



Tools for rule-based program development
by Michael David Turner

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in
Computer Science
Montana State University
© Copyright by Michael David Turner (1988)

Abstract:

Software has been developed for use here at Montana State University as a set of tools to help future students and researchers incorporate their knowledge about given domains of expertise into computer programs. All of the source code has been written in Common Lisp and tested on a Tektronix 4406 workstation. The tools are now available for use by students in artificial intelligence for the development of rule-based programs.

Four features have been designated as critical for determining the usefulness of such software tools. First of all, it should be easy to learn how to use the tools. Also, the tools themselves should not introduce inefficiencies into the problem solving process. They should provide means for producing, upon request, a clarification of the solution obtained for a given problem. Finally, the tools should be general enough to permit their applications to problems of diverse natures.

In a preliminary appraisal of the usefulness of this software, strong points emerge as do potential directions for further work. The formalism for expressing rules is very simple, and effective measures have been taken to prevent loss of efficiency during the search for relevant knowledge. Extensions can be made to provide for greater generality.

**TOOLS FOR RULE-BASED PROGRAM
DEVELOPMENT**

by

Michael David Turner

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

**MONTANA STATE UNIVERSITY
Bozeman, Montana**

May 1988

N378
T8555

APPROVAL

of a thesis submitted by

Michael David Turner

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

May 19, 1988
Date

Rockford J. Ross
Chairperson, Graduate Committee

Approved for the Major Department

May 18th 1988
Date

J. Dwight Starling
Head, Major Department

Approved for the College of Graduate Studies

June 6, 1988
Date

Henry J. Parsons
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature

Michael David Turner

Date

May 19, 1988

TABLE OF CONTENTS

| | Page |
|---|------|
| LIST OF FIGURES | vi |
| ABSTRACT | vii |
| 1. INTRODUCTION | 1 |
| Objective | 3 |
| Approach | 4 |
| 2. PROBLEMS FOR RULE-BASED PROGRAMMING | 6 |
| The Continental Divide Problem | 8 |
| A Machine Translation Problem | 15 |
| Other Problem Types | 23 |
| 3. THE LIFE CYCLE OF A RULE-BASED SYSTEM | 27 |
| 4. THE BACKWARD CHAINING SHELL | 31 |
| Files of Rules | 32 |
| Bacchus: A Language for BACKward CHaining | 34 |
| The Augmentation Phase of Rules File Building..... | 42 |
| Fact Set Encoding | 49 |
| The End User's Interactive Session | 50 |
| 5. THE CONDITION ACTION SHELL | 58 |
| The Rules File | 59 |
| Cactus: A Language for Condition-ACTION Rules | 60 |
| The Augmentation Phase | 66 |
| Fact Set Encoding | 70 |
| The Solution as a Data Structure | 70 |
| Interactive Sessions with an End User | 72 |
| 6. DIRECTIONS FOR THE FUTURE | 75 |
| BIBLIOGRAPHY | 79 |

TABLE OF CONTENTS - Continued

| | |
|---|----|
| APPENDICES | 81 |
| Appendix A - Access to the Software | 82 |
| Appendix B - User's Guide to the Software Tools | 85 |
| Building a File of Rules for the Backward Chaining Shell | 86 |
| Building a File of Facts | 88 |
| Running the Backward Chaining Shell | 90 |
| Building a Rules File for the Condition Action Shell | 92 |
| Running the Condition Action Shell | 94 |
| Appendix C - Sample Bacchus File | 95 |
| Appendix D - Sample Cactus File | 99 |

LIST OF FIGURES

| Figure | Page |
|---|------|
| 1. Examples of Rules | 7 |
| 2. Verb Features and Their Values | 17 |
| 3. Extended BNF Productions for Bacchus | 38 |
| 4. Simple List and Indexed List | 46 |
| 5. WHYNOT Option Output | 56 |
| 6. Extended BNF Productions for Cactus | 62 |
| 7. Options Presented during Session with End User | 91 |
| 8. Bacchus Rules for the Continental Divide Problem | 96 |
| 9. Cactus Rules for the Machine Translation Problem | 100 |

ABSTRACT

Software has been developed for use here at Montana State University as a set of tools to help future students and researchers incorporate their knowledge about given domains of expertise into computer programs. All of the source code has been written in Common Lisp and tested on a Tektronix 4406 workstation. The tools are now available for use by students in artificial intelligence for the development of rule-based programs.

Four features have been designated as critical for determining the usefulness of such software tools. First of all, it should be easy to learn how to use the tools. Also, the tools themselves should not introduce inefficiencies into the problem solving process. They should provide means for producing, upon request, a clarification of the solution obtained for a given problem. Finally, the tools should be general enough to permit their applications to problems of diverse natures.

In a preliminary appraisal of the usefulness of this software, strong points emerge as do potential directions for further work. The formalism for expressing rules is very simple, and effective measures have been taken to prevent loss of efficiency during the search for relevant knowledge. Extensions can be made to provide for greater generality.

CHAPTER 1

INTRODUCTION

Expert systems have been the focus of a great deal of interest in recent years. Indeed, the interest extends well beyond the computer science community. Hayes-Roth et al. claim that expert systems represent the one area within artificial intelligence which has a record of successful applications (Hayes-Roth, 1983). Actually, notable successes in other areas of AI preceded the dawn of expert systems. Samuel's checkers program, for instance, showed that computers could "learn" to play games and play them well (Samuel, 1959). It was not until the development of expert systems, however, that the dazzling commercial and scientific potential of AI applications was made apparent through such achievements, for instance, as the discovery by a computer program of mineral deposits valued in excess of \$100 million (Hayes-Roth, 1983). Expert system technology has been greeted, not surprisingly, with enthusiasm and high expectations.

These expectations must nevertheless be tempered with the realization that building an expert system is no simple undertaking. Experience shows that five to ten person-years are required to build an expert system to solve an even moderately complex problem, and personnel with the competence to design and develop a system are scarce (Waterman, 1986).

The desirability of expert systems and the difficulty currently associated with building them combine to provide strong motivation for finding ways to make the job easier. Various tasks in the development of an expert system can be expedited by software tools, and the quality (and appropriate selection) of such tools can have profound impact.

Expert system building tools perform a wide gamut of operations. The programming language in which the expert system is developed itself constitutes an indispensable tool. Tools aid in eliciting knowledge from human experts and in representing that knowledge. Tools may aid in specifying the design of the expert system. They also provide some specialized support for testing, maintenance, and run-time convenience. This support may include aid in debugging, clarification, knowledge base editing and input/output facilities (Waterman, 1986).

Programming languages in which expert systems are implemented vary widely. They may be all-purpose languages such as Lisp or even C. They may, on the other hand, be highly specialized "knowledge engineering" languages. This latter type includes, for instance, EMYCIN and KAS. Their development involved removing the domain-specific knowledge from existing expert systems (Waterman, 1986). The presumption is that the knowledge base of an expert system, which is specific to the domain of application, ought to be separable from the "inference engine" (which simulates reasoning) and the user interface of the system. A system without a knowledge base can be referred to as a "skeletal system" or an expert system "shell". In practice a

generic shell does not tend to work well with a variety of knowledge bases unless the domains of application are closely related (Bratko, 1986). Although skeletal systems provide structure and built-in support which can expedite system development, they seem to lack flexibility. General-purpose programming languages, on the other hand, provide generality without support facilities. Thus, the development of a good knowledge engineering language involves striking the proper balance between high-level support and generality.

This same balance is to be sought in all expert system building tools. They must be general enough to permit their applications to problems of diverse natures, but at the same time, they need to be powerful in order to generate real savings in expert system development effort. Of course, they must be easy to use, and they must be easy to learn how to use. A powerful tool, after all, is of no use if it is hopelessly mysterious. Flexibility, power and ease of use are among the general requirements placed on software tools during expert system development.

If a tool is to be incorporated into the eventual finished system (as is the case with shells), it must not provoke inefficiency in system performance, but should enhance the clarity and helpfulness of the interface with the end user.

Objective

The goal of the research to be discussed herein has been to prepare a set of software tools for use here at Montana State University. Their use could be pedagogical: to help AI students gain

familiarity with rule-based programming. They might also see use in the building of a rule-based expert system in house.

The tools ought to include, among other things, an inference engine, a clarification facility, and, most especially, a helpful interface with the expert system design team which would facilitate the incorporation of knowledge into the emerging system.

The criteria for evaluating these tools are the same as those outlined above: they have to expedite the system development process; they have to be flexible; they have to be easily mastered.

Approach

How might one go about designing a software tool so that it will enjoy maximum generality? Is there such a thing as a "general problem", or do the problems that artificial intelligence is called upon to solve so differ intrinsically that they "force the hand" of algorithm designers into separate approaches?

A lesson to be drawn from the history of AI is that we should be cautious about glossing over the differences among types of problems. "AI scientists tried to simulate the complicated process of thinking by finding general methods for solving broad classes of problems.... This strategy produced no breakthroughs.... The more classes of problems a single program could handle, the more poorly it seemed to do on any individual problem" (Waterman, 1986).

The approach adopted here to the problem of developing general tools has been to specify toy problems which by their very nature exhibit strident differences. Specialized tools are then designed to

deal with the peculiar needs of a given model problem. After several such problems have been studied, the resulting specialized tools are compared in the interest of developing a single "super tool" which will enjoy the functionality of each of these individual tools, but without loss of naturalness. To date, only limited success has been achieved in this process of "amalgamating super tools". Certainly, more progress can be made on this front, although it is to be expected that some fineness of specialization will always remain in software "tool kits". The skilled designer of rule-based systems will need familiarity with the specific uses of tools in the same way, for instance, as a carpenter must know the appropriate uses of a ball-peen hammer.

In what follows, two model problems will be discussed for which special tools have been designed. Another model problem, now being studied, will be discussed in the light of the extensions to the current tool set which it suggests.

CHAPTER 2

PROBLEMS FOR RULE-BASED PROGRAMMING

A rule is a commonly used mechanism for representing knowledge. Rules take the form of IF-THEN statements. Problem solving systems within which knowledge is expressed in the form of rules are called rule-based systems. The software tools discussed herein are intended to help in the design of rule-based systems.

Two important types of rules include condition-action rules and antecedent-consequent rules (Winston, 1984). Condition-action rules specify that when certain conditions hold, certain actions are to be performed. They are used for problems of synthesis. Antecedent-consequent rules specify that when certain conditions hold, certain consequents are to be considered true as well. These are used for problems of inference. Figure 1 on the following page shows some examples of both kinds of rules.

Figure 1

Condition-Action Rules†

=====

IF: The most current active context is assigning a power supply and an sbi module of any type has been put in a cabinet and the position it occupies in the cabinet (its nexus) is known
and there is space available in the cabinet for a power supply for that nexus
and there is an available power supply

THEN: put the power supply in the cabinet in the available space.

IF: The airfield does have exposed aircraft
and the number of aircraft in the open at the airfield is greater than 0.25 X the total number of aircraft at that airfield,

THEN: Let EXCELLENT be the rating for aircraft at that airfield.

Antecedent-Consequent Rules

=====

IF: 1) The stain of the organism is grampos, and
2) The morphology of the organism is coccus, and
3) The growth conformation of the organism is chains

THEN: There is suggestive evidence (0.7) that the identity of the organism is streptococcus.

IF: The heat transfer from the primary coolant system to the secondary coolant system is inadequate
and the feedwater flow is low

THEN: The accident is loss of feedwater.

† The two examples of condition-action rules are taken, respectively, from the XCON system (McDermott, 1982) and the TATR system (Callero, 1984). The examples of antecedent-consequent rules come, respectively, from the MYCIN system (Shortliffe, 1976) and the REACTOR system (Nelson, 1982).

Given a set of rules, various control paradigms can be used to solve problems. Forward chaining and backward chaining are two frequently used control paradigms. With backward chaining, we begin with a hypothesis which is to be tested, and we try to match this hypothesis either with a fact or with a THEN clause of some antecedent-consequent rule. Backward chaining is used with problems of inference. Forward chaining involves matching facts with the IF clauses of rules, and it is extensively used for problems of synthesis.

The first model problem which we will discuss is an inference problem which lends itself quite naturally to a backward chaining approach. The second model problem is a problem of synthesis for which the governing principles may easily be stated as condition-action rules. These two problems were used as springboards for the development of the software tools.

The Continental Divide Problem

A large land mass can conceptually be divided into two parts by a one-dimensional geographic object called the continental divide, which runs the length of the land mass. All river systems on one side of the continental divide empty into the sea on the same side of the land mass, while all river systems on the opposite side of the continental divide empty into the sea on the opposite side. (We ignore the existence of basins from which no river flows to the sea.) There is no river which flows across the continental divide.

The continental divide is not itself visible. Its location can only be inferred from the behavior of the rivers around it. In order to determine whether a given area lies on a certain side of the continental divide, we look at the rivers which flow through or alongside that area and trace the course of these rivers to the sea. If there are rivers which can be traced to seas on opposite sides of the continent, then we can conclude that the continental divide passes through the area in question.

This simplistic model of ours expresses some general "knowledge" about geography. To the extent that our model is adequate, we can apply its precepts to any specific geographic context we want, and the precepts will still hold. These precepts can be formulated as rules (i.e., IF-THEN statements), while specific contextual information can be referred to as "facts". Rules are general, whereas facts are specific.

We might informally express the rules constituting our general "knowledge" about this model problem as follows:

1. We can conclude that a city lies on a given side of the continental divide if there is a river which flows by that city and if that river flows toward a saltwater body on the given side of the continent.
2. We can conclude that a river flows toward a saltwater body if (i) it flows into that saltwater body, or (ii) it flows into another river which flows toward that saltwater body, or (iii) it flows into a lake out of which flows another river which, in turn, flows toward that saltwater body.
3. We can conclude that a state lies on a given side of the divide if there is a river which flows through that state and which flows toward a saltwater body on the given side of the continent.

4. We can conclude that the continental divide does not pass through a given geographic entity which is a city† if that city lies downstream from some other city.
5. We can conclude that the continental divide does pass through a given area if, by virtue of rules 1 or 3, that area can be shown to lie on one side of the divide as well as on the opposite side of the divide.

These general rules can be coupled with any relevant set of facts to form a knowledge base. Two (among many) possible fact sets are shown below.

Western U.S.A. Facts

The Mississippi River flows into the Gulf of Mexico.
 The Missouri River flows into the Mississippi.
 The Yellowstone River flows into the Missouri.
 The Clark Fork River flows into Pend Oreille Lake.
 The Pend Oreille River flows into the Columbia.
 The Columbia River flows into the Pacific Ocean.
 The Colorado River flows into the Gulf of California.

The Pend Oreille River flows out of Pend Oreille Lake.
 The Yellowstone River flows out of Yellowstone Lake.

The Missouri River flows by Great Falls.
 The Yellowstone River flows by Billings.
 The Clark Fork River flows by Missoula.
 The Milk River flows by Havre.

The Yellowstone River flows through Montana.
 The Missouri River flows through South Dakota.
 The Clark Fork River flows through Montana.
 The Columbia River flows through Washington.

The Gulf of Mexico lies on the east coast.
 The Pacific Ocean lies on the west coast.
 The Gulf of California lies on the west coast.

† The relation of "lying downstream from" will be restricted to cities. We normally think of this as a relation which is irreflexive and asymmetric. However, the Green River, for instance, flows from Utah into Colorado and then from Colorado into Utah. Would we wish to say that the two states lie downstream from each other or that Utah lies downstream from itself?

Western U.S.A. Facts (Continued)

Great Falls lies downstream from East Helena.
 Billings lies downstream from Livingston.

Central European Facts

The Elbe River flows into the North Sea.
 The Vltava River flows into the Elbe.
 The Vistula River flows into the Baltic Sea.
 The Danube River flows into the Black Sea.

