1. The system goals also contains two influences for the above explanation. One is to not to use too many adjectives, the other is to attempt all collapses as soon as they are feasible, rather than waiting until the end. In addition, there is a basic complexity threshold in force that doesn't allow sentences to become too complex.

The agenda-adding expressions are:

```
(add-agenda
 'agenda 'test-perform
 '(progn
   (establish-workspaces () 'agenda)
   (add-right-empty-sentence)
   (putprop 'agenda 1 'current-priority)
   (flush-agenda-name 'agenda 'test-goal-select)
   (add-agenda
    'agenda           ;name of agenda
    'test-goal-select ;name of this entry
    'system-goals     ;the form (a system-goals unit)
    'goal-select      ;means to try to satisfy the
                      ;system-goals unit
    99.               ;priority (high)
    t)                ;runnable predicate
   (relax-all)        ;does the word use degradation discussed
                      ;in chapter 4
   (suicide))         ;sets the runnable predicate to ()
 'perform 100. t)
```

This agenda item just adds the agenda item that starts things off. The system goals unit looks like:

```
(setq -default-influences- '((adjectives . -300)))

(make-system-goals-unit with
 (name 'system-goals)
 (goal-list
  `(((say-sentence program1)(1000 . 0)(0 . 0) program)))
 (plan-list '(()))
 (influences '(((collapse immediately) . 900)))
 (soft-constraints ())
 (meta-method '(meta-system-goals-gc 0 ())))
```

This makes the request to talk about PROGRAM1, although it thinks that a single sentence will do. Of course, Yh will soon discover that this isn't true. The influence to 'collapse immediately' means that the collapsings will be tried fairly early on. The META-METHOD is the method that the meta-unit to this unit has, which is a garbage collection call for the system-goals, which flushes all goals that have an accomplishment rating (CF) greater than `superior-level`. Also, if the entire set is acceptable (Rg $>$ `superior-level`; see Chapter 7), the agenda item to continually work on this situation is flushed.

Next some other items are added; one to merge the sentences when finished:

```
(add-agenda 'agenda 'merge-sentences
            'merge-system-goals 'goal-select 50. t);
```

with the system-goals:

```
(make-system-goals-unit with
 (name 'merge-system-goals)
 (goal-list `(((merge sentences)(1000 . 0)(0 . 0) initial)))
 (plan-list '(()))
 (influences ())
 (soft-constraints ())
 (meta-method  '(meta-system-goals-gc 0 ())));
```

and finally one to print the answer:

```
(add-agenda 'agenda 'print-answer
 '(progn
   (establish-workspaces () 'agenda)
   (format msgfiles '|Final paragraph:|)
   (d-sent)   ;prints the sentences of the text
   (suicide)  ;kills this agenda item
   (relax-all)) 'perform 1 t).
```

8.4.1 Conducting an Inspection

The first thing that happens is that the unit representing the program (PROGRAM1) is examined carefully, especially the description of that unit. For PROGRAM1 the description simply states that it is a program, which means that the situation description is changed by deleting the entry `((say-sentence...)...)` and adding one to `((say-program program1)...)`.

At this point the control returns to the agenda handler, which sees that no modifications were done to the agenda, so it tries to find a unit to handle the current SYSTEM-GOALS, which is what the type of the agenda item says (refer to Chapter 5 for a discussion of the agenda). The previous attempt to satisfy the situation description changed that description, so a type of sequencing is occurring in which the description of the goals of the system are being re-examined, re-defined, and re-formulated as more expertise is brought to bear on it.

The unit that knows about programs is then asked to examine the program. Several actions take place based on this examination, nearly all of which add items to the agenda.

First it checks to see that there are no other programs that are known to it, so that a phrase like *the program* can be generated. This fact is added as an influence rather than as an item in the situation description. This is because this isn't a goal or a fact, but something to keep in mind and which influences the behavior of the system at key points.

The task is divided into two parts: discussing the data structures and discussing the code. The data structures come first (although, if the importance of the code was made higher than that of the data structures, one could expect that that the order might be reversed. If no preference is given, as in the current case, the data structures go first). Later discussing the code will be broken up into discussing the main program and discussing the auxiliary routines.

Recall that the data structures had a weight on their importance:

```
data-structures
((dta-str1 . 1000)    ;The array
 (dta-str2 . 950)     ;Left marker
 (dta-str3 . 925)     ;Middle marker
 (dta-str4 . 900))    ;Right marker
```

This weight basically determines the order of discussion for the data structures. The first of these is entered on the current SYSTEM-GOALS, after the request that called this unit is removed, as a simple goal of talking about the data structure the name points to. The rest are put in agenda items that add the proper request to SYSTEM-GOALS, start up a goal satisfaction request, and then kills itself. The priority on the agenda is proportional to the measures above. Thus, the proper sequencing on the order of generation for these data structures is accomplished with the agenda, although by arranging the measures correctly on the goals added to the situation description one could obtain a close approximation to the order desired. In fact, if one only had a preference for an order, the situation description is the correct sequencing medium.

A request to discuss the program code is placed at a lower priority than any of the data structure tasks, so that will appear later. A paragraph break will be inserted between the the two discussions.

The first data structure is then examined. It is the array that represents the flag. A simple examination of it causes the array expert to be called in.

Without special knowledge about things such as arrays, one wouldn't expect that much could be said about a topic without reverting to randomness or a strict representation.[2]

The array specialist knows about all of the interesting things surrounding arrays, such as the fact that they have dimension, a base, a length, a name, and possibly something they represent.

The first thing that happens is that all of the relevant features of the array are retrieved using the unit pattern matcher, along with the rankings of importance behind everything. In addition, the program that this is a data structure in is retrieved. All other data structures in the program are examined to see if they are arrays, and if not, this is known to be a unique array, justifying the use of the determiner, *the*. This fact is then added as an influence in case it is needed later.

At this point, then, there are a number of things to be said about the array, depending on what was found in the description and the importances associated with each part. There is also the mitigating fact of how high the desire was to talk about this array.

8.4.2 Digression on Importance

Mention will be made of this fact here and in the conclusion, that the measures discussed so far have a tinge of ad hocness to them, but with some good reason. Despite any claims of learning or evolution that might be attached to what these values end up being, they have an effect on how the program behaves, and though each is a unitary thing without much context, in accordance with the other things that are matched and associated with each entry in the situation description the values take on a tendency invoking factor. That is, each atom of description is a pattern and a measure. The measure stands alone, only in relation to the pattern, initially. But, if the pattern comes into contact with others that match it, there is a confluence of measures ending up in a different measure altogether. Therefore when programming the influence machine, the actual numbers are not important except insofar as tuning the behavior is important. The relative greatness or smallness of the measure as over against other measures is the true importance.

So, in this case, if there is only a small amount of interest in talking about the array in question, the details of that array will fade away and possibly not be discussed at all.

8.4.3 Undigression on Importance

Requests to generate adjectives about each of the points about arrays are gathered, but not added to SYSTEM-GOALS—since they will be ordered and added to the sentence by the special control structures in the array expert.

The things that are important to talk about for an array are: what it represents, its length, its base, its first element, its last element, its dimension, and the types of its elements.

Since the sentence that is being considered for generation is simply to talk about the array as an object—describing it—not much has been said about what the sentence will be. All things being equal, the sentence will be something like *There is a <describe the array>*, but not all things are equal all the time. If the array represents something then it makes sense to introduce the array with something like *The <describe array>*

[2]Strict representation. That means a very uniform representation with a tightly prescribed set of meanings for a limited number of options. I don't think that one can easily get by with a strict representation, but that a wide variety of representations with a uniform way of referencing them (that is locating them and invoking the specialized parts) is what will be needed in formative research years.

represents <what it represents>. However, since the thing the array represents is the central item in the user's mind, the topic of the sentence, which should be the subject (or agent), should be that thing, which will trigger the user's memory right away. Thus, a good strategy is to say the sentence as outlined here, but to then try to change the topic from the direct object to the subject, even if this means making the topic the formal subject only.

