PART THREE

Building a Knowledge Base

7

Knowledge Engineering

From early experience building the DENDRAL system, it was obvious to us that putting domain-specific knowledge into a program was a bottleneck in building knowledge-based systems (Buchanan et al., 1970). In other AI systems of the 1960s and early 1970s, items of knowledge were cast as LISP functions. For example, in the earliest version of DENDRAL the fact that the atomic weight of carbon is 12 was built into a function, called WEIGHT, which returned *12* when called with the argument *C*. The function "knew about" several common chemical elements, but when new elements or new isotopes were encountered, the function had to be changed. Because we wanted to keep our programs "lean" to run in 64K of working memory, we gave our programs only as much knowledge as we thought they would have to know. Thus we often encountered missing items in running new test cases. It was very quickly seen that LISP property lists (data structures) were a superior alternative to LISP code as a way of storing simple facts, so definitions of functions like WEIGHT were changed to retrievals from property lists (using GETPROP's and macros). Defining new objects and properties was trivial in comparison to the overhead of editing functions. This was the beginning of our realization that there is considerable flexibility to be gained by separating domain-specific knowledge from the code that uses that knowledge. This was also our first encounter with the problem that has come to be known as knowledge acquisition (Buchanan et al., 1970).

7.1 The Nature of the Knowledge Acquisition Process

Knowledge acquisition is the transfer and transformation of problem-solving expertise from some knowledge source to a program. There are many

Section 7.1 is largely taken from material originally written for Chapter 5 of *Building Expert Systems* (eds., F. Hayes-Roth, D. Waterman, and D. Lenat). Reading, Mass.: Addison-Wesley, 1983.

sources we might turn to, including human experts, textbooks, data bases, and our own experience. In this section we will concentrate mostly on acquiring knowledge from human experts in an enterprise known as knowledge engineering (Hayes-Roth et al., 1983). These experts are specialists (but not necessarily unique individuals) in a narrow area of knowledge about the world. The expertise that we hope to elucidate is a collection of definitions, relations, specialized facts, algorithms, strategies, and heuristics about the narrow domain area. It is different from general knowledge about the domain and from commonsense knowledge about the world, some of which is also needed by expert systems.

A knowledge base for an expert system is constructed through a process of iterative development. After initial design and prototype implementation, the system grows incrementally both in breadth and depth. While other large software systems are sometimes built by accretion, this style of construction is inescapable for expert systems because the requisite knowledge is impossible to define as one complete block.

One of the key ideas in constructing an expert system is *transparency*—making the system understandable despite the complexity of the task. An expert system needs to be understandable for the following reasons:

- the system matures through incremental improvements, which require thorough understanding of previous versions and of the reasons for good and poor performance on test cases;
- the system improves through criticism from persons who are not (or need not be) familiar with the implementation details;
- the system uses heuristic methods and symbolic reasoning because mathematical algorithms do not exist (or are inefficient) for the problems it solves.

7.1.1 Modes of Knowledge Acquisition

The transfer and transformation required to represent expertise for a program may be automated or partially automated in some special cases. Most of the time a person, called a knowledge engineer, is required to communicate with the expert and the program. The most difficult aspect of knowledge acquisition is the initial one of helping the expert conceptualize and structure the domain knowledge for use in problem solving. Because the knowledge engineer has far less knowledge of the domain than does the expert, by definition, the process of transferring expertise into a program is bound to suffer from communication problems. For example, the vocabulary that the expert uses to talk about the domain with a novice is probably inadequate for high-performance problem solving.

There are several modes of knowledge acquisition for an expert system, which can be seen as variations on the process shown in Figure 7-1.

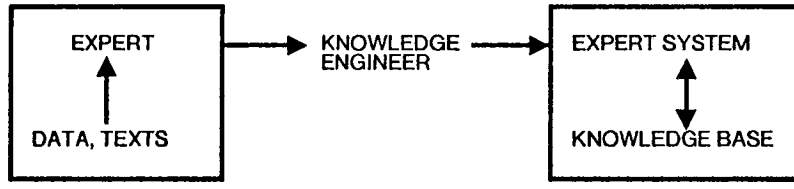

FIGURE 7-1 Important elements in the transfer of expertise. Feedback to the expert about the system's performance on test cases is not shown.