The Danube flows by Bratislava.
 The Vltava flows by Prague.
 The Vistula flows by Torun.

The Vltava flows through Czechoslovakia.
 The Danube flows through Rumania.
 The Elbe flows through Germany.

The Black Sea lies on the south coast.
 The Baltic Sea lies on the north coast.
 The North Sea lies on the north coast.

Torun lies downstream from Warsaw.

To treat a rule set and a fact set as separate components of a knowledge base is advantageous for us as we seek to anticipate the requirements of a larger system. In an expert system application, for instance, the fact set may correspond to the detailed information which any competent technician may be expected to know or to be able to find out easily, while the rule set, which exists independently of the particular fact set with which it at some time happens to be associated, models the more abstract knowledge of the domain expert, the fruit of years of experience. In a diagnostic system, for example, rules which match sets of symptoms with their causes, probable and improbable, ought to reflect the wisdom of medical experts, while a paramedic can be trusted to provide facts concerning the observable symptoms of a given patient.

Given our set of rules regarding continental divides and a corresponding set of facts, we can infer new facts. Queries submitted to such a knowledge base represent an example of the kind of inference problem for which backward chaining is particularly appropriate. To illustrate how backward chaining can be applied here, let's consider the problem of determining whether Missoula is west of the divide.

Our hypothesis to be tested (i.e., that Missoula lies west of the divide) becomes the unique element of a stack of "goals" to be "satisfied". A goal can be satisfied if a fact or rule can be found which "matches" it.

With "does Missoula lie west of the divide" at the top of the goal stack, we find no matching fact in our (Western U.S.A.) fact set. We do, however, find a relevant rule - rule 1. In this case, we pop the top off the goal stack and push the IF-clauses of the relevant rule onto the goal stack. Our goal stack will then consist of (i) "does a river flow by Missoula", (ii) "toward which saltwater body does this river flow", and (iii) "does that saltwater body lie on the west coast".

The goal "does a river flow by Missoula" is matched by the fact that "the Clark Fork River flows by Missoula". When a goal is matched by a fact, we can pop the goal stack without having to push. Our remaining goals will be (i) "toward which saltwater body does the Clark Fork River flow" and (ii) "does that saltwater body lie on the west coast".

When the top of the goal stack is "toward which saltwater body does the Clark Fork River flow", rule 2 is identified as relevant.

Rule 2 is associated with a disjunction of IF-clauses. If-clause sets 2(i) and 2(ii) represent dead-ends which must be abandoned. The third choice eventually enjoys success. When its propositions are first pushed onto the goal stack, the contents of the stack become (i) "is there a lake into which the Clark Fork River flows", (ii) "is there a river which flows out of that lake", (iii) "toward which saltwater body does that river flow", and (iv) "does that saltwater body lie on the west coast".

We find the relevant facts that the Clark Fork River flows into Pend Oreille Lake and that the Pend Oreille River flows out of Pend Oreille Lake. By an additional application of rule 2, we verify that the Pend Oreille River flows toward the Pacific (and so, therefore, does the Clark Fork). The last goal on the goal stack is satisfied by the fact that the Pacific Ocean lies on the west coast. When we pop this last goal, the stack is made empty, and this indicates that our inference has been drawn successfully. The original question of whether Missoula lies west of the divide can be answered affirmatively, and this conclusion has been reached by backward chaining.

Certain limitations of the knowledge base we have been describing ought to be pointed out. A first limitation to consider is that of incomplete fact sets. Suppose we need to determine whether Havre lies east of the divide. The fact that the Milk River flows by Havre is given to us, but we have no access to other information regarding the Milk River. In applications dealing with complex, real-world

problems, it is to be expected that rule-based systems will have to cope in the face of inadequate information.

Sometimes, however, the absence of information is the best indicator we have to rely on. We can, for instance, infer that the divide passes through Montana, since both the Yellowstone and the Clark Fork flow through Montana and they lead (indirectly) to opposite coasts. Suppose, though, that we seek to determine whether the divide passes through South Dakota. We know already that the Missouri flows through South Dakota, and the Missouri flows toward the Gulf of Mexico. But we know of no river which flows through South Dakota and toward the west coast. Are we perhaps dealing again with inadequate information? We could attempt to obtain more "directly observable" facts concerning rivers (and creeks) that flow through South Dakota from our "competent technician". We could obtain the names of any number of obscure streams which flow through South Dakota and toward the east coast, but we would be no closer to resolving the issue at hand than we were to start out with. At some point we will have to decide that the reason why we haven't heard of any rivers which lead to the west coast is because there are no such rivers, and that therefore the divide does not pass through South Dakota. This decision will not be the product of direct observations alone, but also requires the intuition that there is nothing to be gained by seeking additional information.†

† Perhaps we could change our model in a way that will allow us to construct a proof that the divide does not pass through South Dakota. In practice, nevertheless, conclusions are frequently (and sometimes must be) reached in the absence of proof.

A last observation to be made regarding this model problem is that the maximum depth of the search tree generated as we attempt to verify a hypothesis is not wholly dependent on the number of rules in our rule set. When the rule set is used with an arbitrarily intricate fact set (involving tributaries of tributaries of tributaries, etc.), the search tree can grow arbitrarily deep, even though only a handful of rules are being applied. Problems with deep search trees and few rules will be informally characterized as "deep and narrow". Games (like Rubik's Cube) are frequently of this "deep and narrow" type and can be solved with backward chaining.

A Machine Translation Problem

The second model problem that we will consider is a problem of synthesis. It is a part of a larger problem: translating Polish sentences into English sentences. Our task will be to take a set of features associated with the verb in a source (Polish) sentence and to assemble the appropriate set of features for the English verb in the target sentence.

Since this is a synthesis problem, it is not surprising that the knowledge which we bring to bear to solve the problem can quite naturally be stated as condition-action rules. The conditions of these rules will involve values of features of the original Polish verb as well as information about the context in which the verb appears. The actions to be taken will involve assigning values to features of the corresponding English verb.

The use of condition-action rules for representing knowledge suggests the possible choice of forward chaining as a control paradigm. With forward chaining, we will be matching facts with the conditions of rules. In this example, a set of facts will comprise values of features of the Polish verb currently in question as well as relevant contextual information.

As we saw in the previous example, backward chaining is explicitly goal-directed. We maintained a stack of goals to be satisfied. We matched the top of the goal stack with a relevant fact or rule. Success in problem solving was signalled by an empty goal stack.

When forward chaining is used, however, goals must be pursued implicitly. The mechanism merely matches facts with the conditions of rules, and in so doing, it may well solve a host of irrelevant problems in addition to the original one. In this model problem, our goal is to specify a set of features for a translated verb. But there will be no equivalent to an empty goal stack sounding off a buzzer, so to speak, to let us know when success has been achieved. It will be up to us to detect when the feature set has been fully specified. Moreover, the number of features for which values must be assigned is not fixed. This is due to dependencies between features. If an English verb is in the indicative mood, for instance, it can assume one of a number of tenses, but if the verb is in the imperative mood, then it will not have a tense. (It might be argued that all imperative verbs are implicitly future tense, but then too it would be unnecessary to specify the tense, since the feature would be redundant.) Similarly, if the verb is translated into English as an

infinitive, then it will have no mood, since mood is a feature of finite verbs only. Thus, for any particular instance of this machine translation problem (where an instance is a verb inside a Polish sentence), a solution must first be pursued before the necessary components of the solution can be defined.

We will return to the issue of deciding how the synthesis of a solution should terminate and how it should begin. First, it is necessary to introduce the cast of players for this model problem: features of verbs and the values which those features may assume. Figure 2 below presents the relevant features and their possible values for both languages. Our concern, it should be pointed out, is with the translation of predicates of Polish clauses. (One-to-one correspondence should not be expected between the clauses of the source sentence and the clauses of the translated sentence.)

Figure 2

| Verb Features and Their Values | | |
|--------------------------------|--|---|
| <u>Verb Features</u> | <u>Polish Values</u> | <u>English Values</u> |
| Form | Finite Infinitival Participial Other [†] | Finite Infinitive Bare Infinitive Participle |
| Mood | Indicative Imperative | Indicative Imperative |

[†] Some Polish sentences are verbless. The predicate, if any, will be said to have "other" form.
 Strasznie duszno tu w pokoju. --> (It's terribly stuffy in this room.)
 Szkoda pieniędzy. --> (It's a waste of money.)

Figure 2 (Continued)

| <u>Verb Features</u> | <u>Polish Values</u> | <u>English Values</u> |
|----------------------|--|--|
| | Subjunctive† | Subjunctive I Subjunctive II Conditional†† |
| Voice | Active Passive Impersonal-Reflexive††† | Active Passive |

† Tryb przypuszczający. Schenker (1973) refers to this as "conditional" mood. We will refer to "conditional" as a kind of subordinate clause. Within the software environment to be described (See Chapter 5), "conditional" can't be both a mood and a clause type inside of rule conditions.

†† The names "Subjunctive I", "Subjunctive II", and "Conditional" have been arbitrarily adopted in this paper.

"Subjunctive I" is a mood used extensively in American English inside the complements of a family of verbs including demand, insist, require, prefer, etc. "I insisted that he arrive punctually." Verbs in this mood do not differ in form from imperatives or bare infinitives, although the three can be distinguished by the behavior of their respective subjects. Imperatives have no (surface) subject. Subjunctive I verbs take a nominative case subject. Bare infinitives take subjects in the accusative case. E.g., "I saw him arrive."

"Subjunctive II" appears in the if-clause of contrary-to-fact constructions, in the complements of "wish", and in a few other environments. E.g., "I wish I were you."

"Conditional" mood as defined here appears in the main clause of contrary-to-fact constructions. E.g., "I would quit if I were you." Conditional verbs do not differ in form from indicative verbs of future-in-the-past tense. Both are marked by the auxiliary "would". The distinction between them is semantic. By the way, the problem of conflicting uses of terms (alluded to in the preceding footnote) does not arise here, since the names of English feature values only appear (currently) in the actions of rules.

††† From the viewpoint of Polish philology, "impersonal-reflexive" can't really be considered a distinct voice. However, the construction in question presents the translator with sufficient challenges to merit special consideration.

Daje się dzieciom prezenty szóstego grudnia. --> (Children are given presents on the sixth of December.)

Cieężko się oddycha, jak się ma rozedmę. --> (It is difficult to breathe when one has emphysema.)

Tak mi się chciało pić. --> (I was so thirsty.)

Figure 2 (continued)

| <u>Verb Features</u> | <u>Polish Values</u> | <u>English Values</u> |
|----------------------|------------------------------|---|
| Tense | Present Past Future | Present Past Future Future-in-the-Past Present Perfect† Past Perfect Future Perfect |
| Aspect | Perfective Imperfective†† | Simple Continuous |

A nice property of this problem is the way it naturally resolves itself into modules. The subproblems of determining values of individual features can, to a large extent, be tackled separately. The dependencies referred to above merely help to define the sequence in which subproblems should be addressed. That is, form should be considered before mood, and mood should be considered before tense

† It may be argued that "perfect" should be considered an English aspect, rather than considering present perfect, past perfect, and future perfect to be distinct tenses. One argument against this would be that the sequence-of-tense rule would have to be considered a hybrid sequence-of-tense-or-aspect rule. But the main reason here is a formal one: a given verb cannot have more than one value for a given feature in our model. For instance, in the sentence: "I have been living in Bozeman since 1983", we prefer to say that the aspect is continuous and the tense is present perfect, rather than saying that the tense is present and the aspect is both perfect and continuous.

†† There are important subtypes within the perfective and imperfective aspects, but these are not treated in our preliminary model. Verbs of motion, for instance, may be either "determinate" (e.g., "jechać") or "indeterminate" (e.g., "jeździć") (Stillman, 1972). "Sporadic" is also an important subtype of imperfective ("miewać" as opposed to "mieć"). There are other important subtypes of aspects. All of them can be recorded directly in the dictionary since they are not marked by general, active morphological processes.

(and voice).† It is nevertheless possible to partition our set of condition-action rules on the basis of the subproblems with which they are individually concerned. Ease of modularization may not be a universal feature of problems which are naturally expressed in terms of condition-action rules, but it may be a necessary feature if we are to be assured of the tractability of any problem addressed by a system with a large rule base.

When a subproblem is to be addressed, we can restrict our attention to the rules which are of concern to that subproblem. These rules are said to be activated, while all other rules are inactive. As facts are matched to conditions of rules, only activated rules participate in the matching process. The set of rules which are activated in conjunction with a given subproblem are called a "packet", a term which has been borrowed from Marcus (1980). In this model problem, it is never necessary for more than one packet of rules to be activated at any given time.

There is an initial packet which is activated as forward chaining begins. Other packets may in turn be activated by the action of a rule. A feature may be associated with a packet, and the consequence of this association is that the feature is entered into the solution under construction as its packet becomes activated. When a feature is initially entered into the solution, its value is undefined. A defined value may subsequently be assigned to the feature by the

† A passive imperative may occasionally appear, for instance, in a New Testament translation: "Be healed". But this is a highly marked usage. For normal English, imperative mood implies active voice. For that matter, continuous imperatives are also extremely rare.

action of a rule. The solution is complete and success is achieved when there are no features in the solution with undefined values and there are no outstanding actions to be performed.

Natural language problems tend to require large rule sets. Even for this simple model, the rule set is too large to be presented here in entirety. It is presented in encoded form in Appendix D. We will now consider only two of the packets.

When the initial packet is activated - and this is always the very first step to be performed as we set out to solve instances of our model problem - form is identified as a feature for which a value must be defined. The solution is concerned, of course, with values for features of the verb in the English translation. The initial packet contains the following three rules:

IF the form of the source verb is finite
THEN assign finite form to the translated verb and
activate the mood packet.

IF the form of the source verb is infinitival
THEN activate the infinitive-only packet.

IF the form of the source verb is participial
THEN activate the participle-only packet.

Given forward-chaining, we would normally scan all of the rules and attempt to match their conditions with the current state of the fact set. Those rules for which matches are found are said to be "triggered", but they are not "fired" until all of the rules have been scanned. Then a conflict resolution scheme is used to determine which one of the rules that are triggered may now be "fired". When a rule is fired, its actions are performed.

The three rules in the initial packet all have mutually exclusive conditions, so it is impossible for more than one of them to be triggered for any one problem instance. This is not true of all of the rules in all of the packets for this model problem, but there is a large number of mutually exclusive rules. This means that there is no great need for a sophisticated conflict resolution mechanism. Instead, a rule will be fired as soon as it is found to match the facts. That is, the actual order of the rules is what resolves conflicts. Fact (and rule conditions) all refer to the source (Polish) sentence. Since these facts will not be changed, one pass through the rules is all that will be needed for any activated packet.

The tense packet, which identifies tense as a feature for which a value must be defined, contains an interesting pitfall for this strategy of relying on order alone for the resolution of conflicts. To illustrate the problem, it is sufficient to consider two rules:

IF the clause to be translated is embedded in an indirect
speech construction &
the tense of the superordinate clause is past
THEN transform the tense of the translated verb by the
sequence-of-tense rule.

IF the tense of the source verb is past
THEN assign past tense to the translated verb.

Suppose the first rule above were to precede the second rule inside the packet. These rules are not mutually exclusive, so it may be that both should be applied. If we try to apply the first rule before applying the second, then we will have to change the value of the tense even though no value has yet been defined. However, given the reverse scenario, we may assign past tense to the translated verb and then find that all features have defined values and that no actions

are outstanding. We will have to conclude that the solution is complete. Thus, the sequence-of-tense rule will not be applied, although perhaps it should be.