The options for introducing the array are, basically:

The <array> represents <something>.

There is <array description>.

The <array> has n elements.

The <array> is m dimensional.

The first element of <array> is <first element>.

etc.

Since the array represents the flag, and the entry

```
represents ((flag1 . 1000))
```

appears in the unit for that data structure, the sentence schema chosen is *the <array> describes <flag1>*. An influence is added that states that the noun phrases should not just use the names of things if they have them, but should also describe the objects in full.

Then, a SYSTEM-GOALS request is made (using *optimal-call-entries* which is described in Chapter 6) to write a simple declarative sentence with the agent being the array, the actions being *represent*, and the object being FLAG1. At this point there is some judgment that can be exercised regarding how to say the sentence, or at least which stratification should try it. In this case the specialized one for representations in programs is called, because the description matches with greater precision and it is at a higher level than the general sentence producer (see Chapter 6 for a discussion on how units are chosen). The description of the sentence to be produced includes a description of the subject, which is a pointer to the array unit, a description of the action, which is recommended to be a word that means the same thing as *represent*, and a description of FLAG1, which is a unit with a linguistic entry. If no sentence producing unit leaps to say this sentence, it is broken down further into a more detailed description, which will produce word choices that may trigger some sentence producing unit to take an interest. Failing that, the words will simply be produced in the hope that some observers of the process will find a way to make a sentence out of the mess.

However, despite the fact that the special routine is called, it eventually relies on the standard declarative sentence writer to do the main job. The unit, though, looks for ways of turning the subject and object around, but since there is no easy way to turn things around, it makes a passive transformation request. These requests are placed on a special unit that keeps track of where that various passives will be put and has the responsibility of deciding whether there are too many too close to each other.

8.5 Simple Declaratives

The simple declarative sentence generator does the obvious kinds of things. First it adds a blank, simple declarative sentence to the right of where it is, and if a position is given to it, it puts that sentence there. The blank sentence essentially looks like:

SENTn		
SUBJ	PRED	OBJP
NP	VP	NP
NOUN	VERB	NOUN
nil	nil	nil

The semantics of the sentence, SENTn is set by creating a unit which contains the situation description that called this unit, the sentence that this unit represents, where it is in the tower, and a *response* for this unit. The unit that is associated with a particular part the text is called a *response unit* since other units are able to hold a *conversation* with that unit in which the various parts of the unit can be retrieved by sending the response unit patterns that will trigger appropriate answers.

It is SENTn in order to index the sentences, but PRED because that indexing is done through the sentence number or through patterns within sentences.

By this is meant that if there is some other unit that wishes to find out either what a particular sentence means or whether a sentence exists that means a particular thing, a broadcast can be made asking this question in a somewhat stylized manner, which will set up a talk ring a co-routine message passing situation) with the unit that best responds. Messages can then be passed back and forth in a simple manner, where each message is a pattern in the pattern language that underlies the situation description patterns (the patterns that UMATCH can handle, not the patterns with measures in them). The patterns are then examined, usually with UMATCH and responses to that pattern can be sent back as the response. Thus, the internal format of these response units is irrelevant as long as the pattern language is kept relatively consistent throughout, which is easier to do than to keep track of slot names.

The rest of the actions of this unit are trivial in that it moves from left to right generating the agent noun phrase, the action verb phrase, and the object noun phrase. To generate these, a situation description is made up that reflects the request and a unit that can accomplish that is sought after.

8.6 Noun Phrases

The noun phrase generator is somewhat complex, though not as complex as it would need to be in a full generation system.[3] Nevertheless, the main noun phrase stratifications are fairly robust and produce interesting noun phrases.

For the current sentence, noun phrases for DTA-STR1 and FLAG1 must be generated.

The main description allows for position, number, and uniqueness to be passed to it, although it will default these appropriately.

The issue faced right away is whether the unit to be described in the name phrase has a name (a NAME slot) and whether that should be used as the noun phrase. If there is no compunction to use the full description, the name will be used. The way this happens is that an expert is consulted to make the name into a linguistic entry which is later used if possible. To turn a name into a linguistic entry the name is retrieved and turned into a tree structure which is later made into the simple tower:

[3] Yh is simply a demonstration system, not a production system. Some of the mechanisms, particularly the linguistic ones, are implemented in a very simple manner. The hybrid matcher, the stratification system, and the influence-based programming system are fully implemented and debugged, as are the stratified surface structure primitives. The repertoire of linguistic abilities is limited.

(Noun Proper)
<The name>

If there is a linguistic entry then control transfers (in a tail recursive manner) to a unit that will simply insert that entry into the tower.

This is how the expression, *FLAG1*, is generated in the current sentence.

If neither of these possibilities pans out, the description of the unit in question is consulted in order to find a way to refer to this unit. Consider, now, the situation when DTA-STR1 is approached; here the descriptor is simply `(array . 1000)`. This has the effect of searching the lexicon for words that refer to this thing. If there were a number of descriptors, then the word that best described all of them at once would be chosen. For an array, the choices are *array*, *set*, and *group*. *Array* fits best because there is little else to go on. If, for instance, there were unordered aspects to the array (i.e. if the implicit ordering was ignored and thus formed part of the description of the array), *set* might be selected.

Notice that it was not the case that a single word was specified at this point, but just anything that expressed the description of the single object. Had a phrase been generated, that entire phrase would be treated as a noun, even if it were a verb phrase, as would be the case if the unit represented an action, and the result would be a gerund.

8.6.1 Very Simple Noun Phrases

If a word must be chosen to express some unit, the single word selection program is used. This makes sure that something akin to a word (such as a phrase equivalent to a word) is actually selected.

8.6.2 Uniqueness of the Noun Phrase

Now the time comes to see what it is that is being generated; there are two cases: the unit that is being referred to has already been referred to; the description of the unit being referred to matches a description of a unit already referred to. In the first case the actual reference is equal to a prior reference (refers exactly to the same object), in the second the reference can be confused with a prior reference (does not refer to the same object, but to one whose description is similar to a prior reference).

In order to locate the first type of reference, the response unit protocol is used to broadcast a request. Since a unit name is given this will find all such references quite quickly. The second type of reference is a little trickier. The normal descriptive mechanism is used, in short, by putting the description that is associated with the unit that caused a noun phrase to be generated on the response unit as a normal description, with the additional descriptor that uniquely identifies this description as the description of a generated phrase. The indexing into this description, then, is only through this unique descriptor. For example:

```
purpose ((phrase generated for a description))
goals (((phrase generated for a description) . 0)
       ...<the rest of the description>...)
```

With the primitive, *optimal-find-goals*, one can specify the indices on which to retrieve units, so only ones with the above purpose slot can be retrieved. A threshold on how closely the description match determines whether the reference will be ambiguous.

If it is decided that two references are the same (the first ambiguous case) a request to consider collapsing the sentence in which they occur will be made on a special unit, much as the transformations are put on a special agenda-like unit for later consideration. In the case of ambiguous references a request to find

distinguishing descriptors is posted. The descriptions will be scanned for the most important distinguishing descriptors, which will then be placed in prominence in the noun phrases. Other tactics such as increasing the distance between the two references might be tried also.

The semantic unit is created with the same sorts of things that appear in the semantic unit for the sentence, but with care taken that this unit clearly represents a noun phrase.

The next decision is which determiner to use, *the* or *a*, or whether to use no determiner at all. If it is specified that there should be none then none is used. If there are no other noun phrases referring to the same thing, then *the* is used. Then the situation description is examined to see if any words of wisdom about the uniqueness is available. If there is some word then that is used. If no word is given, then *the* is used; and if the answer is unintelligible, *a* is used. Of course, if there are no other units that represent the same generic item (such as other arrays), then *the* is used. If the noun phrase is plural,*a* cannot be used, but the above procedure applies otherwise.