All involve transferring, in one way or another, the expertise needed for high-performance problem solving in a domain from a *source* to a *program.* The source is generally a human expert, but could also be the primary sources from which the expert has learned the material: journal articles (and textbooks) or experimental data. A knowledge engineer translates statements about the domain from the source to the program with more or less assistance from intelligent programs. And there is variability in the extent to which the knowledge base is distinct from the rest of the system.

Handcrafting

Conceptually, the simplest way for a programmer to put knowledge into a program is to code it in. This was the standard mode of building AI programs in the 1950s and 1960s because the main emphasis of most of those systems was demonstrating intelligent behavior for a few problems. AI programmers could be their own experts for many game-playing, puzzle-solving, and mathematics programs. And a few domain specialists became their own AI programmers in order to construct complex systems (Colby, 1981; Hearn, 1971). When the programmer and the specialist are not the same person, however, it is risky to rely on handcrafting to build complex programs embodying large amounts of judgmental knowledge. Generally, it is slow to build and debug such a program, and it is nearly impossible to keep the problem-solving expertise consistent if it grows large by small increments.

Knowledge Engineering

The process of working with an expert to map what he or she knows into a form suitable for an expert system to use has come to be known as *knowledge engineering* (Feigenbaum, 1978; Michie, 1973).

As DENDRAL matured, we began to see patterns in the interactions

between the person responsible for the code and the expert responsible for the knowledge. There is a dialogue which, at first, is much like a systems analysis dialogue between analyst and specialist. The relevant concepts are named, and the relations among them made explicit. The knowledge engineer has to become familiar enough with the terminology and structure of the subject area that his or her questions are meaningful and relevant. As the knowledge engineer learns more about the subject matter, and as the specialist learns more about the structure of the knowledge base and the consequences of expressing knowledge in different forms, the process speeds up.

After the initial period of conceptualization, in which most of the framework for talking about the subject matter is laid out, the knowledge structures can be filled in rather rapidly. This period of rapid growth of the knowledge base is then followed by meticulous testing and refinement. Knowledge-engineering *tools* can speed up this process. For example, intelligent editing programs that help keep track of changes and help find inconsistencies can be useful to both the knowledge engineer and the expert. At times, an expert can use the tools independently of the knowledge engineer, thus approaching McCarthy's idea of a program accepting advice from a specialist (McCarthy, 1958). The ARL editor incorporated in EMYCIN (see Chapters 14–16) is a simple tool; the TEIRESIAS debugging system (discussed in Chapter 9) is a more complex tool. Politakis (1982) has recently developed a tool for examining a knowledge base for the EXPERT system (Kulikowski and Weiss, 1982) and suggesting changes, much like the tool for ONCOCIN discussed in Chapter 8.

A recent experiment in knowledge engineering is the ROGET program (Bennett, 1983), a knowledge-based system that aids in the conceptualization of knowledge bases for EMYCIN systems. Its knowledge is part of what a knowledge engineer knows about helping an expert with the initial process of laying out the structure of a new body of knowledge. It carries on a dialogue about the relationships among objects in the new domain, about the goal of the new system, about the evidence available, and about the inferences from evidence to conclusions. Although it knows nothing (initially) about a new knowledge base, it knows something about the structure of other knowledge bases. For example, it knows that evidence can often be divided into "hard" evidence from instruments and laboratory analysis and "soft" evidence from subjective reports and that both are different from identifying features such as gender and race. Much more remains to be done, but ROGET is an important step in codifying the art of knowledge engineering.

Various Forms of "Learning"

For completeness, we mention briefly several other methods of building knowledge-based programs. We have not experimented with these in the context of MYCIN, so we will not dwell on them.

Learning from examples may automate much of the knowledge acquisition process by exploiting large data bases of recorded experience (e.g., hospital records of patients, field service records of machine failures). The conceptualization stage may be bypassed if the terminology of the records is sufficient for problem solving. Induction of new production rules from examples was used by Waterman (1970) in the context of the game of poker and in Meta-DENDRAL (Lindsay et al., 1980) in the context of mass spectrometry. The RX system (Blum, 1982) uses patient records to discover plausible associations.