Our resolution to this ordering problem will be to maintain the order given above. However, the action of the first rule will take on the following meaning:

THEN record within the solution that the tense of translated verb will have to be transformed by the sequence-of-tense rule.

Then, when a solution has been obtained, a separate procedure is invoked to effect the transformation.

This model problem contains more rules than did the continental divide problem. The number of rules would grow spectacularly if we were to develop a system that could be used for practical translation. However, because only one pass is made through the rules of any given packet, the depth of the execution tree is linearly dependent on the size of the rule set. Problems of this sort will be informally characterized as "wide and shallow". Whether or not this linear dependence on the size of the rule set could be maintained if we were to develop a practical machine translation system lies beyond the scope of this paper, but the "shallowness" property and the ease of modularization make this model problem an attractive one.

Other Problem Types

The two model problems we have discussed so far in this chapter have certain complementary properties. Nevertheless, there are many types of problems which we might want to solve with rule-based systems and which have important features exhibited by neither of these two

model problems. We may wish, for instance, to associate probabilities with the consequents of rules. (See Figure 1 above.) We might want to associate priorities with the actions of rules. Probabilities or priorities may need to be dynamically adjusted in the light of emerging conditions. Moreover, some readily modularized problems may be best solved by a hybrid approach: while one part of the problem suggests backward chaining, other parts may prescribe a forward chaining control paradigm. For still other problems, neither of these control paradigms, as they have so far been described, will be adequate.

Let's consider the needs of a specific problem, which we will call the Robot Route Planning problem. The problem can be stated as follows. There is a large floor which is divided into separate rooms. Some of the rooms are joined by doors, but all the doors are initially locked. For every door there is a key which can open the door from either side and which is itself located somewhere on the floor. The robot begins in a starting room and must make its way to a destination room. If it is possible for the robot to reach its destination, find a route which entails going through a minimal number of doors.

At first glance, this problem looks like a good candidate for backward chaining. The initial goal (or hypothesis) to be validated is that the robot can go from the starting room to the destination room. The fact set consists of the configuration of doors, keys and rooms, including the destination room, as well as information that varies with time, such as the current location of the robot and the

list of keys currently in its possession. A few simple inference rules could be formulated, such as:

A robot can get from room 1 to room 2

IF there is a door leading from room 1 to room 2 &
the robot has the key to this door

A robot can get from room 1 to room 2

IF there is a door leading from some room 3 to room 2 &
the robot has the key to this door &
the robot can get from room 1 to room 3

A robot can get from room1 to room2

IF there is a door leading from some room 3 to room 2 and
the key to this door is in some room 4 and
the robot can get from room 4 to room 3 and
the robot can get from room 1 to room 4.

etc.

However, backward chaining, as we have described it, is inappropriate as a control paradigm for the Robot Route Planning problem, because it searches for a solution in a depth-first manner. That is, when a goal matches the consequent of some rule, the goal is popped off of the goal stack and the IF-clauses of the matching rule are all pushed onto the goal stack. We then pursue a solution with the new goal stack configuration until either success or failure is achieved. The old goal stack configuration will not be restored unless the ultimate failure of the new configuration forces us to backtrack. Working in this depth-first manner, we are certain to find a possible route if there is one, but we will have no assurance that such a route will involve a minimal number of passages through doors. To find a minimal route with a depth-first paradigm would require an exhaustive search, but with a proper breadth-first technique, the first possible route we find (if any) will be minimal. This would involve the use of a goal queue instead of a goal stack.

Although a forward chaining strategy could also be pursued for the Robot Route Planning problem, the problem differs markedly from the model Machine Translation problem discussed above. The set of facts describing the configuration of the floor and the location of the robot will, as we have noted, change during execution. Hence, there will be more than one pass through the rule set. As in the case of the Continental Divide problem, execution time is governed by the intricacy of the facts (associated with a given problem instance), not the rules.

The software tools which have to date been designed for assisting in the development of rule-based systems and which will be discussed in the following chapters have been built in the light of these two model problems: the Continental Divide problem and the Machine Translation problem. Thus, optimal support is not yet provided for certain problem types, but users will be able to choose the control paradigm which is more appropriate for their application: backward chaining or forward chaining. It is highly doubtful that we could really make life easier for system designers by attempting to shield them from decisions of this sort.

CHAPTER 3

THE LIFE CYCLE OF A RULE-BASED SYSTEM

In this chapter, we will conceptualize the stages in the "life" of a rule-based system, the people who are involved with the system through these various stages, and the needs which a software support system might be expected to address at each stage.

Let's consider a stage in the development of a rule-based system at which the system designer or system designing team (which may include a domain expert as well as a knowledge engineer) has assembled a set of rules (in English) expressing the knowledge relevant to a given problem and has selected an appropriate control paradigm. There is already a "shell" or skeletal system which will control the flow of execution. It is now necessary to encode the rule set in order to build a knowledge base which is compatible with that shell. At this stage, the software support system should assume as much responsibility as possible for ensuring the proper encoding of rules as well as compatibility with the shell. We will call this the Rule Set Encoding stage.

Certainly, the Rule Set Encoding stage does not represent the first important stage in the system development process. A great deal of work is required in order to assemble a set of rules which adequately express knowledge of a given problem domain. In fact, it has been asserted that the main "bottleneck of expert system building

lies in knowledge acquisition from experts" (Michie, 1984). There is a lot of potential for expediting the knowledge acquisition process with software tools, but this specific capability has not yet been incorporated into the software support system to be discussed herein.

A subsequent developmental stage which can be significant for many systems involves encoding a set of facts in a form that is compatible with a given shell and rule set. The rules and facts together will then make up a complete knowledge base. For some applications, such as our Machine Translation problem or perhaps diagnostic problems, the (possibly transitory) facts refer only to a specific problem instance and are likely to be entered into the system, not during development, but during an actual interactive session with an end user. For other applications, it may be desirable to use a large, already available data base as a fact set. Under such circumstances, it would be preferable to have the shell and the rule set conform to the data base. Our model of system development, however, will assume that fact sets conform to the rule set and shell. The Continental Divide problem is representative of a class of applications in which a large set of facts can be relevant to the solving of many problems. For such cases, Fact Set Encoding is a developmental stage. Our software support system has a contribution to make at this stage as well.

Of course, we also need to consider the End User's Interactive Session with the rule-based system. The shell sees to it that a solution is pursued expeditiously and in accordance with the stipulations of the rule set. (The correctness of the rule set remains the responsibility of the system designing team.) An

informative explanation of how a solution is derived (or fails to be derived) needs to be presented to the user. There also needs to be a mechanism which will allow the user to present queries to the system. This mechanism has to be both convenient and robust. Finally, in the event of a need to obtain additional facts, the system should have the capability to present the user with intelligible questions, explain why the question is relevant to the derivation, and process the user's response.

Although the software tools designed here at Montana State will primarily be used in the development of small rule-based systems for instructional purposes, we will adhere to a model of the development and use of a commercial expert system. The domain expert and knowledge engineer will select the appropriate control paradigm and will be empowered to write rules. Fact set encoding can be accomplished by technicians who are familiar with the given rule-based system. The end users may also be assumed to be technicians who are familiar with the rule-based system. As such, they may add new facts to the knowledge base, but they are empowered neither to add new rules nor to alter existing ones.

The stages we have discussed should not, of course, be considered chronologically rigid. The system designing team will itself engage in interactive sessions with early versions of a rule-based system. The behavior they observe will point the way toward necessary revisions and expansions of the rule set. These will be followed by a new round of rule encoding (and perhaps fact encoding) and still more interactive sessions, until confidence in system performance is

affirmed by observation. Nevertheless, defining these separate stages provides a perspective within which to measure the utility of our software tools.

A backward chaining shell has been developed in light of the Continental Divide model problem, and a forward chaining shell has been developed in light of the Machine Translation model problem. The shells are accompanied by other supporting software. The Rule Set Encoding stage, the Fact Set Encoding stage, and the End User's Interactive Session (together with attendant assumptions about software users) furnish a framework in which to present and evaluate this software.

CHAPTER 4**THE BACKWARD CHAINING SHELL**

Various software tools have been developed to contribute to the assembly of a rule-based system with a backward chaining control paradigm. The work of drawing inferences is conducted by a basic shell, which also uses a few generic strategies for producing clarifications. A separate module uses menus to obtain a query (which will become the initial goal on the goal stack) from the user. This user-interface module works in consort with the basic shell and a particular knowledge base to provide an environment for an interactive session with an end user. Other software tools help to build a knowledge base, which consists of a file of rules and a file of facts. The software ensures that these rules and facts are compatible with the shell.

All of the software is written in Common Lisp. This means, in particular, that the ensuring of compatibility requires that rules, which may be informally expressed in English, should be converted into objects of data that can be manipulated by a Lisp program, namely, the shell. Similarly, the queries and new facts obtained from the user during a session will have to be brought into the system as objects of the appropriate data type. At the same time, users at all stages (i.e., end users as well as the rule-based system designing team) must

be sheltered (as well as possible) from exigencies imposed by the language of implementation.

Files of Rules

A rules file and a set of facts embody the domain-specific knowledge of our rule-based system. The rules file is the end product of rule set encoding and is built through the joint work of the system designing team and special software tools. This work comes in two phases: a compilation phase and an augmentation phase. The latter is interactive. The rules file which is eventually produced contains Lisp objects that are used during interactive sessions with the end user as well as for the building of fact files.

Actually, a rules file contains a lot of other things besides rules. Major contents of a completed rules file include:

- (1) rules
- (2) property and value bindings for variables
- (3) property bindings for sets
- (4) *FORMS_LIST† - a list of templates for recognizable propositions. Each fact, goal, consequent and antecedent in the system must conform to some template in *FORMS_LIST.
- (5) *FACT_FORMS. All knowledge in the system is expressed in the form of propositions built in accordance with the templates in *FORMS_LIST. For practical purposes, the templates are divided into those which express derived knowledge and those which express knowledge based on observation. The line separating these kinds of knowledge is drawn

† "Special" variables in this software can be recognized by the asterisk at the front (but not the back) of their names. This is an adaptation of the Common Lisp convention of marking special variables with asterisks (e.g., *standard-output*).

arbitrarily. "The patient's blood pressure is 210 over 150" may be considered observable knowledge, whereas "the patient is in danger of an imminent stroke or heart attack" may be considered derived knowledge. At run time, the rule-based system is only permitted to ask the end user to provide observable knowledge. *FACT_FORMS consists of the proposition templates from *FORMS_LIST which correspond to observable knowledge.

- (6) *FACT_SKELETON - an empty, indexed structure into which facts can subsequently be inserted. The indexing is intended to allow the search for a relevant fact more closely to approximate logarithmic time than the linear time search which we would get with a simple list.
- (7) *CONSTRAINT_SET, which expresses functional dependencies that hold among the entities in a fact form (i.e., in an element of *FACT_FORMS). The ability to distinguish one-to-one relations, many-to-one relations, and many-to-many relations will help a rule-based system to prune off bad branches in its search tree more rapidly as well as to recognize contradictions in a fact set as they arise.
- (8) *QUERY_FORMS, from which the correct form for a desired query will be selected by the end user, and
- (9) an instruction to load the desired user-interface module for obtaining queries into the Lisp interpreter.

The precise roles played by these (and other) items in a completed rules file will be scrutinized as we consider later stages in the life of a rule-based system.

Our concern will first be turned toward the building of rules files. This process has evolved somewhat as the tools have been developed. As was noted above, there are now two phases: a compilation phase and an augmentation phase. During compilation, a source file containing rules written in pseudo-English is fed to a translator program. The user (in this case, the rules file builder,

i.e., system designing team) simply waits to receive either a report of successful translation or an error message. If translation is successful, the rules will have been converted into a Lisp data object. The augmentation phase, on the other hand, is intensively interactive. During this phase, many of the contents of a rules file (as listed above) are built. The user answers questions as they are presented by the software. The augmentation phase can be interrupted and later resumed, through the creation of intermediate rules files. In the initial conceptualization, the entire process of building a rules file was to be an interactive one. However, even with this simple Continental Divide model problem, the interactive work has proved to be wearying. The further evolution of this software will therefore probably involved an expansion of the user's opportunities to issue directives in a batch mode.

Bacchus: A Language for BACKward CHaining

A simple formalism has been defined in which a system designing team may express the rules which are to compose a rule base to be used with the Backward Chaining Shell. The formalism will be referred to as Bacchus. Although Bacchus places some constraints on the form in which rules may be expressed, effort has been made to ensure that rules written in Bacchus still sound like rules written in English.

Before discussing Bacchus in depth, we will present a sample transformation from a rule written in English to the same rule expressed in Bacchus:

We can conclude that a city lies on a given side of the continental divide if there is a river which flows by that city and if that river flows toward a saltwater body on the given side of the continent.

```
$ City1 lies on the $ Direction1 side of the divide IF
  $ River1 flows by $ City1 &
  $ River1 flows toward $ Saltwaterbody1 &
  $ Saltwaterbody1 lies on the $ Direction1 coast @
```

A Bacchus rule, such as the one above, consists of three kinds of elements: reserved words, predicative elements and variables. The reserved words appearing in the rule above are 'IF', '&', '\$', and '@'. 'IF' separates the consequent of a rule from its antecedents. '&' conjoins antecedents. '\$' precedes variables. '@' marks the end of a rule.

The same rule can be written in Prolog (after which Bacchus, to some extent, has been modelled) in the usual prefix notation as:

```
lies_on (X, Y, divide) :-
  city (X),
  direction (Y),
  river (Z),
  flows_by (Z, X),
  saltwaterbody (W),
  flows_toward (Z, W),
  lies_on (W, Y, coast).
```

Predicative elements and variables both have the syntactic form of identifiers. An identifier can have the form of any legal Common Lisp symbol which does not correspond to a reserved word of Bacchus.†

The six reserved words of Bacchus include the four mentioned above as

† The possible forms for symbols in Common Lisp constitute an extremely broad set. There are, of course, character strings which cannot be names of symbols, however. Most importantly from our perspective, numbers cannot be Common Lisp symbols. Currently, therefore, numbers cannot appear in Bacchus rules. The task of revising the Backward Chaining Shell so that numbers would be permissible within Bacchus rules would not be a difficult one.

well as 'PRIM' and (currently) 'EOF'. Since predicative elements and variables both have the form of an identifier, the Bacchus translator is only able to distinguish between them on the basis of the context in which they appear.† That is, variables are preceded by the reserved word '\$'; predicative elements are not.

The semantic differences between variables and predicative elements merit immediate comment. The process of satisfying goals consists primarily in pattern matching. A goal matches a fact or a consequent of a rule when the elements of the goal match the corresponding elements of the fact or consequent. Two predicative elements match if and only if they are identical. Determining whether elements match is a more complex issue, though, when variables enter the picture. Variables can be matched with "entities" or with other variables. In the Backward Chaining Shell, variables range over sets. (We may draw an analogy to procedural programming languages, in which variables may take values of specified data types.) The set over which a given variable will range is determined by the Bacchus translator itself on the basis of the variable name. If the variable name ends in a digit, the digit is chopped off to form the name of the set over which the variable will range. 'City1' will range over the set 'City'. (If a variable name in a source file of Bacchus rules

† The attentive reader will have noticed differences in form between the variables and the predicative elements in the Bacchus rule presented above. These differences are stylistic. Variable names begin with an upper case letter. This is intended to be reminiscent of the syntax for variables in Prolog. For the Bacchus translator, however, this upper case letter has no significance. When new symbols are interned in Common Lisp, the names of the symbols are normalized to upper case. So 'City1' and 'city1' would both be interned as 'CITY1'.

does not end in a digit, that name will be associated with the set, and a new name will be built for the variable by appending a digit.) The entities of which a set consists are specified later, e.g., during fact set encoding. A variable matches an entity if the entity is an element of the set over which the given variable ranges. Two variables match if at least one of the sets over which they respectively range is a subset of the other.† When a variable within a pattern is matched, "instantiation" is performed, a process about which we will have more to say later.