8.6.3 Adjectives

The calling unit can specify that no adjective should be used. This would be the case when the caller would want to add adjectives in some order that may not be what the noun phrase unit would choose, and if the calling unit considers itself an expert, this would be what happens.

The adjectives used to describe the noun come from the slots in the unit being described, and the order of appearance is inverse measure order using the RANKINGS in the meta-meta-unit.

In the case of DTA-STR1, of course, the modifers in the noun phrase will be generated by the array expert, since the entire sentence is by way of explaining or introducing the array.

The rest of the modifiers that can be attached consist of special modifiers (mainly other adjectives) that special routines are searched for to accomplish, and prepositional post-modifiers. These fancier types of modifiers are under control of a distance measure to the nearest noun phrase that refers to the same object to the left. That is, the further to the left a noun phrase referring to the same unit is the less the negative influence (via a soft-constraint) there is to put in these fancy modifiers, which have the effect of pinning down the reference better.

8.6.4 Prepositional Phrases

If any interesting post-modifiers are suggested in the situation description in the form of prepositional phrases, these are added at this point.

8.7 Verb Phrases

Verb phrases are handled very simply by performing the same standard semantic checking processes as the noun phrase generator does. The actual verb phrase is selected with the single word generator, but that can produce a number of words as there can be multiple word or word-preposition complexes that express an action.

8.8 The Array Lives

The sentence produced by the operation thus far is:

SENT1				
SUBJ		PRED		
NP		VP	OBJ	
DET	NOUN	VERB	DET	NOUN
The	array	represents	the	flag

The array specialist isn't through, though, since it has a number of other things it wants to say about the array. Those things, in order of importance, are: the length, the name, the base, the dimension, the array markers, the element-type, the first element, and the last element.

Remember that there is a -300 negative influence against being too verbose with adjectives, which will cause some of these modifiers to be left out.

The array specialist goes through this list one at a time in this order and decides which of them to try, keeping in mind that it knows that the sentence it just generated contains the noun phrase *the array*. The first thing considered is the length, which is then advertised for, since there are a number of ways to talk about the length of an array, which will interact with some of the redundant other information on hand to discuss.

For instance, if the first and last elements are mentioned, or the base and the last element, then the length can be skipped. Which of these alternatives to used can depend on whatever aspects of the array have been discussed or whether there is some advice about what to discuss; the evaluation of what to say here is left up to the pattern matcher discussed in Chapter 6.

Given that the length is to be said directly, there are several ways to accomplish this. One is to say, *the. . ., length n,. . . array*; another is . . .*array. . . of n elements*.

In the case at hand, the latter is chosen, but a negative preference is added to it, which decays with time, so that the other possibilities will be used, if the same request is made later.

In each of there cases, the unit in charge controls the activity, using the goal-calling primitives (*optimal-call-goals* for instance) to actually insert words or phrases.

The current sentence is:

The array of n elements represents the flag.

The name of the array is next, and the name is attached as an appositive at the end of the noun phrase, yielding:

The array of n elements, Flag1, represents the flag.

Next comes the base, in this case it is a zero based array, which means that the first element is called 0. *Zero–based* is considered a single word, and is added by a specialist unit on adding adjectives, which locates the noun within the current noun phrase using the primitive, *match-position*, which is discussed in Chapter 4. The list of adjectives is located and the new one is added at the front of this list.

Similarly, the modifier, *one-dimensional* is appended to the front of the sequence of current adjectives. Thus far the sentence is:

The one-dimensional, zero-based array of n elements, Flag1, represents the flag.

While processing these possible modifiers some simple reasoning occurs about what to say about the array once it is known that some things can be said. If the first element of the array is specified, then talking about the base of the array is not as important, and vice versa; if the length and the first element or base is discussed, the last element does not need to be, etc.

During the adding of these modifiers, counterinduction was used three times to recompute the value of adding some marginal modifiers, as discussed in Chapter 7. The result was that the first and last element are not discussed in preference to the base and length.

There is a transformation pending for moving the direct object of this sentence to a more prominent focus.

8.9 Those Pesky Array Markers

An array marker is just an index into an array which is used to "keep one's place." In the Dutch National Flag program there are three array markers, one to mark the place to put red objects, which moves upwards, one to mark the place to put blue objects, which moves downwards, and one to scan through the array examining the color of things it finds, placing them in the right place.

Once it is decided to talk about one array marker, it is often wise to discuss them all in one place, although this can lead to problems associated with parallel constructions that need to be collapsed.

The specialist for array markers checks to see whether this array marker in question has already been discussed, which would have been posted as an influence. The array that this is an array marker for is retrieved from its description, `(an (index into dta-str1))`, and this allows the specialist to retrieve all of the array markers for that array.

It sets up a dispreference for using the name of the array markers as the only referencing expression, and Yh tries to find a unit that will express the stereotyped phrase, *there are* <*n*> <*objects*>. This is fairly straightforward: It adds the sentence to the right:

SENT1	
SUBJ	PRED
NP	VP
(Noun formal)	(Verb plural)
There	are

SENT1	
PRED	
OBJP	
ADJ	(Noun plural)
three	array markers

The array marker specialist then decides to add the names to the right of this sentence as an ordered appositive, which will have a list of names and a parallel list of descriptions attached to the end. This is a standard way to introduce a list of objects with descriptions and names, attaching the names in a parallel construction, and though it is very idiomatic there is no good reason to not use this technique. Moreover, there are some linguistic entries associated with these markers which are 'known' to be meaningful to the reader, although what they mean is a total mystery to the system.

The specialist makes up an extended description of what it wants done: the stereotyped phrase description, the list of names and associated descriptions, and some other hints to the next unit to be called. The descriptions that are of interest are the ones such as this one from the left-hand array marker:

```
name
(multi
 (vp ((verb multiple present) (stands))
     ((verb multiple prep) (for)))
     (objp (noun (left))))
```

The stereotyped phrase specialist adds the phrase,

,L, M, and R, standing for left, standing for middle, and standing for right, respectively.

Notice two things that must be true in order for this to have been successfully generated. First, the gerund form of the phrase, *stands for x* must have been deduced, and, second, a sub-tower had to have been created to hold these gerunds, since they are clause-like and do not properly fit on the, so far, monolithic tower. The *and* that appears in the list of names, *,L, M, and R*, is at a different level than the conjunction that appears between clauses; this is the so-called appositive conjunction. The difference is the level at which the annotation occurs, the *and* in a noun phrase is within the noun phrase and conjoins things within it while the full conjunctive *and* is between phrases or clauses.

Creating the gerund requires making a search for stratifications that might be able to help, which means describing the phrase that is the center of attention, which, one hopes, will be an idiom in the lexicon. There is such a unit, and it proceeds to place the current position marker over the place where *stands* is, which is, by the way, in the relative clause sub-tower, which has been made the current tower by the calling unit. The unit then examines the lexicon entry for *stand for* and extracts the progressive entry, placing it in the tower at the proper place, and annotating the phrase as a `(vp gerund)`, which is a noun phrase made out of a verb phrase.

This happens for all three phrases. *Respectively* is added to the end of the sentence, and a request to consider collapsing the three parallel constructions is made. Since there is an influence that states that it is better to collapse immediately than to wait, the collapsing is attempted right away.

Up to this point Yh has been lucky that all of the things that it needed to do regarding special phrases or circumstances have been handled by a specialist in that area. But, luck can run out, and in the situation of collapsing these parallel phrases, there is none left. The collapsing is done via a unit that holds all of the requests for possible collapses. This unit merely scans the set of proposed collapsings and tries to locate someone to do the job after the proper context is set up. This unit also accepts communications from the units that generate text to find possible collapsings to try and schedules those attempts.

