The Handbook of Artificial Intelligence

Volume II

Edited by

Avron Barr

and

Edward A. Feigenbaum

Department of Computer Science
Stanford University

HEURISTECH PRESS
Stanford, California

WILLIAM KAUFMANN, INC.
Los Altos, California

CONTENTS OF VOLUME II

List of Contributors / ix

Preface / xi

VI. Programming Languages for AI Research / 1

A. Overview / 3
B. LISP / 15
C. AI programming-language features / 30
1. Overview / 30
2. Data structures / 34
3. Control structures / 45
4. Pattern matching / 58
5. Programming environment / 65
D. Dependencies and assumptions / 72

VII. Applications-oriented AI Research: Science / 77

A. Overview / 79
B. TEIRESIAS / 87
C. Applications in chemistry / 102
1. Chemical analysis / 102
2. The DENDRAL programs / 106
a. Heuristic DENDRAL / 106
b. CONGEN and its extensions / 111
c. Meta-DENDRAL / 116
3. CRYSALIS / 124
4. Applications in organic synthesis / 134
D. Other scientific applications / 143
1. MACSYMA / 143
2. The SRI Computer-based Consultant / 150
3. PROSPECTOR / 155
4. Artificial Intelligence in database management / 163

VIII. Applications-oriented AI Research: Medicine / 175

A. Overview / 177

B. Medical systems / 184
1. MYCIN / 184
2. CASNET / 193
3. INTERNIST / 197
4. Present Illness Program / 202
5. Digitalis Therapy Advisor / 206
6. IRIS / 212
7. EXPERT / 217

IX. Applications-oriented AI Research: Education / 223

A. Overview / 225
B. ICAI systems design / 229
C. Intelligent CAI systems / 236
1. SCHOLAR / 236
2. WHY / 242
3. SOPHIE / 247
4. WEST / 254
5. WUMPUS / 261
6. GUIDON / 267
7. BUGGY / 279
8. EXCHECK / 283
D. Other applications of AI to education / 291

X. Automatic Programming / 295

A. Overview / 297
B. Methods of program specification / 306
C. Basic approaches / 312
D. Automatic programming systems / 326
1. PSI and CHI / 326
2. SAFE / 336
3. The Programmer's Apprentice / 343
4. PECOS / 350
5. DEDALUS / 355
6. Protosystem I / 364
7. NLPQ / 370
8. LIBRA / 375

Bibliography for Volume II / 381

Indexes for Volume II / 403

LIST OF CONTRIBUTORS

Non-Stanford affiliations indicated if known.

Chapter Editors

Janice Aikins (Hewlett-Packard)
James S. Bennett
Victor Ciesielski (Rutgers U)
William J. Clancey
Paul R. Cohen
James E. Davidson
Thomas G. Dietterich
Bob Elschlager (Tymshare)
Lawrence Fagan
Anne v.d.L. Gardner
Takeo Kanade (CMU)
Jorge Phillips (Kestrel)
Steve Tappel
Stephen Westfold (Kestrel)

Contributors

Robert Anderson (Rand)
Douglas Appelt (SRI)
David Arnold
Michael Ballantyne (U Texas)
David Barstow (Schlumberger)
Peter Biesel (Rutgers U)
Lee Blaine (Lockheed)
W. W. Bledsoe (U Texas)
David A. Bourne (CMU)
Rodney Brooks (MIT)
Bruce G. Buchanan
Richard Chestek
Kenneth Clarkson
Nancy H. Cornelius (CMU)
James L. Crowley (CMU)
Randall Davis (MIT)
Gerard Dechen
Johan de Kleer (Xerox)
Jon Doyle (CMU)
R. Geoff Dromey (U Wollongong)
Richard Duda (Fairchild)
Robert S. Engelmore (Teknowledge)
Ramez El-Masri (Honeywell)
Susan Epstein (Rutgers U)
Robert E. Filman (Hewlett-Packard)
Fritz Fisher (Ramtek)
Christian Freksa (Max Plank, Munich)
Peter Friedland
Hiromichi Fujisawa (CMU)
Richard P. Gabriel
Michael R. Genesereth
Neil Goldman (ISI)
Ira Goldstein (Hewlett-Packard)
George Heidorn (IBM)
Martin Herman (CMU)
Annette Herskovits
Douglas Hofstadter (Indiana U)
Elaine Kant (CMU)
Fuminobu Komura (CMU)
William Laaser (Xerox)
Douglas B. Lenat
William J. Long (MIT)
Robert London
Bruce D. Lucas (CMU)
Pamela McCorduck
Mark L. Miller (Computer Thought)
Robert C. Moore (SRI)
Richard Pattis
Stanley J. Rosenschein (SRI)
Neil C. Rowe
Gregory R. Ruth (MIT)
Daniel Sagalowicz (SRI)