An important difference between Prolog variables and Bacchus variables may deserve mention at this juncture. In Prolog (Bratko, 1986), variables may be said to range over the universe of Prolog terms in a given program (or module). Due to the automatic "data typing" in Bacchus, on the other hand, the ranges of variables are restricted. For example, in Bacchus, the variable "Saltwaterbody1" will match neither the entity 'billings' nor the entity 'missouri_river', given the proper encoding of a relevant fact set.

A source file to be processed by the Bacchus translator is well-formed if it is a collection of one or more Bacchus rules. The following BNF productions define the possible Bacchus rules.

† Variables ranging over sets which overlap but which do not conform to a subset-superset relation do not match in the current implementation.

Figure 3

Extended BNF Productions for Bacchus†

```

=====
<rule> ::= <consequent> IF <body> @
<consequent> ::= <proposition>
<body> ::= <premise> { & <premise> }
<premise> ::= <proposition> | <semantic primitive>
<proposition> ::= [ $ ] <identifier> { [ $ ] <identifier> }
<semantic primitive> ::= PRIM <action>
<action> ::= EQUATE $ <identifier> and $ <identifier> |
            DISTINGUISH $ <identifier> from $ <identifier> |
            FAILURE
=====

```

Semantic primitives in Bacchus represent deviations from the regimen of solving problems through pattern matching alone. They enhance the expressiveness of the formalism, for instance, when certain conditions are considered evidence against, rather than for, a

† The notation {x} means zero or more occurrences of x; the notation [x] means zero or one occurrence of x. Symbols enclosed in angled brackets are non-terminals; unenclosed symbols are terminals. A Bacchus identifier can be any Common Lisp symbol that is not already a Bacchus reserved word. Concerning Common Lisp symbols, Steele (1984) writes that "any token that is not a potential number and does not consist entirely of dots will always be taken to be a symbol." Any string of constituents is a token. Constituents include the digits (0 through 9), upper case letters, lower case letters, and the following characters: '\$', '%', '&', '#', '+', '-', '.', '/', ':', '<', '+', '>', '?', '@', '[', ']', '^', '_', '{', '}', and ' '. A token is a potential number - in common Lisp - "if it satisfies the following requirements:

- 1) it consists entirely of digits, signs (+ or -), ratio markers (/), decimal points (.), extension characters (^ or _), and number markers. (A number marker is a letter. Whether a letter may be treated as a number marker depends on context, but no letter that is adjacent to another letter may ever be treated as a number marker. Floating-point exponent markers are instances of number markers.)
- 2) it contains at least one digit (Letters may be considered to be digits, depending on the value of *read-base#, but only in tokens containing no decimal points.)
- 3) it begins with a digit, sign, decimal point, or extension character.
- 4) it does not end with a sign. (Steele, 1984)"

particular goal (or hypothesis). Suppose we wish to incorporate the following rule into our system:

We can refute the claim that the continental divide passes through a given city if that city lies downstream from some other city.†

The rule can be expressed in Bacchus†† as follows:

The divide passes through \$ City1 IF
 \$City1 lies downstream from \$ City2 &
 PRIM failure @

Let's consider the effect of encountering this semantic primitive during backward chaining. Suppose the top of the goal stack has the form: "The divide passes through Billings". This goal matches the head (i.e., consequent) of our new rule. The variable "City1" is instantiated to the entity 'Billings'. Because a match has been found, the top goal is popped, and the antecedent clauses of the rule are pushed onto the goal stack with instantiations. The top goal will now be: "Billings lies downstream from City2." This is matched with the fact: "Billings lies downstream from Livingston", causing the goal stack to be popped. The new top goal will be "PRIM FAILURE". The reserved word 'PRIM' indicates that the regular pattern matching activity is to be suspended and that a specified action is to be

† This rule makes sense if we treat cities as points and assume that rivers always flow away from the divide and never along it. (Treating cities as points is a bit more admissible in Montana than in some other places.)

†† This Bacchus rule would have the following Prolog equivalent:
 passes_through (divide, X) :-

city(X), city(Y),
 lies_downstream_from(X, Y),
 !, fail.

'!' and 'fail' are semantic primitives in Prolog.

performed. In this case, the action is specified by the keyword 'FAILURE', which not only causes the current goal to fail, but also causes that failure to propagate back to the "parent" goal, i.e., the goal which matched the head of the rule of which "PRIM FAILURE" was a clause. Here, the parent goal was "The divide passes through Billings". A goal "fails" when it cannot be satisfied. If the parent goal were the original query, we would return a response of "No" to the end user.† Otherwise, backtracking would begin with the immediately preceding goal stack configuration.

Another use for semantic primitives arises when we wish to make sure that two variables will receive distinct instantiations.

Consider how we might formalize the following rule:

We can conclude that the continental divide passes through a given area if that area lies (partly) on one side of the divide and also lies (partly) on the other side.

† With the FAILURE primitive, unexpected results may be obtained when a goal contains an uninstantiated variable. Suppose, for instance, that the divide passes through the city of Shangri-La. Given the initial goal "the divide passes through Shangri-La", we may obtain an affirmative reply, because the refutation does not apply (i.e., there is no City2 such that Shangri-La lies downstream from City2), and we will be free to construct a proof based on other rules. Suppose, on the other hand, that the initial goal were "the divide passes through City1." Our normal interpretation of such a goal would entail existential quantification: is there a city through which the divide passes? In the current implementation, however, the FAILURE primitive imposes universal quantification onto the parent goal. That is, "the divide passes through City1" successfully matches the head of our refutation. The antecedents are pushed onto the stack, and "City1 lies downstream from City2" matches the fact that "Billings lies downstream from Livingston". When this goal is popped, we are left with the FAILURE primitive, which causes failure to propagate back to the initial goal. The shell will assert that the initial goal failed, even though the divide does pass through Shangri-La, because there exists a city (Billings) for which the refutation is valid. Bratko (1986) refers to the same wrinkle with respect to the "not" primitive in Prolog on page 135.

A first try at expressing this in Bacchus might result in something like this:

```
The divide passes through $ Area1 IF
  $ Area1 lies on the $ Direction1 side of the divide &
  $ Area1 lies on the $ Direction2 side of the divide @
```

'Direction1' and 'Direction2' both range over the set Direction, where, let's say, Direction = {east, west}. The problem with this first formulation is that Direction1 and Direction2 may receive the same instantiation. We could, for instance, prove that the continental divide passes through South Dakota because South Dakota lies east of the divide as well as east of the divide.

One way to resolve this problem is through the use of a semantic primitive. We reformulate our Bacchus rule:

```
The divide passes through $ Area1 IF
  PRIM DISTINGUISH $ Direction1 FROM $ Direction2 &
  $ Area1 lies on the $ Direction1 side of the divide &
  $ Area1 lies on the $ Direction2 side of the divide @
```

When it is on the top of the goal stack, the 'DISTINGUISH' primitive will suspend pattern matching and cause the eventual instantiations of Direction1 and Direction2 to be distinct.

The Bacchus translator produces four Lisp objects as the result of a successful translation: a list of rules, a list of variable names, a corresponding list of set names, and *FORMS_LIST. The translated form of the rule above might be:

```
((THE DIVIDE PASSES THROUGH AREA1)†
 (AND (PRIM (DISTINGUISH DIRECTION1 FROM DIRECTION2))
 (AREA1 LIES ON THE DIRECTION1 SIDE OF THE DIVIDE)
 (AREA1 LIES ON THE DIRECTION2 SIDE OF THE DIVIDE)))
```

† Rule numbers are inserted during translation.

Bacchus, which is a regular language, is translated by recursive descent. Most of the error recovery is by panic mode, but inside of semantic primitives, missing keywords are simply inserted.

The Augmentation Phase of Rules File Building

Much of the information contained in a completed rules file is assembled after the translation process has been completed. This additional information is interactively obtained from the system designing team during the augmentation phase. Four issues which are addressed during this time include: (i) which pairs of sets are in a subset-superset relation, (ii) which sets are closed and which can be assigned new elements during an interactive session with an end user, (iii) what constraints must be obeyed by the facts of a potential fact set, and (iv) what forms will describe the permissible queries to which the system will respond. Additional structures are built independently by the software. These include indexed lists, which will be described below.

A proper representation of subsets and supersets is necessary for the expression and enforcement of constraints. Suppose, for instance, that we wish to impose the restriction that every river must have a unique mouth. Our rules suggest that any given river may flow into a saltwater body, or it may flow into another river, or it may flow into a lake. If, during execution, the top of the goal stack seeks to determine whether there is a river into which the Clark Fork River flows, we would like the Backward Chaining Shell to be able to discontinue its search once it has uncovered the fact that the Clark

Fork River flows into Pend Oreille Lake. That is, given that there is a lake into which the river flows, we know that it cannot flow into a river or sea. One way to achieve the proper interaction among these sets is to define a superset. Let's call it Waterbody. Let $\text{Waterbody} = \text{River} \cup \text{Lake} \cup \text{Saltwaterbody}$. A resultant way to express our constraint is that every river flows into a unique waterbody.

The software support system scans the elements of `*sets` (as assembled by the Bacchus translator) and asks the user to indicate which elements are subsets of which other elements of `*sets`. In addition, the software scans the propositions in `*FORMS_LIST` to determine which ones match "weakly". Propositions match weakly if they match with respect to their predicative elements, even though their respective variables may range over different sets. Hence, "River1 flows into Saltwaterbody1", "River1 flows into River2", and "River1 flows into Lake1" will be found to match weakly. The software support system will ask the user to provide a name for the superset comprising the union of the sets Saltwaterbody, River and Lake. In this way, the new set Waterbody is added to `*sets`, and for each element of `*sets` we obtain a specification of its subsets or supersets.

Each element of `*sets` must be designated as an open set or a closed set. Actually, the compositions of sets are not defined until a fact set is encoded. These designations of open and closed sets take effect later, during sessions with the end user. Consider the Central European fact set given in Chapter2. Suppose we wished to

show that the divide passes through Czechoslovakia.† The Backward Chaining Shell would need to show that Czechoslovakia lies (partly) north of the divide and (partly) south as well. The proof that Czechoslovakia lies north of the divide would be easily constructed: the Vltava flows through Czechoslovakia; it flows into the Elbe; the Elbe flows into the North Sea; the North Sea is on the north coast. In order to show that Czechoslovakia also lies south of the divide, however, it would be necessary to obtain a new fact about a river flowing through Czechoslovakia. If River has been designated as an open set, the user could respond, for instance, that the Hron River flows through Czechoslovakia (and then into the Danube and so on toward the Black Sea). The Hron becomes a new element of the set River, and this new fact plays a crucial role in deriving a validation for the initial goal. This was made possible, in part, by the open set designation.

Sometimes, though, we would want to prevent the introduction of new elements into a set. From this Central European perspective, the continent is divided into north and south. If a user sought to enter the fact that the Adriatic Sea lies on the east coast (which would make sense from an Italian perspective), we would want to reject this

† We can easily show that Prague lies north of the divide and that Bratislava lies to the south, but we can't make use of this information as end users during a session with the Continental Divide rule-based system. The fact that Prague is in Czechoslovakia cannot even be formulated if it does not correspond to one of the proposition templates in *FORMS_LIST. It is also inadequate to assert that the Danube flows alongside Czechoslovakia (which will be an entity in the set State), because the rules set forces us to show that a river passes through (not alongside) a state in order to show that the state lies on a given side of the divide.

new proposition, because it violates our world view. This censorship could be achieved by designating Direction to be a closed set. The Western U.S.A. fact set defines Direction to be {east, west}, whereas the Central European fact set defines Direction to be {north, south}. Each of these taken individually with the Continental Divide rules file forms its own knowledge base. When a particular knowledge base is selected for a given session with an end user, the corresponding definition of the set Direction is fixed for the duration of the session, by virtue of the closed set designation.

Knowledge of the constraints obeyed by relations of interest will help the backward chaining rule-based system to recognize dead-ends in a derivation as well as to recognize inconsistencies in the knowledge base. For each of the propositions in #FACT_FORMS, software asks the rules file builder whether more than one instantiation is possible for one of the variables if all of the other variables of the proposition have already been instantiated. Given the proposition: River1 flows into Waterbody1, for example, the rules file builder may say that many instantiations are possible for River1 if Waterbody1 is fixed, but that only one instantiation is possible for Waterbody1 if we fix River1. That is, the Missouri and the Ohio and the Arkansas Rivers all flow into the Mississippi, but the Mississippi flows only into the Gulf of Mexico. "Flowing into" is thus a many-to-one relation. The rules file builder designates it as such interactively, and the rules file records this designation in #CONSTRAINT_SET for later reference.

The software also obtains query formulae from the rules file builder interactively. (Entering these formulae is a particularly

wearisome task. Automatic generation of query formulae may turn out to be a worthwhile extension to the software support system.) These query formulae will be used to build menus to be presented to end users in order to obtain initial goals for backward chaining.

A significant job that is carried out by software during the augmentation phase without need for human interaction is the selection of indices for the construction of indexed lists. The indices are selected from among the predicative elements of the propositions (currently) in #FACT_FORMS. For a given list (or sublist) of propositions, a predicative element in a given location is sought which appears in (close to) half of the elements of the list. Figure 4 below shows an example of an original list and a corresponding indexed list built by this process.

Figure 4

A list (in Lisp notation) of 23 proposition templates without indexing (from the Machine Translation problem):

```
=====
((Context Does1 indicate an imperative meaning directed to an
  undefined audience)
 (The voice of the source verb is Voice1)
 (There Is1 a direct or indirect object)
 (There Is1 a dative argument in the role of beneficiary)
 (Context Does1 indicate an iterative meaning)
 (The translated verb Does1 support continuous aspect)
 (Context Does1 indicate a durative meaning)
 (The subordinate clause is Subtype1)
 (The tense of the source verb is Tense1)
 (Context Does1 indicate the current applicability of the
  proposition)
 (Context Does1 emphasize that the time reference strictly precedes
  that of a neighboring clause)
 (The time reference Does1 extend from some point in the past up to
  the present)
 (The tense of the superordinate clause is Tense1)
 (The clause Is1 embedded in an indirect speech construction)
 (The time reference Is1 restricted to the past)
```

Figure 4 (continued)

(The time reference of the verb to be translated Does1 strictly preceded that of the superordinate clause).
 (The aspect of the source verb is Aspect1)
 (The verb of the superordinate clause Is1 sensory)
 (The mood of the source verb is Mood1)
 (The translated verb of the superordinate clause Does1 support subjunctive complementation)
 (The translated verb of the superordinate clause Does1 support infinitival complementation)
 (The clause to be translated is Clausetype1)
 (The form of the source verb is Form1).