So the collapser makes up a full description of what is to be done, including a description of the phrase in the hope that a specialist can be found, but none is found in this case. The three phrases are handed off to a general phrase collapser which just tries to eliminate all but the non-common words from each phrase but the first. Thus given the phrases, *standing for left*, *standing for middle*, and *standing for right* it will try to get rid of *standing for* from the second two.

The transformation, however, does not eliminate the extra words *per se*, but simply embeds the words in the structure, `(formal <word>)`, which has the effect of hiding the words from view (the sentence printer ignores these entries), but leaves the original wording available.

8.10 Note on Multiple Nouns

The noun, *array marker* is a multiple noun, and is marked in the stratified surface structure as (NOUN MULTIPLE), which indicates that the noun is compound and basically made up out of noun-noun or other unspecified modifications. Remember, since the semantic content of each part of the text is readily available, there is not such an onus on making the syntactic representation precise, and less on making it traditional (see the discussion in Chapter 4).

8.11 Pondering the Issues

Sometimes a simple examination of the explicit properties of an object does not bring forth all of the interesting things that might prove useful. For instance, one interesting thing about an array marker is what it is initialized to. In the representation above this fact is not mentioned outright, but is hidden in the code.

In this case the array marker specialist invokes a unit that reads the code and finds out what the various markers are initialized to. Since the annotated code states the purpose of the lambda binding it is possible to specify which *lambda*s cause initialization rather than saving/restoring.

Here the lines:

```
(([lambda bind
   (initialize (l 0)(m 0)(r (1- n)))
   (step step1)]
```

hint that the initial values are 0, 0, and $n - 1$.

Thus, three new sentences are proposed, one for each initialization, results in the sentences:

L is initialized to 0. M is initialized to 0. R is initialized to $n - 1$.

The names, L, M, and R, are used because there is no requirement to fully describe the markers since they have already been introduced.

To express `(1- n)`, a special routine is called that will convert the standard LISP prefix to mathematical infix for external printing purposes. The internal LISP syntax is retained for the semantic units, however.

As these last three sentences are generated, it is noticed that the first two have the same direct objects, and all three have the same verb phrase. Additionally, the previous sentence about the array markers are noted to have used the names, L, M, and R.

8.12 The Fun Begins

Given that the initial paragraph looks like:

> *The one-dimensional, zero-based array of* n *elements, Flag1, represents the flag. There are three array markers, L, M, and R, standing for left, middle and right, respectively. L is initialized to 0. M is initialized to 0. R is initialized to* $n - 1$.

there are still some loose ends and transformations to apply. The first is to transform the first to the passive voice in order to change the focus from the array to the flag. The passive transformation was described in some detail in Chapter 4. The first sentence becomes:

> *The flag is represented by the one-dimensional, zero-based array of* n *elements, Flag1.*

Next the possible collapsing of sentence 2 with some of the last three is considered, since they all mention L, M, or R, but since in sentence 2 they are used as direct objects, more or less, and as the subjects in the last three, there is no known way of doing this except for making further relative clauses on the direct objects, which would result in a sentence like:

> *There are three array markers, L, which is initialized to 0, M, which is initialized to 0, and R, which is initialized to* $n - 1$*, standing for left, middle, and right, resp.*

which, combined with the other appositives in parallel would end up a complete mess. This option is rejected because of an admonition to not mix relative clauses amongst parallel appositives.[4]

The last three sentences pose something of a problem because they are so closely related to each other. All three have the same verb phrase structure, and the first two of the last three have the same direct objects. The latter fact means that the first two are collapsed by merging the predicate parts. This means that the subjects are conjoined with *and* and the verb phrase is turned into the plural. The direct object is left as it is. The situation description that resides in the sentence and phrase level of the text is patched to reflect the fact of the multiple noun phrases.

[4]It is important to note that this is a choice that is made according to style rather than correctness. It is not incorrect to use such a complex construction, and it may or may not be ambiguous to do it that way, but it is a matter of taste to choose a different path.

The new third sentence is then:

L and M are initialized to 0.

The last sentence has the same verb phrase as the one before it, but that previous sentence is fairly complex, to some degree, already, so it choses to simply bring them closer together with a punctuation change. The last sentence of the paragraph, hence, becomes:

L and M are initialized to 0; R is initialized to $n - 1$.

Another possible route for these last three sentences is to use the parallel construction:

L, M, and R are initialized to 0, 0, *and* $n - 1$, *resp.*

This is not done because it produces a sentence with the same structure are the one before it. This is determined by producing the description of the sentence that would result from the collapse of the original last three sentences in the paragraph and comparing it with the descriptions of other sentences in the current, transformed paragraph.

8.13 The Rest of the Paragraphs

The rest of the paragraphs will only be discussed in rough form, the main choices simply outlined.

Much of the rest of the explanation is groveling over the code and talking about it in standard meta-evaluation order.

While M is not bigger than R the program does the following:

Since there is no particularly good way to talk about WHILE loops, this stylized form is used. The first sentence is produced with the colon and a notation is made to indicate that the following sentences are part of the topic of the colon.

If (flag m) is red, it exchanges (flag l) with (flag m), incrementing L and M by 1.

This starts off as *If (flag m) is red, the program exchanges (flag l) with (flag m). The program increments L by 1. The program increments M by 1.*

In order to obtain the phrase, *(flag m)*, Yh looks at the linguistic entry on META-MACRO1, which is what the entry in the annotated code for the program says to try:

```
([cond conditional (case-statement with 3 cases)]
 ([(redp m) predicate (macro-call macro1)
            (tests (red (flag m)))
   (cf linguistic-entry in meta-macro1)]
```

Similarly, the phrase for *incrementing* (and *decrementing* below) are obtained through built in linguistic entries.

Also, the noun phrase generator notices the repeated reference to *the program*, discusses it with the response unit that has been set up for that purpose, and thus makes proposals for collapsing the various sentences and for turning occurrences of *the program* to *it*.

Likewise the next sentences are produced:

If (flag m) is blue, it exchanges (flag r) and (flag m), decrementing R by 1. Otherwise, it increments M by 1.

A new sentence is enforced for each clause in the COND, which is judged to be clearer than any attempted collapsing. A further built in restriction is that the length of a resulting sentence is a negative factor in making any proposed collapsed sentence beyond some amount. These are represented as influences in the system. Actually, the length influence has a measure that floats, and as shorter sentences are stated

longer ones are encouraged. In the rest of the explanation, longer sentences, and hence more collapsing, are encouraged.

8.13.1 Speaking of Macros

The macros discussed are rather simple as well, depending on a simple tree-walk through the code. The only interesting new thing here is that the LAMBDA that is used is a BIND, not an initialize, which means that the value is saved as a temporary rather than initialized to some value. The exchange macro and the predicate macros are discussed by Yh since they are not of type SIMPLE-ACTION.

As mentioned above, longer sentences are encouraged in this last paragraph[5] so that a fairly rambling style is produced:

In order to exchange (flag x) and (flag y), the program saves the value of (flag y), stores the value of (flag x) in (flag y), and then stores the value of the temporary in (flag x). An element of FLAG is red if it contains R, blue if it contains B, and white if it contains W.

Talking about what an array has in its elements is known to be accomplished by using phrases such as *contain*, *has in it*, etc. The *then* is used to reinforce the sequential nature of the process (built in to the sequential code description generator).

In any case, by looking at the representation of the input it is fairly clear where most of this information originates.

8.14 Alternate First Paragraphs

By increasing the dispreference of adjectives and adjusting the influences on how things, such as modifiers, can be introduced, the following paragraphs would result in place of the first one:[6] *The flag is represented by an array of n elements, FLAG1. It is a 1-dimensional array. There are three array markers, L, M, and R, standing for Left, Middle, and Right, resp. L and M are initialized to 0, the first element; R is initialized to $n-1$, the last element.*

The flag is represented by an array of n elements, FLAG1. It is a 1-dimensional array with N elements. There are three array markers, L, M, and R, standing for Left, Middle, and Right, resp. L and M are initialized to 0, the first element; R is initialized to $n-1$.