Other methods of learning are discovery by exploration of new concepts and relations (Lenat, 1983), reading published accounts (Ciesielski, 1980), learning by watching (Waterman, 1978), and learning by analogy (Winston, 1979). See Buchanan et al. (1978) and Barr and Feigenbaum (1982) for reviews of automatic learning methods.

7.2 Knowledge Acquisition in MYCIN

In the MYCIN work we experimented with computer-based tools to acquire knowledge from experts through interactive dialogues. TEIRESIAS, discussed in Chapter 9, is the best-known example. In discussing knowledge acquisition, it is important to remember that there are separate programs under discussion: the expert system, i.e., MYCIN, and the programs that provide help in knowledge acquisition, i.e., TEIRESIAS.

As mentioned above, MYCIN itself was an experiment in keeping medical knowledge separate from the rest of the program. We believed that this would simplify knowledge acquisition, and it does, but not to the extent we had hoped. Because the syntax of the elements carrying knowledge was simplified, however, our focus shifted from the *mechanics* of editing those elements to the *contents* of those knowledge structures. That is, there was an important conceptual shift from thinking of editing data structures to thinking of modifying knowledge structures; we have come to call the latter process *knowledge programming.*

The processes of constructing and editing a knowledge base became interesting subjects of our research. We could see that the communication between expert and program was very slow. So we began investigating computer-based tools that would facilitate the transfer of expertise. In the original version of MYCIN, there were some tools for helping Shortliffe, as knowledge engineer, build and modify the infectious disease knowledge base. No attempt was made to get experts to use the tools directly, although that was clearly a next step. These first tools included a rule language (syntax and parser) that allowed entering a new rule in a quasi-English form. In the example shown in Figure 7-2, the user indicates a desire to enter a new rule by typing NR. He or she is then asked for a rule in English,

```
**NR                                                    [The knowledge engineer starts
The new rule will be called RULE200.                    the rule acquisition routine by
     If:  1-** THE ORGANISM IS A GRAM NEGATIVE ROD      typing NR for New Rule.]
     and  2-** IT IS ANAEROBIC
     and  3-** IT WAS ISOLATED FROM THE BLOOD
     and  4-** YOU THINK THE PORTAL WAS THE GI TRACT
     and  5-**
                                                        [user: carriage return with no
                                                        entry]
     Then: 1-** IT IS PROBABLY A BACTEROIDES
           On a scale of 1 to 10, how much certainty would
           you affix to this conclusion?
           ** 9
     and   2-**
                                                        [user: carriage return with no
                                                        entry]
This is my understanding of your rule:
RULE200

     IF: 1) The site of the culture is blood, and       [Note that the original clause 1
         2) The stain of the organism is gramneg, and    has been expanded to separate
         3) The morphology of the organism is rod, and   the two attributes, stain and
         4) The aerobicity of the organism is anaerobic, and  morphology.]
         5) The portal of entry of the organism is GI
     THEN: There is strongly suggestive evidence (.9) that the
           organism is bacteroides

Okay? (YES or NO)
** YES
```

FIGURE 7-2 Example of rule acquisition in the original (1974) MYCIN program. (User's input follows double asterisks.)

following the format of other rules in the system. MYCIN translates the rule into its internal LISP representation and then translates it back into English to print out a version of the rule as it has understood the meaning. The user is then asked to approve the rule or modify it. The original system also allowed simple changes to rules in a quick and easy interaction, much as is shown in Figure 7-2 for acquiring a new rule.

This simple model of knowledge acquisition was subsequently expanded, most notably in the work on TEIRESIAS (Chapter 9). Many of the ideas (and lines of LISP code) from TEIRESIAS were incorporated in EMYCIN (Part Five). Contrast Figure 7-2 with the TEIRESIAS example in Section 9.2 and the EMYCIN example in Chapter 14 for snapshots of our ideas on knowledge acquisition. Research on this problem continues.