Contributors (continued)

Behrokh Samadi (UCLA)
William Scherlis (CMU)
Steven A. Shafer (CMU)
Andrew Silverman
David R. Smith (CMU)
Donald Smith (Rutgers U)
Phillip Smith (U Waterloo)
Reid G. Smith (Schlumberger)
William R. Swartout (ISI)
Steven L. Tanimoto (U Washington)
Charles E. Thorpe (CMU)
William van Melle (Xerox)
Richard J. Waldinger (SRI)
Richard C. Waters (MIT)
Sholom Weiss (Rutgers U)
David Wilkins (SRI)
Terry Winograd

Reviewers

Harold Abelson (MIT)
Saul Amarel (Rutgers U)
Robert Balzer (ISI)
Harry Barrow (Fairchild)
Thomas Binford
Daniel Bobrow (Xerox)
John Seely Brown (Xerox)
Richard Burton (Xerox)
Lewis Creary
Andrea diSessa (MIT)
Daniel Dolata (UC Santa Cruz)
Lee Erman (ISI)
Adele Goldberg (Xerox)
Cordell Green (Kestrel)
Norman Haas (Symantec)
Kenneth Kahn (MIT)
Jonathan J. King (Hewlett-Packard)
Casimir Kulikowski (Rutgers U)
John Kunz
Brian P. McCune (AI&DS)
Jock Mackinlay
Ryszard S. Michalski (U Illinois)
Donald Michie (U Edinburgh)
Thomas M. Mitchell (Rutgers U)
D. Jack Mostow (ISI)
Nils Nilsson (SRI)
Glen Ouchi (UC Santa Cruz)
Ira Pohl (UC Santa Cruz)
Arthur L. Samuel
David Shur
Herbert A. Simon (CMU)
David E. Smith
Dennis H. Smith (Lederle)
Mark Stefik (Xerox)
Albert L. Stevens (BBN)
Allan Terry
Perry W. Thorndyke (Perceptronics)
Paul E. Utgoff (Rutgers U)
Donald Walker (SRI)
Harald Wertz (U Paris)
Keith Wescourt (Rand)

Production

Max Diaz
David Eppstein
Lester Ernest
Marion Hazen
Janet Feigenbaum
David Fuchs
José L. González
Dianne G. Kanerva
Jonni M. Kanerva
Dikran Karagueuzian
Arthur M. Keller
Barbara R. Laddaga
Roy Nordblom
Thomas C. Rindfleisch
Ellen Smith
Helen Tognetti
Christopher Tucci

PREFACE

THE PROJECT to write the *Handbook of Artificial Intelligence* was born in the mid-1970s, at a low ebb in the fortunes of the field. AI, in our view, had made remarkable contributions to one of the grandest of scientific problems—the nature of intelligence, in humans and in artifacts. Yet it had failed to communicate its concepts, its working methods, its techniques, and its successes to the broad scientific and engineering communities. The work remaining to be done was almost limitless, but the number of practitioners was few. If AI were to succeed, it would have to communicate more clearly and widely to others in science and engineering. So we thought, and thus were we motivated to assemble and edit these volumes.