8.14.1 A Weird Alternative

Suppose that all of the structure producing units are removed from Yh, leaving only the programming knowledge experts and the lexicon, what will the system say? It says:

Array. N elements. Flag1. Represent. One-dimensional. Zero-based. Array markers. Three. L. M. R. Standing for Left. Middle. Right. L. Initialize to. 0. M. Initialize to. 0. R. Initialize to. $n-1$.

8.15 Conclusion

Yh does not produce particularly startling explanations of programs, yet they are certainly fairly easy to understand and to at least a small extent in a style that is not inhuman. In order to become a good system,

[5] To test this part of the system.

[6] These paragraphs were only simulated by having Yh manipulate the descriptions without ever modifying the text data structure, which was done by hand. In other words, I let Yh make the decisions and I did the work.

many more transformations and basic sentence types would need to be put in. Better descriptions for the existing stratifications would be useful.

The explanation system described is simply to demonstrate the feasibility of this approach, not to baffle the reader or to be part of a very good production system. Much information must be provided to the program in order to get it to produce the explanations it does, but these are produced by the program synthesis system.

Chapter 9

Conclusion

And would it have been worth it, after all,
Would it have been worth while,
After the sunsets and the dooryards and the sprinkled streets,
After the novels, after the teacups, after the skirts that trail along
the floor—
And this, and so much more?—
It is impossible to say just what I mean!
But as if a magic lantern threw the nerves in patterns on a
screen:
Would it have been worth while
If one, settling a pillow or throwing off a shawl,
And turning toward the window should say:
'That is not it at all,
That is not what I meant, at all.'
Eliot

After having read several hundred pages about what seems to be a very complex system, Yh, that appears to do comparatively little, the question must arise: What is the point of all of this?

Several hundred pages?

The original typesetting had larger line spacing and wider margins, yielding a 200+ page document.

9.1 Creativity and Discipline

Yh is a program that generates natural language descriptions of simple algorithms from their semantic representations. The system emphasizes the planning, linguistic, and creative aspects of generation rather than the theory of explanations *per se*. Most of the planning process involves making stylistic compromises after reasoning about the linguistic abilities (and inabilities) of the system. The planning process produces an adequate and diversified set of initial sentences that are later subject to stylistic and semantic transformations which manipulate the text into its final form. That is, the system models deliberate writing rather than spontaneous speech.

Most importantly, Yh is built on a system that encourages a different style or methodology for writing very large systems that demonstrate a wide variety of behavior in a rich environment. What I mean by this is that in some domains there is an abundance of information and techniques that are applicable to a given

situation, while in others—and in particular in those that have traditionally been studied by researchers in AI—there is often an underabundance of techniques, or only exactly the number required.

9.1.1 Two as One

Natural language generation, also, is a unique domain for AI research by emphasizing that richness is a source of relief in solving the problems encountered as well as a source of problems. Repetition of a phrase can be taken as an indication of relatedness of referent when that may not be the case. In other words, if I use similar expressions to refer to different things, there is a tendency to think that the two different things are related, if not the same.

As expected, I will use an unusual example from literature to demonstrate this point, one which will hint at the subtlety that exists in real writing, and, hence, creativity. The example is from MOBY DICK by Herman Melville [Melville 1851]. In that book there is some evidence that Melville intends to identify in some ways Ahab with the Whale, some saying that he is making an analogy to Christ and God. To support this is the fact that several of the descriptions of each are similar. For instance, in Chapter 41, Melville writes of the Whale:

> *For, it was not so much his uncommon bulk that so much distinguished him from other sperm whales, but, as was elsewhere thrown out—a peculiar snow-white wrinkled forehead, and a high, pyramidical white hump.*

and in Chapter 44, he writes of Ahab:

> *. . .the heavy pewter lamp. . .for ever threw shifting gleams and shadows of lines upon his wrinkled brow, till it almost seemed that while he himself was marking out lines and courses on the wrinkled charts, some invisible pencil was also tracing lines and courses upon the deeply marked chart of his forehead.*

Thus, though he never states outright that Ahab and the Whale are related except as hunter and hunted, or as obsessed and obsession, the fact of two similar descriptions of the two bring them close, and perhaps suggest a closer, familial or even identical relationship.

9.1.2 One as Two

In generation there is a great sense of being able to kill two birds with one stone, or even more birds with a little bigger one. For example, suppose you want to talk about the color of a young girl's eyes, and the girl is 3 years old; and suppose that you don't want to specifically refer to the age, but to hint at the youth—that is, in the jargon built up in the rest of the thesis, there are influences to specify youth but no direct request. Then you might use a phrase like . . .*girl's baby blue eyes*.... The *baby blue* part refers to a color, which is a little lighter than regular blue, but the possible lie associated with the exact color may be compensated for by the nice tendency the reader has to think of the girl as younger rather than older. This is a matter of judgment, not of iron-clad deduction or reasoning.

Creativity involves the ability to make choices on the basis of current and past tendencies, but it requires a large number of choices, and it requires that the distinctions available be subtle enough to capture the nuances desired. Creativity is NOT being able to solve a puzzle cleverly, necessarily, nor is it coming up with a complex, involved plan of attack, but it is using techniques available in an unexpected, new way, or putting things in unprecedented order.

9.1.3 What the Thesis is About

This thesis is really about methods for programming a system that worries about such things as making a number of choices in a rich environment, altering behavior over time and experience, taking risks in a

soft world. Such environments or worlds are what I mean by a *fluid domain*. That a system is written that generates explanations of programs is only of minor interest, since if I were to do exactly that and nothing else, a simple, fast, unimaginative program could have been written in a small fraction of the time with a higher performance, but in a restricted domain.

If you are going to write a program explanation system, my advice is to do it some other way, though some of the ideas here may be of use; if you want to think about creativity in writing, try the methodology used in Yh.

9.2 Underlying Language Issues

Yh as a language generation system is transformational in nature, and this fact follows mainly from the fact that deliberate writing is being modelled. Deliberate writing, by definition, means that the text is being re-written, which means that the current object is being transformed in some way. It turns out that a transformational system of sorts was the easiest to use in this application, and no relationship to Chomksy's Transformational Grammar is intended by this choice.

The transformations apply to tree structures, in the normal way, to strings of phrases, clauses, and classes of words, and to strings of words in the text. This meant that a representation that allowed access to the tree structure and to the string structures was needed, and hence I invented the *stratified surface structure* which is simply the first thing I thought of that would solve the tree-string puzzle.

Transformations provide, also, a method for talking about, in a uniform manner, the kinds of syntactic observations we want to make about the text, such as where various things occur. The semantic observations are made by units which are attached to the syntactic entries in the text. These units can be accessed by the full descriptive mechanisms discussed throughout the thesis, which means that a scan through the text need not be made in order to find out various facts about the text.

The fact that the entire semantic content—as it can make sense to Yh—is available directly, there is no need to have a full parse tree for the text, only the main features need be represented in standard formats. This has proven to be a big plus since a full parse tree with frills is a pain to work with.[1]

9.3 Observation and Participation

There is a distinction between doing something and noticing that you are doing it, and some actions can only be based on the noticing. In Yh there are units that produce noun phrases, and there are units that notice that the same noun phrase appears more than once. Finally, other actions are taken on the basis of the observation.

The process of describing an object is the proper domain for an observer, for only in the context of an observer can the distinctions necessary to the description be made clear. When I describe a noun phrase I can choose to describe the letters in it, the words in it, the reasons I chose those words, why I said the noun phrase in that sentence, or why I said *that* noun phrase (to get you to believe something, for instance).

Since a description is within a context, meaningless outside, it then makes sense to separate the description from the object, and thus the context can be placed outside the object as well. This is not to say that any given implementation of an observational item needs to be physically (or implementationally) separate from the object, but any available distinguishing entities in the system (and in our minds too) should be able to distinguish the participant from the observer, and hence the description from the described.