Two of our initial working hypotheses about knowledge acquisition have had to be qualified. We had assumed that the rules were sufficiently independent of one another that an expert could always write new rules without examining the rest of the knowledge base. Such modularity is desirable because the less interaction there is among rules, the easier and safer it is to modify the rule set. However, we found that some experts are

1. Expert tells knowledge engineer what rules to add or modify.
2. Knowledge engineer makes changes to the knowledge base.
3. Knowledge engineer runs one or more old cases for consistency checking.
4. If any problems with old cases, knowledge engineer discusses them with expert, then goes to Step 1.
5. Expert runs modified system on new case(s) until problems are discovered.
6. If no problems on substantial number of cases, then stops; otherwise, goes to Step 1.

FIGURE 7-3 The major steps of rule writing and refinement after conceptualization.

helped if they see the existing rules that are similar to a new rule under consideration, where similar means either that the *conclusion* mentions the same parameter (but perhaps different values) or that the *premise* clauses mention the same parameters. The desire to compare a proposed rule with similar rules stems largely from the difficulty of assigning CF's to new rules. Comparing other evidence and other conclusions puts the strength of the proposed rule into a partial ordering. For example, evidence *e1* for conclusion *C* could be seen to be stronger than *e2* but weaker than *e3* for the same conclusion. We also assumed, incorrectly, that the control structure and CF propagation method were details that the expert could avoid learning. That is, an expert writing a new rule sometimes needs to understand how the rule will be used and what its effect will be in the overall solution to a problem. These two problems are illustrated in the transcripts of several electronic mail messages reprinted at the end of Chapter 10. The transcripts also reveal much about the vigorous questioning of assumptions that was taking place as rules were being written.

Throughout the development of MYCIN's knowledge base about infectious diseases (once a satisfactory conceptualization for the problem was found), the primary mode of interaction between the knowledge engineer and expert was a recurring cycle as shown in Figure 7-3. Much of the actual time, particularly in the early years, was spent on changes to the code, outside of this loop, in order to get the system to work efficiently (or sometimes to work at all) with new kinds of knowledge suggested by experts. Considerable time was spent with the experts trying to understand their larger perspective on diagnosis and therapy in infectious disease. And some time was spent trying to reconceptualize the program's problem-solving framework. We believed that the time-consuming nature of the six-step loop shown in Figure 7-3 was one of the key problems in building an expert system, although the framework itself was simple and effective. Thus we looked at several ways to improve the expert's and knowledge engineer's efficiency in the loop.

For Step 1 of the loop we created facilities for experts (or other users)

to leave comments for the knowledge engineers. We gave them an English-like language for describing new relationships. And we created the explanation facility described in Part Six, so they could understand a faulty line of reasoning well enough to correct the knowledge base. For Step 2, as mentioned, we created tools for the knowledge engineer, to facilitate entry and modification of rules. For Step 3, we created an indexed library of test cases and facilities for running many cases in batch mode overnight. For Step 4, the batch system recorded differences caused by a set of modifications in the advice given on the test cases. The record was then used by the knowledge engineer to assess the detrimental effects, if any, of recent changes to the rules. Some of our concern with human engineering, discussed in Part Eleven, was motivated by Step 5 because we realized the necessity of an expert's "playing with" the system in order to discover its weaknesses.

The TEIRESIAS system discussed in Chapter 9 was the product of an experiment on interactive transfer of expertise. TEIRESIAS was designed to help an expert at Steps 1, 2, and 5. Although the program was never used routinely in its entirety by collaborating infectious disease specialists, we considered the experiment to be highly successful. It showed the power of using a model of the domain-specific knowledge with syntactic editors. It showed that debugging in the context of a specific case is an effective means to focus the expert's attention. TEIRESIAS analyzed a rule set statically to build rule models, which, in turn, were used during the dynamic debugging. It thus "knew what it knew," that is, it had models of the knowledge base. It used the rule models to provide advice about incomplete areas of the knowledge base, to provide suggestions and help during interactive debugging sessions and to provide summary explanations. Much of TEIRESIAS is now embedded in the knowledge acquisition code of EMYCIN.