The same 23 templates automatically organized into an indexed list by rules file building software:

```

=====
((the 4
  (source 5
    (voice 2 ((The voice of the source verb is Voice1)))
    (form 2 ((The form of the source verb is Form1)))
    (tense 2 ((The tense of the source verb is Tense1)))
    (mood 2 ((The mood of the source verb is Mood1)))
    (the 1 ((The aspect of the source verb is Aspect1)))
  (context 1 ((Context Does1 indicate the current applicability of
    the proposition)))
  (verb 2 ((The verb of the superordinate clause Is1 sensory)))
  (the 1 ((The tense of the superordinate clause is Tense1)))
  (context 1
    (an 4
      (iterative 5 ((Context Does1 indicate an iterative meaning)))
      (indicate 3 ((Context Does1 indicate an imperative meaning
        directed to an undefined audience)))
      (emphasize 3 ((Context Does1 emphasize that the time reference
        strictly precedes that of a neighboring clause)))
      (indicate 3 ((Context Does1 indicate a durative meaning)))
    (translated 2
      (of 4
        (subjunctive 10 ((The translated verb of the superordinate
          clause Does1 support subjunctive complementation)))
        (the 1 ((The translated verb of the superordinate clause
          Does1 support infinitival complementation)))
        (the 1 ((The translated verb Does1 support continuous aspect)))
      (time 2
        (extend 5 ((The time reference Does1 extend from some point in the
          past up to the present)))
        (of 4 ((The time reference of the verb to be translated Does1
          strictly precede that of the superordinate clause)))
        (the 1 ((The time reference Is1 restricted to the past)))
      (there 1

```

Figure 4 (continued)

(dative 4 ((There Is1 a dative argument in the role of
beneficiary)))
(a 3 ((There Is1 a direct or indirect object)))
(clause 2
(embedded 4 ((The clause Is1 embedded in an indirect speech
construction)))
(the 1 ((The clause to be translated is Clausetype1)))
(subordinate 2 ((The subordinate clause is Subtype1)))

An index such as "the 4" at the head of a portion of an indexed list indicates that the propositions contained in that portion of the list all have the predicative element "the" in their fourth position. Predicative elements are used to the exclusion of variables for selecting indices. In this way, the propositions "Amazon flows into Atlantic_Ocean" and "Zambezi flows into Indian_Ocean" can be stored and retrieved with the same indices as is the template "River1 flows into Waterbody1", without having to address the complicated issue of instantiation during the search itself. The predicative elements "flows" and "into" are common to all three forms.

If we were searching sequentially for the position within a simple list of 23 elements (as in Figure 4 above) of a proposition known to be contained in the list, we could find its exact position using possibly only one probe or possibly as many as 22. On the average (given that all 23 positions are equally likely), we would need 11.96 probes. (The number of probes needed in an average case will of course grow linearly with the number of elements in the list.) If those same 23 elements were stored in an optimally balanced binary tree, the position of a proposition could be discovered in either 4 or 5 probes. On the average, 4.61 probes would be required. (The number

of probes needed in an average case would grow as a logarithmic function of the number of elements in the list.)

Using the indexed list shown in Figure 4, the position of a proposition can be found by using possibly as few as 3 probes or possibly as many as 8. On the average, 5.39 probes will be required.† How well the indexed list generating mechanism approximates the swift retrievability of an optimally balanced binary tree depends on the distribution of predicative elements among the propositions.

The indexing structure that is built for #FACT_FORMS is also used for #CONSTRAINT_SET and for #QUESTION_TREE (which contains some of the formulae of #QUERY_FORMS to be used during backward chaining when additional facts need to be obtained from the end user) as well as for #FACT_SKELETON, into which facts will subsequently be inserted.

Fact Set Encoding

The facts file builder is a technician who is familiar with the forms (i.e., templates) of facts as specified by #FACT_FORMS in the rules file with which he/she is interfacing. Menus are presented for entering facts. First, a form is selected from a list of possible forms. Then, entities are specified for the individual variables of the selected form. In this way, the compositions of sets are defined at the same time as the facts themselves are stored in an indexed list. (The values of variables are the names of the sets over which

† With the indexed list, a probe will involve checking the identity of an individual element of a proposition. In the case of the sequential search through a simple list of propositions, a probe involves testing two whole (arbitrarily long) propositions for equality.

they range. Thus, when new values are assigned to sets, there is no need to assign new values to variables concomitantly.)

The software detects violations of constraints as they occur and presents the facts file builder with the opportunity to correct these violations. (The facts file builder, however, is not allowed to change the constraints themselves.)

The completed facts file will contain an indexed list of facts as well as the values of all of the elements of *sets.

The End User's Interactive Session

When a user invokes the Backward Chaining Shell, the shell can quickly ascertain whether a knowledge base has also been loaded into the Lisp interpreter (within which the entire session takes place). If not, the shell will ask for the names of the rules file and the facts file and will attempt to load them. When these are in place, the business of drawing inferences may begin. The shell is ready to obtain a query, which will become the first goal on the goal stack.

The user is assumed to have some familiarity with the individual rule base. The current default strategy for obtaining a query involves presenting a sequence of menus from which the user selects an appropriate form. (Appendix B discusses how to use this software in order to enter queries effectively.) Let's assume the user wishes to determine whether Missoula is east of the divide. An initial goal is created on the basis of interaction with the user. This goal may have the form: "(Missoula lies on the east side of the divide)".

Backward chaining commences with a goal stack containing only the initial goal. It is passed to a function called 'explore'.† 'Explore' reports success when the goal stack it receives is empty. Otherwise, 'explore' attempts to satisfy the goal on the top of the stack. If this top goal is a semantic primitive, the task of the semantic primitive will be carried out. When the top goal is not a semantic primitive, pattern matching is attempted.

First, we attempt to find a matching fact. The facts are arranged in an indexed list. The indices can be used to find a section of the fact set which is relevant to the current search, if indeed there is such a section.††

Upon return from the attempt to find a matching fact, the rules are scanned for a consequent (i.e., rule header) which matches the top goal. For this example, a match will be found with the consequent: "(City1 lies on the Direction1 side of the divide)". The match results in two instantiations: City1 --> Missoula, Direction1 --> east. These instantiations are effected in the body of the matching rule. The function 'swapgoal' pops the old top off the goal stack and

† Some of the functions in this shell bear the same names as do predicates which play analogous roles in the expert system shell described in Bratko (1986).

†† The current top goal in our example, "(Missoula lies on the east side of the divide)", has a form associated with derived knowledge. In its present state, the shell does not add derived facts to the fact set (although it does add new "observable" facts obtained from users). Therefore, this first search presents a circumstance in which no relevant section will be found. For some applications, of course, it will be desirable to retain derived facts within the fact set. The ability to retain derived facts constitutes a natural candidate for an extension to the existing software support system.

pushes the antecedent clauses of the matching rule (with instantiations) onto the stack. 'Swapgoal' then invokes 'explore'.

For this invocation to 'explore', the top goal will be: "(River1 flows by Missoula)". It turns out that the relevant section of the fact set is marked by the indices "flows 2" and "by 3". Within this section a match is found resulting in the instantiation: River1 --> Clark Fork River. This instantiation propagates through the goal stack.

The following behavior can be observed in our matching operations. Facts are at least as specific as goals. When goals are matched to facts, instantiation propagates through the goal stack. On the other hand, rule headers are at least as general as goals. When rule headers are matched to goals, instantiation propagates through the (antecedent) clauses of the rule body. Thus, whenever we set out to test whether two propositions match, we can order these propositions with respect to their generality.

In Prolog we can match two terms for which no such ordering can be made with respect to generality. For example, the term "flows_thru (River1, montana)" matches the term "flows_thru (missouri_river, State1)". Consider, however, the set of all terms which match the first term and the set of all terms which match the second. Although these sets have a non-empty intersection ("flows_thru (missouri_river, montana)" lies in the intersection), neither set is a subset of the other. (Only the first contains "flows_thru (milk_river, montana)", whereas only the second contains "flows_thru (missouri_river, north_dakota)".) In this situation, the two terms are not effectively

comparable with respect to generality. Such a situation never arises with the Backward Chaining Shell, given that the knowledge base is built with the tools supplied.

After a matching fact has been found, the top goal is popped from the goal stack, and the remainder of the stack (with instantiations) is passed, once again, to 'explore'.

With "(Clark_Fork_River flows toward Saltwaterbody1)" on the top of the goal stack, we will attempt first to find a matching fact. Given that only "observable" facts are stored in the fact set, no relevant section will be found, and control will pass swiftly to the pursuit of a matching rule. The "flowing toward" rule has a disjunction of antecedents. 'Swapgoal' attempts to push each of these disjoint clause sets, in turn, onto the goal stack. For the top goals "(Clark_Fork_River flows into Saltwaterbody1)" and "(Clark_Fork_River flows into River2)", the relevant section of the fact set is located, but no match is found. Furthermore, *CONSTRAINT_SET indicates that no match is possible for either of these goals, because the waterbody into which a river flows must be unique and we have the fact that the Clark Fork flows into Pend Oreille Lake. 'Swapgoal' places the third set of antecedent clauses onto the goal stack and calls 'explore'.

Shortly thereafter, a search is made through the fact set for a match to the goal: "(River2 flows out of the Pend_Oreille_Lake)". Indeed, a match is found, for the Pend Oreille River flows out of Pend Oreille Lake. However, the pursuit of a solution given this fact proves to be fruitless. (After all, this line of reasoning would show Missoula to lie west of the divide, not east.) We backtrack to the

search for a matching fact. No new matches are found. However, *CONSTRAINT_SET does not preclude the possibility that more than one river may flow out of a given lake. (Could it be that the Missouri River also flows out of Pend Oreille Lake? Our model does not expressly forbid such a scenario.) At this point, the shell puts the question to the user: Are there any other rivers that flow out of Pend Oreille Lake?

This question is the first thing to appear on the screen since the user entered the query. (All the reasoning which led up to this question has been carried on privately by the rule-based system.) Therefore, the user is given the opportunity to respond: "Why do you ask?" A representation of the steps in the argument which led up to this need for new information is then presented. Eventually, the user will have to respond with the name of a new river that flows out of Pend Oreille Lake or else respond that there is no such river.

Once a user responds negatively to such a request for new information, the request cannot be repeated through the course of the session. It is not appropriate, however, to add the negation to the fact set. Rather, the request itself is added to a data structure called *DONT_ASK. Before any request for a new fact is presented to the user, *DONT_ASK is checked to make sure that the given request is not forbidden. Users may wish to load their own definitions of *DONT_ASK into the interpreter at the start of a session to prevent the system from pursuing a line of questioning which they foresee to be irrelevant to their applications.

Let's assume that no other rivers flow out of Pend Oreille Lake. Execution will backtrack to the search for a fact which matches "(River1 flows by Missoula)". The "flows by" relation is defined (in #CONSTRAINT_SET as constructed by the system designing team) to be a many-to-many relation. Therefore, although no new matches are found in the fact set, the shell may ask (if #DONT_ASK does not prevent it from doing so) whether any other rivers flow by Missoula. Once again, the user will have the opportunity to ask why this question is relevant to the query. If the user stipulates that no other rivers flow by Missoula, the whole search will fail in short order. The original call to 'explore' will terminate and control returns to 'tell', from which 'explore' was first called. 'Tell' responds "No" to the user's query, that is, Missoula does not lie east of the divide.

The session continues. Several options are presented to the user. A new query may be entered, or the old query may be pursued further. The Whynot option allows the user to investigate the causes for a query's failure. At present, the Whynot option uses the brute force strategy of presenting the entire derivation of failure. Figure 5 on the next page presents the output produced by our model rule-based system running the Whynot option on the query: "Does Missoula lie on the east side of the divide".

Figure 5

Does Missoula Lie on the East Side of the Divide?

=====

The current partial solution is:

Whynot option output

Missoula lies on the east side of the divide --- no matching fact

Missoula lies on the east side of the divide --- by rule 2

Clark_Fork flows by Missoula --- is a fact

Clark_Fork flows toward Pacific_Ocean --- by rule 1

Clark_Fork flows into Pacific_Ocean --- no matching fact

Clark_Fork flows into Pend_Oreille --- no matching fact

Clark_Fork flows into Pend_Oreille_Lake --- is a fact

Pend_Oreille flows out of Pend_Oreille_Lake --- is a fact

Pend_Oreille flows toward Pacific_Ocean --- by rule 1

Pend_Oreille flows into Pacific_Ocean --- no matching fact

Pend_Oreille flows into Columbia --- is a fact

Columbia flows toward Pacific_Ocean --- is a fact

Columbia flows into Pacific_Ocean --- is a fact

Pacific_Ocean lies on the east coast --- no matching fact

Columbia flows into saltwaterbody --- no matching fact

Columbia flows into river --- no matching fact

Columbia flows into lake --- no matching fact

Pend_Oreille flows into river --- no matching fact

Pend_Oreille flows into lake --- no matching fact

River flows out of Pend_Oreille_Lake --- no matching fact

Clark_Fork flows into lake --- no matching fact

River flows by Missoula --- no matching fact

Missoula lies on the coast of saltwaterbody --- no matching fact

The user also has the opportunity to examine the solutions which could be derived for the same query under hypothetical circumstances.

When the user selects the Whatif option, the current fact set is

stored and a temporary copy of it is made to represent the

hypothetical world. To this temporary copy the user adds a new

proposition, such as "The Yellowstone River flows by Missoula". Now,

when the shell seeks to determine whether Missoula lies east of the

divide, a proof is obtained which makes use of this new "fact". The Yellowstone River, which flows by Missoula, also flows toward the Gulf of Mexico, which is on the east coast, and therefore, Missoula is east of the divide.

When success has been attained, the goal stack is empty.

'Explore' calls 'present', which shows the answer and the steps (without the dead-ends) by which the answer was derived. 'Present' then asks the user whether an additional derivation should be pursued. If the user says yes, control is returned to 'explore' and normal backtracking is undertaken in order to find additional solutions. If the user does not wish to seek an additional derivation, control leaps out of an arbitrarily deep run-time stack of module calls from 'present' to 'tell' via the Common Lisp 'throw-catch' mechanism.

The user can also create hypothetical circumstances by selecting the Whatifnot option. This again involves storing the true fact set and creating a facsimile, from which the user will remove a fact. A solution for the same query is then pursued with the reduced (hypothetical) fact set.

When the user seeks to enter a new query after having run the Whatif or Whatifnot option, the permanent fact set will be restored and the copy will be discarded. The session will continue in this way until the user elects to quit.

CHAPTER 5**THE CONDITION ACTION SHELL**

Software tools have also been developed to support the construction of systems which use condition-action rules and a forward chaining control paradigm. There is a shell which synthesizes solutions and which also provides clarifications. The shell works in consort with a knowledge base to form a rule-based system with which an end user may interact. There are also software tools which help to build the knowledge base, which will consist of a file of rules as well as an optional facts file.

All of the software is written in Common Lisp. A completed rules file (or facts file) has to be composed of Lisp objects which can be used by the shell. The supporting software is responsible for ensuring this compatibility.

Some (but not all) of the software that is used to build backward chaining rule-based systems is used here as well. In addition to this shared software, there is software which addresses the special needs of the Condition Action Shell.

In keeping with our discussion of forward chaining in Chapter 2, it is expected that the rule base will maintain its own agenda. It is not necessary to obtain a query in order to begin synthesizing a solution.

The Condition Action Shell and its related software have been developed in light of the Machine Translation problem described in Chapter 2. This problem has many tidy properties which have allowed the software to be streamlined. Facets of this streamlining include: (1) all sets are closed, (2) all pairs of sets (not including the Universe) have null intersections, that is, there are no subset-superset pairs, (3) there is exactly one possible instantiation for each variable in each fact form (although a well-formed fact set does not necessarily contain the unique fact for every form). It is to be expected that the study of new problems will illuminate appropriate directions for extending the condition-action software.

An interesting feature of rule-based systems which use the Condition Action Shell and are built with the related support software is the division of the rule set into packets. This division protects the time efficiency of the system from the severe degradations which would otherwise follow from growth of the rule set. Responsibility for identifying packets currently lies with the system designing team.