[1] Plus, keep in mind that I am NOT a linguist, and so I don't claim to make a big point in that area. I'm working on mind models and creativity—variable behavior.

9.3.1 Descriptions and Stuntmen

Descriptions can act as stand-ins for the actual participants in making judgments in a domain. More precisely, if there is some description of an object that purports to describe the activities of a unit, we can manipulate that description and *pretend* that the actions have taken place in order to get some speculation on what other actions can then be taken.

With a rich descriptive language, and a matching language that does not mind ignoring things when necessary, there is no reason to avoid full descriptions of every facet of an object. We can use the measures on descriptors to put things in their place, but to leave possible distinctions out is to rob the system of potential power.

9.3.2 The Problem with Puns

With descriptions as part of the observing entities we can also attach many different observers to any given object and thus partition the views of an object without unwanted interference. So, for words with several meanings, we can have a different description for that word, each of which can be manipulated by the system, but with limitations on the interactions between them, and we can thus gain additional context as part of the bargain.

For instance, consider the word, *bug*, which means either an insect or a problem in a program. In the notation given earlier we have two choices on how to represent this curious fact. One is:

D_1 ((<insect> . n_1)
(<problem> . n_2))

in which n_1 and n_2 are of comparable measures since they about equally are adequate expressions for the concepts they represent; that is, *bug* is as good a way to say *insect* as it is to say *problem with a program*. Another way is to have the multiple descriptions

D_2 ((<insect> . n_1)
(<problem> . n_3))

D_3 ((<problem> . n_2)
(<insect> . n_3))

where n_3 is some smaller measure.

In the first case, when there is interference between the concepts, such as when we are talking about programs about insects (in either pun-ish interpretation), the first representation may be too sensitive about the crossover: the strength of the match is too high because of this presence of the other interpretation in the picture. In the latter case, multiple descriptions are in better control, the interference strength there being decoupled from the normal, expressive strength.

And the problem with puns is that in a generation system done right, in which reminding is an important part of the behavior of the system, you are never going to be able to avoid puns, and you may even have to go to great lengths to be rid of them, such as using multiple observers. If there is some influences that are added due to a certain choice of words, then the results of letting these influences help choose further words can be a problem. So, if there is a program about insects that has a problem, then the use of the word 'bug' for this problem may be intolerable. On the other hand, if I am describing a love relationship which is also a physical relationship, I might use the word 'passion' in place of 'love,' which is entirely acceptable, and both arise from the same mechanisms.

9.4 Representations and Distinctions

A representation is a set of objects and operations on those objects which allow an internal manipulation to reflect a perceived external manipulation. So a representation of an engine should allow a program, or whoever, to look at its parts the way we can look at the parts of a real engine. And postulating putting a part in a certain place in the representation requires that part can be put in that place in reality as well. But to have such a faithful representation means that there is an observer who can observe that the faithfulness is exhibited. Without an observer of this correspondence the question is truly meaningless. This means that there is no objective measure of the appropriateness of a representation; faithfulness is a judgment in the life of an observer, not in the life of a representation.

More importantly for practical applications, a representation maintains distinctions. A distinction is a partitioning of the world (whether internal or external) into things that can be distinguished. A representation that allows few distinctions is worthless. But, a distinction is not worthwhile either unless the basis of distinction is available as well. For if all we can say about two things that are distinct is that they are distinct, then the distinction does not help us decide in which case one distinction is appropriate and the other inappropriate.

In the simple blocks world example in Chapter 7 I made the point that since TABLE and BLOCKA, for example, are not distinct that a planning system has difficulty understanding that they are to be used differently. Since the representation scheme that underlies this can always distinguish using LISP EQ, it can always make the distinction. The distinction of interest in this case is that of differing use. A table cannot be placed on a block, and every structure of blocks has the table at its base. These facts trim the search space, and solve some anomalous situations. These distinctions are required to be made explicit in the representation and not only in the operations on that representation.

For, suppose that the MOVE operation knows that the TABLE must be at the bottom of a tower (by knowing that it cannot be moved onto anything), then this does not help the planning system know that if you want a tower A on B on C, that C is on the TABLE. This is the distinction that needs to be made to allow unintuitive—sometimes—choices while planning in this domain.

9.4.1 Webs Versus Monoliths

The main point of the descriptive mechanism is that it allows one to specify distinctions in a monolithic manner in order to make them more easily available. By putting measures on the unordered set of descriptors one is able to distinguish on the basis of those measures, or ignore their influence altogether.

In hierarchical or tangled hierarchical representation schemes, such as KRL or semantic nets, these distinctions are made available through search chains of, for instance, ISA links. For example, one can represent the blocks and table world in such a system where the ISA links on a block and the table eventually lead to prototypes which state the distinction. And one can obtain the fine dusting of distinction in the structure of the web that exists around the objects.

So, in the measure of a descriptor we can observe the relative importance directly without needing to untangle webs and, hence, use resource limited searches to diminish in proportion to the importance.

I find, too, that the description pattern matchers explained in Chapters 5, 6, and part of 7 capture nicely some of the things we want to get out of the resource-limited matchers, in which there is some sense of partial matches or undefined results. In these matchers, as in the KRL matcher, when we match two things we end up with a set of corresponding features that substantiate the match and some notion of how strong the match is based on the amount of resources dumped into the effort. In this matcher, we end up with a set of possible pairings of descriptors and the total strength of the match, which can then be thresholded or whatever to produce the behavior you want.

Further, in the framework of Yh there is a mechanism for saying "match these two things, and I guess I'm sort of willing to let these strange things be true to some extent during that match." This is the INFLUENCE and SOFT-CONSTRAINT mechanism.

In short, this descriptive framework and matching procedure can form a common ground for expressing many nuances and temporary tendencies without the need to bury these things in a static, difficult to work with, and thus more permanent, nest-like structure.

I find that it is nice to be able to match two descriptions and have something other than T or NIL come back, and this means that I can expect to compare apples and oranges to some profitable conclusion.

On the other hand, many people balk at the idea of a non-symbolic (i.e. quantitative) methodology for accomplishing these ends, though they find the idea of resource expenditure profitable.

Here I want to make a distinction between *intelligence* on one hand and *judgment* on the other. judgment is not a matter of right and wrong *per se*, but *intelligence* often is such a matter.

In judgment, we make a choice based on what we think is the best or most appropriate thing to do or believe, and if there is a choice, then there is a ranking of the choices as evidenced by the result of the judgment. How different a thing than intelligence where the right decision is often a unique item.[2]

9.5 A Commitment to Nonsense

The measures associated with the descriptors in the situation description do *not* correspond to probabilities, since the latter have a firm connection with *truth* and *fact*, in that a probability, p, means the likelihood that some thing is true. In this system, I mean by a measure, p, the amount of commitment to a fact with respect to other possibilities. That is, one can state for any two entries which one the system has more commitment to, but these measures are *subjective to Yh*!

The situation description that is used can easily represent the *fact* that an object is believed to be at point x and at point y, where $x \neq y$. Of course the measures of the two facts may not be equal, and they may not be absolute, but nevertheless there is a good sense in which the data base reflects an *inconsistent* fact.

To many systems and to many people this is intolerable: it means that we can derive anything! However, this can only be a problem in systems where it is a priori true that this is a problem. In my system, there is some numeric measure of a fact beyond which it is willing to commit itself to action on that validity of that fact. In the Blind Robot problem, the fact of contradictory statements in the data base is used to switch the tide of belief from some objects being at some locations to some other locations. And since there is no observational facility to determine which facts are true, manipulating these contradictory facts is a good method for dealing with the situation.