The rule checker discussed in Chapter 8 was an experiment in static analysis of a rule set, in contrast to TEIRESIAS' dynamic analysis in context. It was not a large project, but it does demonstrate the power of analyzing a rule set for the expert. Its analysis of rules is simpler than TEIRESIAS' static analysis for two reasons: the rules it considers all make conclusions with certainty (i.e., CF = 1); and the clusterings of rules are easier to identify as a result of an extra slot attached to each rule naming the context in which it applies. It analyzes rules for the ONCOCIN system, described in more detail in Chapters 32 and 35.

As we had believed from the start, the kind of analysis performed by the rule checker provides helpful information to the expert writing new rules. To some extent, it is orthogonal to the six-step interactive loop mentioned above, but it might also be seen as Step 2a between entering a set of changes and running test cases. After the expert adds several new rules (through the interactive loop or not), the rule checker will point out logical problems of inconsistency and subsumption and pragmatic problems of redundancy and incompleteness. Any of these is a signal to the expert to

examine the subsets of rules in which the rule checker identifies problems. Because this analysis is more systematic than the empirical testing in Steps 3–5 of the six-step loop, it can catch potential problems long before they would manifest themselves in test cases.

Some checking of rules is also done in EMYCIN, as described in the EMYCIN manual (van Melle et al., 1981). As each rule is entered or edited, it is checked for syntactic validity to catch common input errors. By syntactic, we mean issues of rule form—viz., that terms are spelled correctly, values are legal for the parameters with which they are associated, etc.—rather than the actual information (semantic) content (i.e., whether or not the rule "makes sense"). Performing the syntactic checks at acquisition time reduces the likelihood that the consultation program will later fail due to "obvious" errors. This permits the expert to concentrate on debugging logical errors and omissions.

The purely syntactic checks are made by comparing each rule clause with the internal *function template* corresponding to the predicate or action function used in the clause. Using this template, EMYCIN determines whether the argument slots for these functions are correctly filled. For example, each argument requiring a parameter must be assigned a valid parameter (of some context), and any argument requiring a value must be assigned a legal value for the associated parameter. If an unknown parameter is found, the checker tries to correct it with the Interlisp spelling corrector, using a spelling list of all parameters in the system. If that fails, it asks if this is a new (previously unmentioned) parameter. If so, it defines the new parameter and, in a brief diversion, prompts the system builder to describe it. Similar action is also taken if an unrecognized value for a parameter is found.

A limited semantic check is also performed: each new or changed rule is compared with any existing rules that conclude about the same parameter to make sure it does not directly contradict or subsume any of them. A contradiction occurs when two rules with the same set of premise clauses make conflicting conclusions (contradictory values of CF's for the same parameter); subsumption occurs when one rule's premise is a subset of the other's, so that the first rule succeeds whenever the second one does (i.e., the second rule is more specific), and both conclude about the same values. In either case, the interaction is reported to the expert, who may then examine or edit any of the conflicting or redundant rules.

Another experimental system we incorporated into MYCIN was a small body of code that kept statistics on the use of rules and presented the statistical results to the knowledge base builders.[1] It provided another way of analyzing the contents of a knowledge base so potential problems could be examined. It revealed, for example, that some rules never succeeded, even though they were called many times. Even though their con-

[1]This code was largely written by Jan Aikins.

clusions were relevant (mentioned a subgoal that was traced), their premise conditions never matched the specific facts of the cases. Sometimes this happens because a rule is covering a very unusual set of circumstances not instantiated in the test cases. Since much expertise resides in such rules, we did not modify them if they were in the knowledge base for that reason. Sometimes, though, the lack of successful invocation of rules indicated a problem. The premises might be too specific, perhaps because of transcription errors in premise clauses, and these did need attention. This experimental system also revealed that some rules *always* succeeded when called, occasionally on cases where they were not supposed to. Although it was a small experiment, it was successful: empirically derived statistics on rule use can provide valuable information to the persons building the knowledge base.

One of the most important questions we have been asking in our work on knowledge acquisition is

> How (or to what extent) can an intelligent system replace a knowledge engineer in helping an expert build a knowledge base?

The experimental systems we have written are encouraging in pointing toward automated assistance (see Chapter 16), but they are far from a definitive solution. We have built tools for the knowledge engineer more readily than for the expert. In retrospect we now believe that we underestimated both the intellectual effort involved in building a good knowledge base and the amount of global information about the expert system that the expert needs to know.