The Rules File

Files of condition-action rules are also built in two phases: a translation phase and an augmentation phase. The translation phase produces the rule packets. Other data structures are produced during the augmentation phase. Major contents of a completed file of condition-action rules include:

- (1) rule packets
- (2) property and value bindings for variables

- (3) *VALID_FORMS: an indexed list of proposition templates. Each fact and each rule condition must conform to a template in *VALID_FORMS.
- (4) *FACT_SKELETON, which fills the same role as it did with the Backward Chaining Shell
- (5) *CONSTRAINT_SET, which also fills the same role as it did with the Backward Chaining Shell
- (6) *QUESTION_TREE, which is used to formulate questions to be presented to the end user during an interactive session when new information is required
- (7) value bindings for sets
- (8) output instructions which, when the rules file is loaded, will remind the user of the need to load function definitions for the transforming of solution features.

Cactus: A Language for Condition-ACTION Rules

The system designing team (or rules file builder) creates a source file for translation. The source file will contain a series of rule packet definitions. Each rule packet will contain a name, (possibly) a designated feature, and a list of condition-action rules. When a packet is activated during a session, as discussed in Chapter 2, the designated feature is added to the solution with a null (logically undefined) value, and the rules of the packet are matched with the fact set. When a rule matches the facts, its actions are performed. When there are no actions waiting to be performed and no features in the solution have null values, forward chaining is interrupted and the solution is presented to the user.

The solution may comprise a list of items, each of which is described by a set of features. For the Machine Translation problem,

however, the solution is a list of only one item: the translated verb.

The source file is written in a formalism called Cactus. The following is a portion of a rule packet definition written in Cactus:

Mood Packet

All mood

IF the mood of the source verb is % imperative THEN
 ASSIGN imperative mood to the translated verb &
 ASSIGN active voice to the translated verb @

otherwise

ACTIVATE VoicePacket @

.
 .
 .

IF the mood of the source verb is % indicative &
 the clause to be translated is % subordinate &
 the verb of the superordinate clause is % sensory &
 the aspect of the source verb is % imperfective THEN
 ASSIGN present-participle form to the translated verb &
 EXTRACT mood field of the translated verb @

.
 .
 .

Reserved words in Cactus include: 'IF', 'THEN', 'otherwise', '%', '\$', '@', '^', '&', and (currently) 'EOF'. Any legal Common Lisp symbol which is not a reserved word of Cactus will be treated in Cactus as an identifier. 'IF' usually marks the beginning of a rule. The end of a rule is always marked by '@'. 'THEN' separates the conditions of a rule from its actions. '&' conjoins individual conditions or actions within a rule. 'otherwise' is a special condition which is satisfied if and only if no previous rule within a given packet has been fired. 'otherwise' is always the last condition in a rule; 'THEN' does not appear alongside 'otherwise'. If 'otherwise' is also the first condition in a rule, 'IF' does not

appear either. '^' marks the end of a packet. '%' precedes identifiers and marks them as "entities" within a set over which a variable will range. '\$' marks the following identifier as a variable, but does not see use for the Machine Translation problem. 'EOF' is a token generated by the Cactus translator when it reaches the end of a source file.

In the example given above, "MoodPacket" is the packet name, "ALL mood" is the feature designation, and the remainder (down to the end-of-packet marker) consists of condition-action rules. A well-formed Cactus source file is composed of one or more rule packet definitions. Possible rule packet definitions are described in the BNF productions given in Figure 5. (Any Common Lisp symbol that is not a Cactus reserved word will be considered an identifier. For a description of Common Lisp symbols, see Figure 3.)

Figure 6

Extended BNF Productions for Cactus

```

=====
<rule packet> ::= <packet name> [ <designated feature> ]
                <rule> { <rule> } ^

<packet name> ::= <identifier>
<designated feature> ::= <item name> <feature name>
<item name> ::= ALL | <identifier> { <identifier> }
<feature name> ::= <identifier>

<rule> ::= <condition list> <action list> @

<condition list> ::= IF <condition> { & <condition> } THEN |
                  [ IF <condition> & ] otherwise

<condition> ::= [ % | $ ] <identifier> { [ % | $ ] <identifier> }

<action list> ::= <action> { <action> }

```

Figure 6 (continued)

```

<action> ::= ASSIGN <value> <feature name> to <item name> |
          ACTIVATE <packet name> |
          EXTRACT <feature name> field of <item name> |
          TRANSFORM <feature name> of <item name> by <func name> |
          TAG-ON <item name> ~ <identifier> { <identifier> }

<value> ::= <identifier>
<func name> ::= <identifier>

```

The keywords 'ALL', 'ASSIGN', 'ACTIVATE', 'EXTRACT', 'TRANSFORM', 'TAG-ON', 'to', 'field', 'of', 'by', and '~' are all legal Cactus identifiers, but they have predefined meanings when they appear in the contexts specified in the syntax diagrams of Figure 5.

When the Cactus translator encounters a new packet (either at the start of a source file or after an end-of-packet marker), the first identifier it encounters is understood to be the name of the new packet. Packet names are stored in a list. If the name of the new packet is the same as that of a previous packet, the older packet is lost.

The feature designation for a packet, if present, consists of an item name and a feature name. The named feature is to be inserted with a null value into the named item of the solution. If the keyword 'ALL' appears in the place of the item name, then all of the items of the solution will receive this feature of (logically) undefined value. (A worthwhile extension to the current definition of Cactus might allow the use, in the place of the item name, of a variable ranging over the items of the solution.)

The parsing of the conditions of rules serves the additional purpose of allowing the translator to build preliminary lists of

CHAPTER 6

DIRECTIONS FOR THE FUTURE

Problem types which we might wish to examine with rule-based systems but which are not addressed by either of the shells now built were discussed in Chapter 2. Types mentioned therein include problems which require (or suggest) a hybrid control paradigm, problems which require breadth-first search for a solution, problems which associate probabilities (or measures of belief) with consequents of rules, and problems which associate priorities with the actions of rules. Each of these (and other) new problem types presents its own requirements for software support. New software tools (or extensive changes to existing tools) would be needed to bring such types into the scope of our software support system for rule-based programming.

Probably more immediate effort should be invested in the application of the existing tools to problems which are similar to (although perhaps more intricate than) the two model problems we have been discussing. This might provide a better measure of the desirability of certain features of the present software than could be attained through the investigation of radically different problems.

Some questions concerning features of the present software tools have already been mentioned. For instance, changes made to the solution during the operation of the Condition Action Shell are not undone by backtracking. In a sense, this gives the actions of our

"entities" and of the forms of propositions (which can later be matched with facts).

The actions of rules guide the building of a solution. Keywords specify the type of action to be undertaken. Values are assigned to individual feature of an item (or all of the items) of a solution by the 'ASSIGN' action. 'ACTIVATE' selects a new rules packet to be used for forward chaining, and if this new packet contains a designated feature, this action ensures that a feature (of null value) is inserted into the appropriate item (or items).

'EXTRACT' removes features from items. An example of the 'EXTRACT' action is seen in the portion listed above of the 'Mood Packet' from the Machine Translation problem. The action is contained in a rule† which could be fired, for instance, by the source verb "wchodzil" in the sentence: "Slyszalem, jak wchodzil do domu." ("I heard him entering the house.") Because "wchodzil" is a finite verb, MoodPacket is activated, which causes the feature "mood" to be designated. However, since sensory verbs in English receive participial complementation (which is also possible in literary, but not colloquial Polish), the form of the translated verb is changed to the present participle, to which no mood can be assigned. Thus, the designation of the mood feature must be revoked, and this is accomplished by the 'EXTRACT' action.

The 'TRANSFORM' action allows us to mark features within the solution as subject to transformation. In accordance with our

† The rules developed for this prototype system, of course, do not attempt to provide rigorous definitions of linguistic environments for the translating operations they perform.

discussion of the sequence of tenses rule in Chapter 2, such transformations will be applied to the values of features after the solution has been fully specified (i.e., after non-null values have been assigned to every feature of every item). The transformations are carried out by external routines (which are loaded into the Lisp interpreter at the beginning of a session with an end user). This transformation process is one possible solution to problems of rule ordering. It remains to be determined (in the light of new applications) whether this particular mechanism injects excessive complications into the task of building a solution or whether, on the other hand, the mechanism is simply not powerful enough.

The last of the currently available actions for synthesizing solutions is the 'TAG-ON'. This appends a comment to an item in the solution. It is to be expected that a need for additions as well as changes to this set of actions will become apparent as new problems are confronted.

The parsing of the actions of rules serves the additional purpose of allowing the translator to build *TRANSFORMS, a list of the names of externally defined functions which will have to be loaded by the end user at run time. Names of rules packets and items are also added to *PACKET_LIST and *ITEMS_LIST, respectively, during the parsing of actions of rules. It is the responsibility of the system designing team, however, to ensure that rules do not attempt to activate packets for which there is no definition.

Cactus, which is a regular language, is translated by recursive descent. Error recovery is primarily by panic mode, although certain keywords are inserted if missing.

The Augmentation Phase

After the source file of condition-action rules written in Cactus have been translated, several jobs remain to be performed in order for a completed rules file to be assembled. These jobs are performed during the augmentation phase of rules file assembly. Some of these jobs require interaction with the system designing team. These include (1) the categorization of entities, (2) verification of the contents of sets, (3) verification of the elements of *ITEMS_LIST, and (4) the construction of QUESTION_TREE. Other jobs are performed entirely by the supporting software. These include the selection of indices for indexed lists and the construction of *CONSTRAINT_SET.

During translation, identifiers appearing after the Cactus reserved word '%' in a rule condition are added to a list called *ENTITIES. For each element of *ENTITIES, it is necessary to determine the set to which that element belongs. This is done during the augmentation phase. Certain entities are recognized by the software to be elements of predefined sets. For instance, "is" and "isn't" (input as: isn't) are recognized to be elements of the set *Is. The majority of entities, however, must be explicitly categorized by the system designing team. For each such entity, the software prints a request for a classification, such as: "Please enter the name of the set to which IMPERATIVE belongs." Suppose that

"Mood" is the user's response to this prompt. If Mood is already an element of *SETS, IMPERATIVE will be added to the value of Mood. Otherwise, Mood will be added to *SETS, its initial value will be the list whose element is IMPERATIVE, and a variable called Mood1 will be created. (Mood will be the value of Mood1.) Proposition templates will be built by replacing IMPERATIVE with the variable Mood1 in every proposition (i.e., rule condition) in which IMPERATIVE appears.

It has been mentioned that "conditional" cannot be both a mood and a clause type within the conditions of a rule set. That is because only one classification request is printed for each element of *ENTITIES. Suppose that our source file contained the following three conditions: "the mood of the source verb is % conditional", "the subordinate clause is % conditional", and "the subordinate clause is % temporal". The translator would add CONDITIONAL and TEMPORAL to *ENTITIES. The user would then be asked to enter the name of the set to which CONDITIONAL belongs. The response might be "Mood". Given this response, the software would be able to build the proposition template: "the mood of the source verb is Mood1". Unfortunately, it would also build the inappropriate template: "the subordinate clause is Mood1". Furthermore, if the user subsequently identified TEMPORAL as an element of the set Subtype, a rival template would be built: "the subordinate clause is Subtype1". Subsequently, the software would ask for the name of the set which is the union of Mood and Subtype. Whereas, in Chapter 4, the introduction of the set Waterbody (which was the union of River, Lake, and Saltwaterbody) was a sensible and positive contribution to our knowledge representation, the union

of Mood (a feature of a verb) and Subtype (a classification for a clause) makes no sense at all. Thus, if conditional mood and a conditional clause are to be considered distinct phenomena and both are to be mentioned in rule conditions in the same Cactus source file, they must receive distinct names.

When the categorization of entities has been completed, *SETS, *VARS (a list of variables appearing in proposition templates), and *FORMS_LIST (the list of proposition templates) are all defined. Because the Condition Action Shell currently assumes that all sets are closed, however, it may be appropriate to extend the definitions of sets obtained during entity categorization. The system designing team is asked to verify the contents of each set. In our model problem, on the basis of entities appearing in rule conditions, Subtype = {conditional, temporal}. Supporting software asks the designing team whether (1) the current value of Subtype is correct, or (2) a new element should be added, or (3) an element should be removed. The need to remove an element would signal a bug in the source file. In this case, the element "other" might be added so that the definition of Subtype would be tenable under a closed world assumption. Thus, in contrast the Backward Chaining Shell, the value of each element of *sets is fixed during the building of the rules file.

The items of the solution which are explicitly named (in rule actions and feature designations) in a Cactus source file are stored in *ITEMS_LIST during translation. This may not be an exhaustive list. During the augmentation phase, the system designing team is asked to provide the names of any additional items to be described by

a solution. These new names, if any, are added to *ITEMS_LIST. It is on the basis of *ITEMS_LIST that the Condition Action Shell will initialize its representation of the solution to be synthesized during an interactive session with an end user.

The fully automated process of building indexed lists is the same for the Condition Action Shell as it was for the Backward Chaining Shell. *FORMS_LIST (a simple list structure) is used to build the indexed list *VALID_FORMS. The indices selected to build *VALID_FORMS are also used to build *FACT_SKELETON, *CONSTRAINT_SET and *QUESTION_TREE.

*CONSTRAINT_SET is currently built without human interaction, since the Condition Action Shell assumes that every variable is constrained. (If this assumption proved to be untenable in the context of a future application, it would be easy to extend the rules file building software to provide the system designing team with the opportunity to select the module - currently associated with the Backward Chaining Shell - which requests the specification of constraints for individual propositions.)

*QUESTION_TREE is built in a process analogous to the construction of *QUERY_FORMS for the Backward Chaining Shell. For every proposition template in *VALID_FORMS, the system designing team is asked to formulate a question which can be presented to the end user during an interactive session whenever additional facts are required. The questions may require a "yes" or "no" answer, or they may require the specification of an entity (to which a variable should be instantiated). An example of the first question type would be: "Is

the clause embedded in an indirect speech construction?" An example of the second type would be: "What is the tense of the source verb?" The end user's response to this second question would be to select a possible tense from a menu. A future version of this supporting software could generate some of these questions automatically (especially those of the yes/no type) from the given proposition templates, but for now the questions are specified by the system designing team.

Fact Set Encoding

A file of facts intended for use with a given file of condition-action rules can be built with the same software used to build fact files for the Backward Chaining Shell.

In the case of the Machine Translation problem, a fact set is simply a description of a single verb and its context within a Polish sentence. In general, since the Condition Action Shell presumes that a rules file has its own "built-in agenda", an individual fact set amounts to a statement of a single problem instance. The normal mode of operation will be to enter facts as they are needed during sessions with the system, rather than to build separate fact files.

The Solution as a Data Structure

It was pointed out in Chapter 2 that forward chaining is frequently used for problems of synthesis. A first step to take in examining the synthesizing work of the Condition Action Shell is to consider the structure of that which we hope to build.

The solution we build will be a list of items. An item will be a Common Lisp structure defined to consist of three elements: **(defstruct item name features comments)**. The "name" of an item will be a simple list. In the Machine Translation problem, our solution will contain a single item, and the name of that item will be the list: (the translated verb). The 'comments' field of an item is a (possibly null) list of lists, each of which is a comment about the translation, such as: (convert the dative to the subject of the translated clause). 'features', in turn, is a list of Common Lisp structures, each called a "field" and defined by the statement: **(defstruct field name value transformation)**. A possible field may consist of the name TENSE, the value PAST and the transformation SEQUENCE-OF-TENSE. The name, the value and the transformation of a field are all atoms.

In the current implementation, values assigned to parts of the solution must be expressly overridden if they are to be changed. That is, backtracking does not restore previous states of the solution. In this respect, our rule-based system successfully models the behavior of obligatory rules. When the conditions for a rule are met, its actions must be taken and are irreversible, unless counteracted by the firing of a subsequent rule. An implementation which allowed the solution to revert to a previous state upon back tracking would successfully model the behavior of optional rules. Thus, either mechanism - or even a mixture of the two - might be desirable, depending on the application.