The rationality of this system of representation can be seen readily when you forget about facts and absolute systems and think about generation as the paradigmatic problem. Here you have to worry about whether you have been able to express something adequately. So, in the first reference you might be able to express some fraction of the full fact, measured subjectively. But it takes a second reference to complete the thought. So if you wanted to talk about an array, but didn't want to have very long noun phrases, you could say "The zero-based, one-dimensional array. . .R is initialized to the last element, 27. . .." Here the first reference didn't say the length of the array, so a future reference to it detailing this fact is made.

The representation and description systems used in Yh are very good for this sort of thing.

[2] This is not to say that there isn't some shading of intelligence to judgment, just that there is some dimension between the two.

9.6 Bondage and Influences

The idea of programming a system to demonstrate interesting behavior, such as generating stylistically pleasing English, is a funny sort of thing in that the operant word in programming languages is *imperative*, even in applicative languages. This means that the exact activities of the program are essentially dictated by the programmer using unambiguous instructions with conditionals. Of course the exact program behavior may depend on outside input through sensors, but basically the normal style of programming is like writing an algorithm.

I hope you have noticed that although certain parts of Yh are like this, such as most of the procedural part of the units, there is a strong sense of leaving control up to a search or pattern matching process, which takes into account the entire current situation. As I stated in Chapter 5 there are several methods of sequencing in Yh: the normal applicative/imperative methods, the agenda, and the situation description.

Of these, the first fairly explicitly determines control without much doubt about what is to happen. The agenda mechanism is a step away from this philosophy in that the things that need to be done are put on a list which has some priorities or other predicates that are used to determine when and if an action is to take place. Thus the unavoidability of what to do next is taken away to the extent that the final course of things can be altered easily by changing the agenda priorities.

For example, suppose that we start out in a traditional program to do some things; except for conditional control based on variables (or whatever) that are set as time goes on, the flow of control is determined at code creation time. In an agenda-based system one part of the program can say to do some things and a later part can countermand that.

Now, I don't think I need to say that there is no gain in theoretical power from one type of control to the next, but there is a gain in expressive power and a shift in the way we look at programming.

With influence-based programming (my jargon) there is a description of the supposedly relevant facts and goals, which form the core of what is to be done in the near future. In addition, there are influences and soft-constraints which may affect what gets done, depending on the distinctions that the descriptions of the participants provide. The choice of what things get done is up to a matching process which locates the individual who satisfies the most needs the best at the moment.

Programming in this environment is more like saying things such as: *These things need to be done, keep these other things in mind, and I'd prefer to see these conditions true.* So the flavor is that of suggesting and manipulating the description of the situation to reflect the best idea of what needs to be done.

I feel that normal programming is simply taking one's own general description of what has to be done and finding a sequence of actions, data structures, etc., in the language which accomplishes this. So in assembly language, if I want to get the quotient of one number by another, which is very important, and I also later want the remainder, which is not so important, I eventually find the instruction that does both (IDIV on the PDP10).

In the generation that was shown as an example there was an admonition to not use too many adjectives. This was expressed as an influence to the entire system. I could have programmed it to check at every place for the value of some variable, but instead I can just express that admonition itself along with how strongly I want it to be enforced, and when the descriptions of the units that do contain the distinctions necessary to recognize this admonition are present, everything is taken care of. In some sense, I give advice to Yh about how it should act, and that is how I control its style and its creative capabilities.

9.6.1 Counterinduction and Rationality

Counterinduction is a search technique predicated on the commitment to preferring variety and exploration to correctness, which is not a particularly important issue (in generation anyway). Counterinduction was

carried out several times in the example given in Chapter 8, but mainly for the purpose of recalculating the worth of trying some course of action.

Counterinduction is a source of information gain in the system in that it allows the program to examine the admittedly ad hoc measures that are used in the descriptions.

9.7 Cult of the Superintelligent

I have mentioned earlier that the problem with most AI research to date is that there has been too much focus on *intelligence* and too little on *judgment* or on anything else that is not associated with doing things that intelligent people do. I am not interested in writing a program that cleverly solves puzzles, figures out how to get to the airport with potatoes in the way, plays chess, makes medical diagnoses, speaks like Oliver Wendell Holmes, or otherwise is better at something than 95% of the population of the world. Of course things like inverting 500 by 500 matrices programs can do better than most people, but I certainly don't have any prejudice in favor of machines over people.

My point in writing this program is to make something of an introspectively accurate (to me) model for obtaining the sort of behavior in deliberate writing that I think occurs with normal human writers. I wanted to be able to at least express the things I thought were important and to be able to give advice to the program and have that advice acted upon, in ways that I think are similar to the way people take advice and act.

Catch phrases such as *knowledge-based*, *intelligent*, *planning*, and *rule-based* have little place in the fundamental process of understanding the mind, and to worry about creating idiot savants seems to be a misdirected goal to me. If AI is taking the idea that the mind is like a computer in some ways, and if the idea of what a computer is like *must* remain as it has for decades, and if the current antiquated ideas of computing must form the basis of understanding the mind, then I'm afraid I fall outside of the realm of AI for the moment, since I think that we must redefine computation and expand our horizons as failures and boundaries of our abilities are encountered.

9.8 The Final Curtain

Yh does a fair job of generating English about a small class of programs, but it is not a production level program. This small class of programs could be better explained with some ad hoc program for exactly that purpose. Yh does manage to demonstrate interesting behavior in exploring the courses of action available for expressing the program in question. It is possible to program Yh in ways that correspond more closely to way that people are given advice, which means that we can begin to converse with these programs—though not in English—in ways that are natural for both rather than in ways that are like a surgeon (the programmers) operating on a patient (the programs), which is the case at present.

Yh represents a methodological shift away from standard kinds of programming techniques and assumptions towards ones which provide an opportunity to interact with our programs more freely. I hope that this thesis will allow people to stop being obsessed with puzzles and correct solutions and to get on with the business of understanding ordinary, everyday minds, which is the first step in creating colleagues rather than slaves.

References

[Appelt 1980] Appelt, D., *Problem Solving Applied to Language Generation* in **Proceedings of the 18th Annual Meeting of the ACL**, Philadelphia, June 1980.

[Berliner 1980] Berliner, Hans, **Some Observations on Problem Solving** CMU-CS-80-113, CMU, April 1980.

[Bobrow 1977] Bobrow, Daniel G.; Winograd, Terry, *An Overview of KRL, a Knowledge Representation Language* in **COGNITIVE SCIENCE**, Vol. 1, No. 1, 1977.

[Buchanan 1969] Buchanan, B.; Sutherland; G., Feigenbaum, E. A., *Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry*, in **Machine Intelligence 4**, Metzler, B. and Michie, D., eds., American Elsevier, New York, 1969.

[Cohen 1978] Cohen, P. R., **On Knowing What to Say: Planning Speech Acts**, Technical Report #118, University of Toronto, Toronto, Canada, 1978.

[Dreyfus 1979] Dreyfus, Hubert L., **What Computers Can't Do**, Harper Colophon Books, Harper and Row, Publishers, 1979.

[Eliot 1930] Eliot, T. S., *The Love Song of J. Alfred Prufrock* in **Collected Poems**, Harcourt, Brace, & World, Inc., 1970.

[Ernst 1969] Ernst, G,; Newell, A., **GPS: A Case Study in Generality and Problem Solving**, Academic Press, New York, 1969.

[Feigenbaum 1971] Feigenbaum, E. A.; Buchanan, B. G.; Lederberg, J., *Generality and Problem Solving: A Case Study using the DENDRAL Program* in **Machine Intelligence 16**, Metzler, B. and Michie, D., eds., American Elsevier, New York, 1971.

[Feigenbaum 1980] Feigenbaum, E. A., **Knowledge Engineering: The Applied Side of Artificial Intelligence**, STAN-CS-80-812 (HPP-80-14), Stanford, 1980.

[Feyerabend 1975] Feyerabend, Paul, **Against Method**, Verso Editions, 1975.

[Fillmore 1968] Fillmore, C. J., *The Case for Case* in **Universals in Linguistic Theory**, Bach and Harms, eds., Holt Rinehart, and Winston, New York, 1968.