Interactive Sessions With an End User

When the Condition Action Shell is invoked, it needs to verify that a file of rules has been loaded into the Lisp interpreter. It is also necessary to load the definitions of any transformation functions. (It is generally not necessary for a facts file to have been loaded, since values of sets are defined directly in the rule files.)

The Condition Action Shell does not need to obtain a query in order to commence forward chaining, because the rules file sets its own agenda. The default commencement is the activation of StartPacket. (If, for instance, there is no StartPacket, the system designing team must provide their own version of a 'start_session' function in order to get forward chaining started during interactive sessions with the given rule-based system.)

The initial value of the solution is a list of items with names but no features. The forward chaining process attempts to find matches between facts and the rules of the currently activated packet (beginning, in the default scenario, with StartPacket). When the conditions of a rule are found to match the fact set, the actions of that rule are pushed onto a stack of actions to be performed. These actions may modify the value of the solution and/or change the flow of execution (by activating a new packet).

Actions (currently) do not involve changing the fact set. In this respect, the Condition Action Shell resembles the Backward Chaining Shell, which does not insert derived knowledge into its fact sets. In the case of the Machine Translation problem, it might well be improper

to endow any assertion concerning the translated verb with factual status. On the other hand, the inability to refer directly to features of the translated verb from within rule conditions (which must after all be matched with facts) may result in the awkward formulation of Cactus rules. (An example of such awkwardness can be seen in the Time2Packet of the Machine Translation source file in Appendix D.)

Of course, additions may be made to a fact set during an interactive session by obtaining new facts from the human user. These new facts correspond to "observable" knowledge. (In the case of the Machine Translation problem, they will always concern the source verb.) The Condition Action Shell issues a request for such a fact whenever the section of the fact set relevant to a given rule condition is currently empty. (It is because of this mechanism that no facts file needs to be loaded into the interpreter in order for a session to be conducted.)

The function 'perform' removes actions one by one from their stack and sees to it that they are carried out. This continues until either a new packet is activated or the stack is emptied, signalling that no actions remain to be performed. If the stack is emptied and the solution is fully specified (i.e., if every feature of every item has a non-null value), the solution and its derivation are displayed. The user will then be able to choose whether or not to seek an alternative solution. If the user does want an alternative solution (or if the solution is not fully specified at the time that the action stack empties), backtracking begins. If no alternative solution is

requested, control leaps out of the forward chaining process via the Common Lisp 'throw-catch' mechanism.

When the Condition Action Shell completes its work with the initial problem instance, a menu of options is presented to the end user. If no solution has been obtained for the given fact set, the user can investigate further with the Whynot option (which is exactly as described for the Backward Chaining Shell). The effects of introducing minor changes to a fact set can be investigated with the Whatif and Whatifnot options. New fact sets can be explored by choosing the Load option, which will load a facts file into the Lisp interpreter, or the Erase option, which creates an empty fact set and requires that individual facts be obtained from the user as needed. The session continues in this way until the user selects the Quit option.

rules an obligatory status. We may prefer to have the actions of rules be optional, and this could be modelled by the automatic restoration of previous states of the solution upon backtracking, which would not be difficult to achieve. How would we go about implementing a mixture of obligatory and optional rules, however? It would seem to be necessary to maintain a history of every construct within the solution so that when backtracking took place, the effects of optional actions would be revoked, while the effects of obligatory actions would be maintained. The situation becomes even more clouded if the conditions for obligatory rules are satisfied as a result of the actions of optional rules. An example of a task which might call for such a mixture of optional and obligatory condition-action rules is the determination of the word order of a sentence.

Another issue of concern is the fate of derived knowledge. Currently, the Backward Chaining Shell, for example, allows only propositions which have the form of an "observable fact" to be stored in the fact set. Thus, if "the Missouri River flows toward the Gulf of Mexico" is considered "derived knowledge", it cannot be stored in the fact set and must be derived all over again as required by subsequent queries. The efficiency of our rule-based system would be improved if commonly needed results could be retrieved from storage rather than having to be regenerated, but which results are going to be commonly needed? If all of the intermediate results obtained during the processing of queries are to be stored, the fact set can grow very large, and soon retrieval becomes itself an expensive

operation.† Efficiency will suffer rather than benefit from the indiscriminate storage of derived knowledge. Perhaps we should seek some middle ground between retaining everything and retaining nothing.

In addition to evaluating the capabilities and the efficiency of a shell, we also want to ask how informative its output is. When the derivation of a solution is presented or when a request for a new fact is explained, can the end user readily understand the information which is displayed? We have referred to the end user as a competent technician, but this can't be used to justify excessively cryptic output. At the same time, the output should be selective, if possible, rather than exhaustive, when the problem to be solved is a complex one. (Exhaustive clarifications have worked just fine for our little model problems.)

The specific needs of the system designing team must also be considered. How well do the Bacchus and Cactus formalisms allow us to express knowledge? How appropriate and convenient are the interactive operations of the augmentation phase? To what extent can these operations be automated? Certain modifications of the current language definitions will have to be made. Numbers (which play no role in the two model problems we have considered) are not even legal tokens of Bacchus or Cactus. There is no way to express disjunctions within rules in Bacchus or Cactus; separate rules must be written. Furthermore, it will be desirable for some of the information which is

† Indexing only allows us to locate a "relevant section" of the fact set rapidly. The search within a section for a matching fact is sequential.

currently obtained interactively during the augmentation phase to be directly expressible in the Bacchus or Cactus source file.

Some of the features contained in the current language definitions have never been tested. In particular, no variables appear in the Cactus source file for the Machine Translation problem.

The necessity for (or desirability of) various revisions and extensions to the current set of software tools for rule-based program development ought to be measured in the light of specific, new applications. The directions change should take will also become clearer as the software is introduced to a wider circle of users.

BIBLIOGRAPHY

BIBLIOGRAPHY

- Bratko, I. Prolog Programming for Artificial Intelligence, Addison-Wesley Publishers Limited, Workingham, England, 1986.
- Callero, M., Waterman, D.A., and Kipps, J. "TATR: A Prototype Expert System for Tactical Air Targeting", Rand Report R-3096-ARPA, The Rand Corporation, Santa Monica, CA, 1984.
- Hayes-Roth, F., Waterman, D.A., and Lenat, D.B., eds. Building Expert Systems, Addison-Wesley Publishing Company, Reading, MA, 1983.
- Marcus, M.P. A Theory of Syntactic Recognition for Natural Language, MIT Press, Cambridge, MA, 1980.
- McDermott, J. "R1: A Rule-Based Configurer of Computer Systems", Artificial Intelligence, vol. 19, pp. 39-88, 1982.
- Michie, D., Muggleton, S., Riese, C., and Zubrick, S. "Rulemaster: A Second-Generation Knowledge-Engineering Facility", Proceedings of the First Conference on Artificial Intelligence Applications, IEEE Computer Society, 1984.
- Nelson, W.R. "Reactor: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents", AAAI Proceedings, 1982.
- Samuel, A.L. "Some Studies in Machine Learning Using the Game of Checkers", IBM Journal of Research and Development, vol. 3, no. 3, 1959.
- Schenker, A.M. Beginning Polish, Vol. 1, Yale University Press, New Haven, CT, 1978.
- Shortliffe, E.H. Computer-Based Medical Consultations: MYCIN, Elsevier, New York, NY, 1976.
- Steele, G.L. Common Lisp: The Language, Digital Press, Bedford, MA, 1984.
- Stilman, G., Stilman, L., and Harkins, W.E. Introductory Russian Grammar, Xerox College Publishing, Lexington, MA, 1972.
- Waterman, D.A. A Guide to Expert Systems, Addison-Wesley Publishing Company, Reading, MA, 1986.
- Winston, P.H. Artificial Intelligence, Addison-Wesley Publishing Company, Reading, MA, 1984.

APPENDICES

APPENDIX A

Access to the Software

Access to the Software

All of the software described in this thesis has been submitted to the Computer Science Department at Montana State University. Persons interested in reviewing the source code or output produced by this software should contact the Computer Science Department Office.

The materials submitted to the CS Department have been organized into sections called subdirectories. The names of these subdirectories are:

- 1) bacchus, which contains software used to build rules files that will be compatible with the Backward Chaining Shell,
- 2) cactus, which contains software used to build rules files that will be compatible with the Condition Action Shell,
- 3) common-build, which contains software used to build rules files of either type,
- 4) backchain, which contains all software peculiar to the Backward Chaining Shell,
- 5) conduct, which contains all software peculiar to the Condition Action Shell,
- 6) comshell, which contains software that is used by both of the shells,
- 7) rule, which contains source files of rules, intermediate files, and completed rules files,
- 8) fact, which contains files of facts,
- 9) dribbles, which contains recordings of sessions with the software tools,

- 10) bugs, which contains instances of functions which have had to be discarded or revamped or which simply became obsolete as the development of this software progressed, and
- 11) reference, which contains reference material.

Each subdirectory is a collection of files. Files of source code contain Lisp functions (and declarations). A file called "function-index" inside the "reference" subdirectory[†] provides a listing of the functions contained in each file of source code.

[†] References to files will have the form: subdirectory-name/file-name. This particular file would thus be called: reference/function-index.

APPENDIX B

User's Guide to the Software Tools

User's Guide to the Software Tools

Building a File of Rules for the Backward Chaining Shell

The system designing team will formulate a set of antecedent-consequent rules expressing the knowledge relevant to their application. These rules are to be expressed in the Bacchus formalism and stored in a file. (Each comment within a Bacchus file is delimited by a semicolon and an end-of-line.)

In order to build a rules file to be used with the shell, it is necessary to enter the Lisp interpreter. Within the interpreter, load the rules file building software by typing: (load "bacchus/build-rules"). The interpreter will respond with the names of several files which are being loaded. All of these support the 'build-rules' programs. When the interpreter returns a prompt, invoke the program by typing: (build-rules). The program will ask for the name of the Bacchus source file and will print its characteristic prompt: ==> , at which point the user should enter the name of the source file without quotes.† If the program's attempt to open the file named by the user results in failure, the failure is reported, and a new request for a source file name is printed. This continues until either the user decides to quit trying or a file is successfully opened.

Once the source file has been opened, the translation begins. A successful translation is signalled by the message: **"Translation completed. Building support structures."** The detection of an error

† Leading blanks may also cause the filename to be improperly represented.

in the source file during translation is signalled by the printing of an error message. When the translation of a source file that contains errors is completed, the program may or may not signal an intent to proceed to build support structures. (In either event, users should interpret an error message as an indication that the translated set of rules is likely to have deviated from their intentions.)

After translation, the augmentation phase of the rules file building process is begun. At intervals within this phase, the user is given the opportunity to interrupt the work and let it be resumed at a later time. If the user does choose to interrupt, the current state of the rules file is stored in a file designated by the user. To resume work during a later session, the user will once again load bacchus/build-rules into the Lisp interpreter and then type:

(resume). The 'resume' program will ask the user for the name of the incomplete rules file and will attempt to load it. The incomplete rules file will contain information which will let the software know the point at which to resume the work of the augmentation phase.

There are five tasks for which the user's help is sought during the augmentation phase. These include: (i) defining subset-superset relations among the set names obtained from the Bacchus source file, (ii) determining which sets are closed and which are open, (iii) coining names for supersets formed by the union of sets which are linked by weakly matching proposition templates, (iv) specifying functional dependencies among the variables in proposition templates, and (v) making forms for queries to be selected by users of the Backward Chaining Shell. These tasks are discussed in Chapter 4.

Sample sessions have been recorded which illustrate how these rules file building tools are to be used. Tasks (i) and (ii) are illustrated in dribbles/build_riv_temp2. (See Appendix A for an explanation of the naming conventions used for source materials.) Task (iii) is illustrated in dribbles/build_riv_temp3. Task (iv) is illustrated in dribbles/build_riv_temp4, and task (v) is illustrated in dribbles/build_gen_rivers.

At the end of the augmentation phase, the program prints the following message to the screen:

```
The preparation of your rules file is now
complete. Please enter the name under which
your completed rules file will be stored.
==>
```

If the user chooses to interrupt the augmentation phase, the program prints a different message:

```
Your rules file has been only partially
prepared. Please enter the name under which
your temporary rules file will be stored.
==>
```

In either case, the user types in the name of the file which is to contain the results of this rules file building session.

Building a File of Facts

In order to build a file of facts, it is necessary to load the appropriate software into the Lisp interpreter. This can be done by typing: (load "comshell/build-facts"). The fact building program can then be initiated by typing: (build-facts). The fact building program seeks to build a fact set which is compatible with a given file of rules. When the program begins to execute, it asks the user for the name of such a file of rules. If the new fact set is to be an

extension to (or modification of) some existing set, the user may enter the name of the appropriate facts file as well. (Of course, the old facts file must also be compatible with the specified rules file.) The program loads (or attempts to load) the files named by the user.

Once the necessary files have been loaded, the main work of the program begins. A numbered list of proposition forms is displayed. The user either types the number of a form corresponding to a fact which needs to be added or removed or else the user requests to exit the program.

When the user selects a form, all the elements of the current fact set which match this form (if any) are displayed. The user may then choose to add a new fact of this form, to remove (by number) one of the existing facts, or to cease working with the given form. New facts are entered by specifying instantiations for the variables in the form. New facts which would cause the fact set to violate constraints are rejected, and an error message is printed.

When the user indicates that there are no more changes to be made, the program asks for the name of the file to which the new fact set should be written. The program reports its success in writing the facts to the file named and then terminates.

A sample run of the facts file builder can be reviewed in `dribbles/build_river_facts1`.

Running the Backward Chaining Shell

In order to load the Backward Chaining Shell into the Lisp interpreter, type: (load "backchain/expert"). Then, to start a session with the shell, type: (expert). A rules file and a facts file must also be loaded into the interpreter in order to conduct a session. The shell will ask for the names of any necessary files before proceeding to obtain a query.

To elicit a query, the shell prints (and numbers) the propositions in *FORMS_LIST. The user selects the proposition from this list which best expressed the "theme" of the intended query. If, for instance, the user wishes to find out which side of the continental divide Wyoming lies on, it will be necessary to recognize that "Landmark lies on the Direction side of the divide" is the relevant proposition form.

After the user selects this form, the shell will present some query forms stored in the rules file and will ask the user to pick one of them. The following is a list of such query forms:

1. Does a given landmark lie on the given side of the divide?
2. What landmarks lie on the given side of the divide?
3. On which side of the divide does a given landmark lie?
4. What landmarks lie on which sides of the divide?

For this query regarding Wyoming, the user might select query form #3 from the list above. In response, the shell will print the request: Enter the name of the specific landmark you have in mind. The user should then type: Wyoming. The shell will then ask the user

to confirm that the actual query is: **On which side of the divide does Wyoming lie?**[†] As soon as confirmation of the query has been obtained, backward chaining may begin.

Backward chaining proceeds (possibly with intermittent requests for additional information) until success or failure is achieved. The success or failure is then reported. (The report of a successfully drawn inference is accompanied by a derivation.)

After the first query has been processed in this way, a menu of options is presented to the user. The same menu is shown during sessions with the Condition Action Shell. It is shown below in Figure 7.

Figure 7

Options Presented During Session With End User

=====

1. Option SOLVE
Solve a problem, given the current fact set.
2. Option WHYNOT
Examine why a different solution to the preceding problem was not reached.
3. Option WHATIF
Consider what solution would have been reached if a new fact were added.
4. Option WHATIFNOT
Consider what solution would have been reached if a fact were removed.
5. Option FACTS
Obtain a listing of the current working fact set.