[Filman 1976] Filman, Robert; Weyhrauch, Richard, **An FOL Primer**, STAN-CS-76-572 (AIM-288), Stanford, September 1976.

[Friedman 1969a] Friedman, J., *Directed Random Generation of Sentences* in **Communications of the ACM**, Vol. 12, No. 1, January 1969.

[Friedman 1969b] Friedman, J., *A Computer System for Transformation Grammar* in **Communications of the ACM**, Vol. 12, No. 6, June 1969.

[Gabriel 1980] Gabriel, Richard P., **An Organization for Programs in Fluid Domains**, Dissertatation, Stanford University, December 1980.

[Goldman 1974] Goldman, Neil, **Computer Generation of Natural Language from a Deep Conceptual Base** STAN-CS-74-461 (AIM 247), Stanford, January 1974.

[Green 1977] Green, Cordell, *A Summary of the PSI Program Synthesis System* in **5th International Joint Conference on Artificial Intelligence—1977**, Cambridge, Mass, 1977.

[Halliday 1976] Halliday, M. A. K.; Hasan, Ruqaiya, **Cohesion in English**, Longman Group Limited, London, 1976.

[Hewitt 1972] Hewitt, Carl, **Description and Analysis (using Schemata) of Planner: A language for proving theorems and manipulating models in robots**, MIT TR-258, MIT, 1972.

[Jackson 1974] Jackson Jr., Philip C., **Introduction to Artificial Intelligence**, Petrocelli Books, New York, 1974.

[Klein 1965a] Klein, S., *Automatic Paraphrasing in Essay Format* in **Mechanical Translation**, Vol. 8, No. 3, June, 1965.

[Klein 1965b] Klein, S., *Control of Style with a Generative Grammar* in **Language**, Vol. 41, No. 4, December 1965

[Kowalski 1974] Kowalski, Robert, *Predicate Calculus as Programming Language*, in **Proc. IFIP 74**, North Holland Publishing Co., Amsterdam, 1974.

[Levin 1978] Levin, J. A.; Goldman, Neil, **Process Models of Reference in Context**, USC/Information Sciences Institute, Research report 78-72, 1978.

[Mann 1980] Mann, William C., Moore, James A., **Computer as Author—Results and Prospects**, USC/Information Sciences Institute, Research report 79-82, January 1980.

[McCarthy 1977] McCarthy, J., *Another SAMEFRINGE* in **SIGART Newsletter**, No. 61, February 1977.

[McCarthy 1980] McCarthy, J., **Circumscription—A Form of Non-Monotonic Reasoning**, STAN-CS-80-788 (AIM-334, AD-A086 574), February 1980.

[McDonald 1979] McDonald, David, **Steps toward a Psycholinguistic Model of Language Production**, MIT AI Lab Working Paper 193, MIT, April 1979.

[McKeown 1980] McKeown, Kathleen R., *Generating Relevant Explanations: Natural Language Responses to Questions about Database Structure* in **Proceedings of the First Annual National Conference on Artificial Intelligence**, August 1980.

[Maturana 1970] Maturana, Humberto R., **Biology of Cognition**, B.C.L. Report No. 9.0, UILU-ENG-70-300, November 1970.

[Meehan 1976] Meehan, James, **The Metanovel: Writing Stories by Computer**, Yale Research Report #74, September 1976.

[Melville 1851] Melville, Herman, **Moby Dick**, W. W. Norton & Co., Inc, New York, 1967.

[Michie 1974] Michie, Donald, **On Machine Intelligence**, John Wiley & Sons, New York, 1974.

[Minsky 1975] Minsky, Marvin, *A Framework for Representing Knowledge* in **The Psychology of Computer Vision**, Patrick Henry Winston, Ed., McGraw-Hill Book Company, New York, 1975.

[Minsky 1977] Minsky, Marvin, *Plain Talk about Neurodevelopmental Epistemology* in **5th International Joint Conference on Artificial Intelligence—1977**, Cambridge, Mass, 1977.

[Moravec 1981] Moravec, Hans P., *The Endless Frontier and The Thinking Machine*, in **The Endless Frontier, Vol. 2**, Jerry Pournelle, ed., Grosset & Dunlap, Ace Books, 1981. (to appear)

[Roberts 1977] Roberts, B. R.; Goldstein, I. P., **The FRL Manual**, MIT AI Memo 409, MIT, September, 1977.

[Ryle 1949] Ryle, Gilbert, **The Concept of Mind**, Barnes & Noble Books (Harper & Row, Publishers), 1949.

[Sacerdoti 1977] Sacerdoti, Earl, **A Structure for Plans and Behavior**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1977.

[Schank 1977] Schank, Roger; Abelson, Robert, **Scripts, Plans, Goals, and Understanding**, Lawrence Erlbaum Associates, New Jersey, 1977.

[Shortliffe 1974] Shortliffe, Edward H., **MYCIN: A Rule-based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection**, STAN-CS-74-465, Stanford, October 1974.

[Simmons 1972a] Simmons, R.; Slocum, J., *Generating English Discourse from Semantic Networks* in **Communications of the ACM**, Vol. 15, No. 10, October, 1972.

[Simmons 1973] Simmons, R., *Semantic Networks: Their Computation and use for Understanding English Sentences* in **Computer Models of Thought and Language**, Schank and Colby, eds., San Francisco, 1973.

[Simmons 1972b] Simmons, R., *Some Semantic Structures for Representing English Meanings* in **Language Comprehension and the Acquisition of Knowledge**, Freedle and Carrol, eds., V. H. Winston and Sons. Washington, 1972.

[Simon 1969] Simon, Herbert A., **The Sciences of the Artificial**, MIT Press, Cambridge, 1969.

[Stefik 1980] Stefik, Mark Jeffrey, **Planning with Constraints**, STAN-CS-80-784, January 1980.

[Sussman 1972] Sussman, Gerald J.; McDermott, Drew V., **Why Conniving is Better than Planning**, MIT AI Memo 255a, MIT, 1972.

[Sussman 1973] Sussman, Gerald J., **A Computational Model of Skill Acquisition**, AI-TR-297, MIT, August 1973.

[Thomas 1938] Thomas, Dylan, *A Grief Ago* in **Collected Poems**, New Directions Publishing, 1971.

[Thomas 1974] Thomas, Lewis, *The Lives of a Cell* in **The Lives of a Cell**, Bantam Books, Inc., 1974.

[Tracy 1979] Tracy, Kathleen B.; Montague, Elaine C.; Gabriel, Richard P.; Kent, Barbara E., *Computer-Assisted Diagnosis of Orthopedic Gait Disorders* in **PHYSICAL THERAPY**, Vol. 59, No. 3, March 1979.

[Varela 1979] Varela, Francisco J., **Principles of Biological Autonomy**, North Holland, 1979.

[Weyhrauch 1974] Weyhrauch, Richard; Thomas, Arthur, **FOL: A Proof Checker for First-order Logic**, STAN-CS-74-432 (AIM 235), Stanford, September 1974.

[Weyhrauch 1978] Weyhrauch, Richard, **Prolegomena to a Theory of Formal Reasoning**, STAN-CS-78-687 (AIM 315), Stanford, December 1978.

[Winograd 1972] Winograd, T., **Understanding Natural Language** Academic Press, New York, 1972.

[Winograd 1980] Winograd, T., *Extended Inference Modes in Reasoning by Computer Systems* in **Artifical Intelligence**, Vol. 13, 5-26, 1980.

[Woods 1970] Woods, W., *Transition Network Grammars for Natural Language Analysis* in **Communications of the ACM**, Vol. 13, No. 10, October 1970.

[Yngve 1962] Yngve, V., *Random Generation of English Sentences* in **1961 International Conference on Machine Translation of Languages and Applied Language Analysis, Teddington**, Her Majesty's Stationery Office, London, 1962.