[†] The corresponding initial goal to be placed on the goal stack will be: (Wyoming lies on the Direction1 side of the divide)

Figure 7 (continued)

6. Option LOAD
Load a new file of facts.
 7. Option ERASE
Create a null fact set, so that all facts may be entered interactively.
 8. Option QUIT
Leave the Experimental Shell.
-

The WHYNOT, WHATIF, and WHATIFNOT options have been described in Chapter 4. The selection of the SOLVE option will trigger a new presentation of the menus for selecting a query.

Building a Rules File for the Condition Action Shell

The system designing team will formulate a set of condition-action rules expressing the knowledge relevant to their application. These rules are to be expressed in the Cactus formalism and stored in a file.

Load the rules file building software into the Lisp interpreter by typing: (load "cactus/build-rules"). When the interpreter returns a prompt, invoke the program by typing: (build-rules). The program will ask for the name of the Cactus source file and will print its characteristic prompt: ==>, at which point the user should enter the name of the source file without quotes. If the program's attempt to open the file named by the user results in failure, the failure is reported, and a new request for a source file name is printed. This continues until either the user decides to quit trying or a file is successfully opened.

Once the source file has been opened, the translation begins. The translator announces each new rules packet that it encounters. A successful translation is signalled by the message: **"Translation completed. Building support structures."**

After translation, the augmentation phase of the rules file building process is begun. At intervals within this phase, the user is given the opportunity to interrupt the work and let it be resumed at a later time. If the user does choose to interrupt, the current state of the rules file is stored in a file designated by the user. To resume work during a later session, the user must once again load cactus/build-rules into the interpreter. Then type: **(resume)**. The 'resume' program will ask the user for the name of the incomplete rules file and will attempt to load it. The incomplete rules file will contain information which will let the software know the point at which to resume the work of the augmentation phase.

There are generally four tasks for which the user's help is sought during the augmentation phase. These include: (i) categorizing the entities named in the rule conditions of the source file, (ii) declaring the values of sets, (iii) verifying the names of the "items" to be described by a solution, and (iv) formulating questions to be presented to end users when the Condition Action Shell needs to obtain additional information. These tasks are discussed in Chapter 5. Sample sessions have been recorded which illustrate how this interactive work proceeds. Task (i) is illustrated in dribbles/building_temp2. (See Appendix A for an explanation of the naming conventions used for source materials.) Task (ii) is

illustrated in dribbles/building_temp3. Task (iii) is illustrated in dribbles/building_temp4. Task (iv) is illustrated in dribbles/building_gentrans.

At the end of the augmentation phase, the program asks for the name to be given to the completed rules file and writes the rules (and associated data structures) to a file of that name.

Running the Condition Action Shell

In order to load the Condition Action Shell into the Lisp interpreter, type: (load "conduct/expert"). Then, to start a session with the shell, type: (expert). A rules file must be loaded into the interpreter in order to conduct a session. The shell will ask for the names of any necessary files before forward chaining commences.

Forward chaining will proceed in accordance with the "built-in agenda" of the rules file. When information is found to be missing, the shell will pose questions to the user. Failure or success in constructing a solution will be reported. A report of success will include a specification of the solution and a history of its derivation.

Then a menu of options is presented to the user. This is the same menu used for the Backward Chaining Shell, and the options are displayed in Figure 7. However, with the Condition Action Shell, the SOLVE option merely amounts to a repetition of the preceding problem, since each fact set specifies a single problem instance. New problems can be submitted to the shell through the use of the LOAD or ERASE option.

APPENDIX C

Sample Bacchus File

Sample Bacchus File

Figure 8 below shows a set of Bacchus rules that can be used as a source file for the Continental Divide problem.

Figure 8

Bacchus Rules for the Continental Divide Problem

```

=====
$ River1 flows toward $ SaltwaterBody1 IF
  $ River1 flows into $ River2 &
  $ River2 flows toward $ SaltwaterBody1 @

$ River1 flows toward $ SaltwaterBody1 IF
  $ River1 flows into $ SaltwaterBody1 @

$ River1 flows toward $ SaltwaterBody1 IF
  $ River1 flows into $ Lake1 &
  $ River2 flows out of $ Lake1 &
  $ River2 flows toward $ SaltwaterBody1 @

$ City1 lies on the $ Direction1 side of the divide IF
  $ River1 flows by $ City1 &
  $ River1 flows toward $ SaltwaterBody1 &
  $ SaltwaterBody1 lies on the $ Direction1 coast @

$ City1 lies on the $ Direction1 side of the divide IF
  $ City1 lies on the coast of $ SaltwaterBody1 &
  $ SaltwaterBody1 lies on the $ Direction1 coast @

$ Lake1 lies on the $ Direction1 side of the divide IF
  $ River1 flows out of $ Lake1 &
  $ River1 flows toward $ SaltwaterBody1 &
  $ SaltwaterBody1 lies on the $ Direction1 coast @

$ State1 lies on the $ Direction1 side of the divide IF
  $ State1 lies on the coast of $ SaltwaterBody1 &
  $ SaltwaterBody1 lies on the $ Direction1 coast @

$ State1 lies on the $ Direction1 side of the divide IF
  $ River1 flows thru $ State1 &
  $ River1 flows toward $ SaltwaterBody1 &
  $ SaltwaterBody1 lies on the $ Direction1 coast @

the divide passes thru $ Landmark1 IF
  PRIM EQUATE $ Landmark1 and $ City1 &
  $ City1 lies downstream from $ City2 &
  PRIM FAILURE @

```

Figure 8 (continued)

```

the divide passes thru $ Landmark1 IF
  PRIM DISTINGUISH $ Direction1 FROM $ Direction2 &
  $ Landmark1 lies on the $ Direction1 side of the divide &
  $ Landmark1 lies on the $ Direction2 side of the divide @

```

Let's consider the rule which allows us to conclude that a city lies on a certain side of the divide if that city lies on the coast of some saltwater body. Given the proper ordering of rules, this rule may help us to draw an inference more rapidly. For instance, we could use it to show that San Diego lies west of the divide without having to identify any river that flows by San Diego (and toward some ocean). The proof could therefore be drawn more quickly.

The problem with this, of course, is that very few users will be surprised to learn that San Diego lies west of the divide. Very few sessions with the rule-based system will actually make use of this rule, since most queries presumably would deal with cities lying deep in the interior of the country. As discussed in Chapter 4, an individual user can mask out questions which are anticipated to be irrelevant by loading his/her own definition of *DONT_ASK. But a rule which is not relevant to any of the system's users ought to be removed from the system entirely. (Even if it is not used in failed attempts to draw inferences, it will be a burden to the system's efficiency.) A full-blown system would keep track of the use made of individual rules, so that useless rules could be identified.

The Bacchus programmer should be aware of the relative usefulness of rules before incorporating them into a program. The programmer also has to give thought to the ordering of rules. (That is, the

"procedural meaning" of a Bacchus program is of concern to the programmer. (Bratko, 1986)). In the example under discussion, the fact that San Diego lies on the Pacific coast will not even be taken into account until after the shell has sought the name of a river which passes by San Diego.

The EQUATE primitive should also be mentioned. The variables mentioned in the EQUATE clause will receive the same instantiation as a result of the action of this primitive. Thus, its effect is the exact opposite of that of DISTINGUISH, which was discussed in Chapter 4.

APPENDIX D

Sample Cactus File

Sample Cactus File

Figure 9 below shows a set of Cactus rule packets that can be used as a source file for the Machine Translation Problem.

Figure 9

Cactus Rules for the Machine Translation Problem

StartPacket

ALL form

IF the form of the source verb is % finite THEN
 ASSIGN finite form to the translated verb &
 ACTIVATE MoodPacket @

IF the form of the source verb is % infinitival THEN
 ACTIVATE InfOnlyPacket @

IF the form of the source verb is % participial THEN
 ACTIVATE PartOnlyPacket @

MoodPacket

ALL mood

IF the mood of the source verb is % imperative THEN
 ASSIGN imperative mood to the translated verb &
 ASSIGN active voice to the translated verb @

otherwise

ACTIVATE VoicePacket @

IF the mood of the source verb is % subjunctive &
 the clause to be translated is % main THEN
 ASSIGN conditional mood to the translated verb &
 ACTIVATE Time2Packet &
 ACTIVATE AspectPacket @

IF the mood of the source verb is % subjunctive &
 the clause to be translated is % subordinate &
 the subordinate clause is % conditional THEN
 ASSIGN Subjunctive2 mood to the translated verb &
 ACTIVATE Time2Packet &
 ACTIVATE AspectPacket @

Figure 9 (continued)

IF† the mood of the source verb is % subjunctive &
 the clause to be translated is % subordinate &
 the translated verb of the superordinate clause % does support
 infinitival complementation THEN

ASSIGN infinitive form to the translated verb &
 EXTRACT mood field of the translated verb &
 ACTIVATE Time2Packet &
 ACTIVATE AspectPacket @

IF the mood of the source verb is % subjunctive &
 the clause to be translated is % subordinate &
 the translated verb of the superordinate clause % does support
 subjunctive complementation THEN

ASSIGN Subjunctive mood to the translated verb &
 ACTIVATE AspectPacket @

IF the mood of the source verb is % subjunctive &
 otherwise

ASSIGN conditional mood to the translated verb &
 ACTIVATE Time2Packet &
 ACTIVATE AspectPacket @

IF the mood of the source verb is % indicative &
 the clause to be translated is % subordinate &
 the verb of the superordinate clause % is sensory &
 the aspect of the source verb is % imperfective THEN

ASSIGN present-participle form to the translated verb &
 EXTRACT mood field of the translated verb @

IF the mood of the source verb is % indicative &
 the clause to be translated is % subordinate &
 the verb of the superordinate clause % is sensory &
 the aspect of the source verb is % perfective THEN

ASSIGN bare-infinitive form to the translated verb &
 EXTRACT mood field of the translated verb @

IF the mood of the source verb is % indicative &
 otherwise

ASSIGN indicative mood to the translated verb &
 ACTIVATE TensePacket &
 ACTIVATE AspectPacket @

† This rule models sentence pairs such as: "Chciałem, żebyś
 poszła ten list, jak byś na pocztę." --> "I wanted you to mail that
 letter while you were at the post office."

Figure 9 (continued)

Time2Packet

All tense-form

IF the clause to be translated is % subordinate &
 the translated verb of the superordinate clause % does support
 infinitival complementation &
 the time reference of the verb to be translated % does strictly
 precede that of the superordinate clause THEN
 ASSIGN perfect tense-form to the translated verb @

IF† the clause to be translated is % subordinate &
 the translated verb of the superordinate clause % doesn't support
 infinitival complementation &
 the time reference % is restricted to the past THEN
 ASSIGN perfect tense-form to the translated verb @

IF the clause to be translated is % subordinate &
 otherwise
 ASSIGN simple tense-form to the translated verb @

IF the clause to be translated is % main &
 the time reference % is restricted to the past THEN
 ASSIGN perfect tense-form to the translated verb @

IF the clause to be translated is % main &
 otherwise
 ASSIGN simple tense-form to the translated verb @

TensePacket

All tense

IF the clause to be translated is % subordinate &
 the clause % is embedded in an indirect speech construction &
 the tense of the superordinate clause is % past THEN
 TRANSFORM tense of the translated verb by sequence-of-tense
 @

IF the tense of the source verb is % present &
 the time reference % does extend from some point in the past up to
 the present THEN
 ASSIGN present-perfect tense to the translated verb @

† This awkwardly phrased rule is intended to model sentence pairs such as: "Gdybys' tu wtedy byla, to bys' inaczej zareagowala"
 --> "If you had been here then, you would have reacted differently."

Figure 9 (continued)

IF the tense of the source verb is % present &
otherwise

ASSIGN present tense to the translated verb @

IF the tense of the source verb is % past &
context % does emphasize that the time reference strictly precedes
that of a neighboring clause THEN

ASSIGN past-perfect tense to the translated verb @

IF† the tense of the source verb is % past &
context % does indicate the current applicability of the
proposition THEN

ASSIGN present-perfect tense to the translated verb @

IF the tense of the source verb is % past THEN
ASSIGN past tense to the translated verb @

IF the tense of the source verb is % future &
the clause to be translated is % subordinate &
the subordinate clause is % conditional THEN

ASSIGN present tense to the translated verb @

IF†† the tense of the source verb is % future &
the clause to be translated is % subordinate &
the subordinate clause is % temporal THEN

ASSIGN present tense to the translated verb @

IF the tense of the source verb is % future &
otherwise

ASSIGN future tense to the translated verb @

AspectPacket

ALL aspect

IF the aspect of the source verb is % imperfective &
context % does indicate a durative meaning &
the translated verb % does support continuous aspect THEN

ASSIGN continuous aspect to the translated verb @

† This rule models sentence pairs such as: "Czy kiedykolwiek
widziałeś ten film?" --> "Have you ever seen this movie?"

†† This rule models sentence pairs such as: "Jak będziesz jechał
do Montany, na pewno zabierz narty." --> "When you come to Montana, be
sure to bring your skis."

Figure 9 (continued)

IF† the aspect of the source verb is % imperfective &
 context % doesn\'t indicate a durative meaning &
 context % does indicate an iterative meaning &
 the tense of the source verb is % past THEN
 ASSIGN iterative aspect to the translated verb @

IF the aspect of the source verb is % imperfective &
 otherwise
 ASSIGN simple aspect to the translated verb @

IF the aspect of the source verb is % perfective THEN
 ASSIGN simple aspect to the translated verb &
 TAG-ON†† the translated verb ~ perfective meaning may be
 expressed by a particle such as "up" or "out" @

^
 VoicePacket
 ALL Voice

IF the voice of the source verb is % active THEN
 ASSIGN active voice to the translated verb @

IF the voice of the source verb is % impersonal-reflexive &
 there % is a dative argument in the role of beneficiary THEN
 TAG-ON the translated verb ~ convert the dative to subject
 of the translated clause &
 ASSIGN active voice to the translated verb @

IF the voice of the source verb is % impersonal-reflexive &
 there % isn't a dative argument in the role of beneficiary &
 there % is a direct or indirect object THEN
 TAG-ON the translated verb ~ convert the object to subject
 of the translated clause &
 ASSIGN passive voice to the translated verb @

IF the voice of the source verb is % impersonal-reflexive &
 otherwise
 TAG-ON the translated verb ~ introduce a "dummy" subject &
 ASSIGN active voice to the translated verb @

IF the voice of the source verb is % passive THEN
 ASSIGN passive voice to the translated verb @

† This rule models sentence pairs such as "Jeździliśmy na kamping, jak byłem dzieckiem." --> "We used to go camping when I was a kid."

†† The tag-on is used to model sentence pairs such as: "Wypij mleko" --> "Drink your milk up."

Figure 9 (continued)

InfOnlyPacket

IF† context % does indicate an imperative meaning directed to an
 undefined audience THEN
 ASSIGN finite form to the translated verb &
 ASSIGN imperative mood to the translated verb @

PartOnlyPacket

IF there % is a direct or indirect object THEN
 TAG-ON the translated verb ~ convert the object to subject
 &
 ASSIGN finite form to the translated verb &
 ASSIGN indicative mood to the translated verb &
 TAG-ON the translated verb ~ tense may be inferred from
 context &
 ASSIGN passive voice to the translated verb @

otherwise††

TAG-ON the translated verb ~ introduce a "dummy" subject &
 ASSIGN finite form to the translated verb &
 ASSIGN indicative mood to the translated verb &
 TAG-ON the translated verb ~ tense may be inferred from
 context &
 ASSIGN passive voice to the translated verb @

† This rule models sentence pairs such as: "Nie wychylać się
 przez okno" --> "Don't lean out the window."

†† This rule models sentence pairs such as: "Mówiono, że będą
 dalsze podwyżki cen." --> "It is said that there will be more price hikes."

MONTANA STATE UNIVERSITY LIBRARIES



3 1762 10023757 5