In the last few years, we have seen an astonishing change in the perception and recognition of AI as a science and as a technology. Many large industrial firms have committed millions of dollars to the establishment of AI laboratories. The Japanese have even committed a national project, the so-called Fifth Generation, to the engineering of "knowledge information processing machines," that is, AI-oriented hardware and software. Newspaper, magazine, and broadcast features on AI are common. A lively new professional society, the American Association for Artificial Intelligence, has been formed. And university graduate-school enrollments in AI are booming. Indeed, Volume I of the *Handbook* was the main selection, in August 1981, of one of the major book clubs; it is now undergoing its second printing and is being translated into Japanese.

The crisis we face now is a crisis of success, and many wonder if the substance of the field can support the high hopes. We believe that it can, and we offer the material of the three massive volumes of the *Handbook* as testimony to the strength and intellectual vigor of the enterprise.

The five chapters in this volume cover three subfields of AI. Chapter VI, on *AI programming languages*, describes the kinds of programming-language features and environments developed by AI researchers. These languages are, like all programming languages, not only software tools with which the many different kinds of AI programs are constructed, but also "tools of thought" in which new ideas and perspectives on the understanding of cognition are first explored. Of note here is the extended discussion of LISP—by far the most important tool of either kind yet invented in AI.

Chapters VII through IX are about *expert systems*, in science, medicine, and education. These systems, which vary widely in structure and behavior, all focus on one important methodology, called the *transfer of expertise*. Early in AI's history, researchers agreed that high performance on difficult problems would require large amounts of *real-world knowledge*, the knowledge that a

human expert in a particular domain has extracted from his (or her) experience with the problems he solves. The idea of expert-systems research was to find ways of transferring the necessary kinds and quantities of knowledge from human experts to AI systems. This technology has advanced to the point where these systems can perform at the level of human experts and may be available commercially in the next few years.

Finally, Chapter X reviews the area of AI research called *automatic programming*. This research has focused on systems that can produce a program from some "natural" description of what it is to do or that attend to some other important aspect of programming, like verifying that a program does what it was intended to do. For example, some automatic-programming systems produce simple programs from examples of input/output pairs or from English *specifications* of the program's intended behavior. But there is a much deeper purpose to automatic-programming research than just easing the burden of the programmer. To achieve their pragmatic goals, these systems must *understand* programs just as other AI systems understand language or chess or medical diagnosis. They must reason about programs and about themselves as programs, and, as we discuss in Chapter X, this is a central and characteristic feature of many AI systems.

Acknowledgments

The chapter on AI programming languages was first drafted by Steve Tappel and Stephen Westfold. Johann de Kleer and Jon Doyle contributed the excellent article on dependencies and assumptions. A thorough review and additional material were supplied by Christian Freksa. Other reviewers included Robert Balzer, Cordell Green, Brian McCune, and Harald Wertz.

The chapter on scientific-applications research was edited by James Bennett, Bruce Buchanan, Paul Cohen, and Fritz Fisher. Original material and comments were contributed by, among others, James Davidson, Randall Davis, Daniel Dolata, Richard Duda, Robert Engelmore, Peter Friedland, Michael Genesereth, Jonathan King, Glen Ouchi, and Daniel Sagalowicz.

Chapter VIII, on research in medical applications of AI, was edited by Victor Ciesielski and his colleagues at Rutgers University. James Bennett and Paul Cohen continued work on the material. Others who contributed to or reviewed this material include Saul Amarel, Peter Biesel, Bruce Buchanan, Randall Davis, Casimir Kulikowksi, Donald Smith, William Swartout, and Sholom Weiss.

The educational-applications chapter was compiled by Avron Barr and William Clancey, and, once again, James Bennett and Paul Cohen continued the editing process. Contributors and reviewers included Harold Abelson, Lee Blaine, John Seely Brown, Richard Burton, Andrea diSessa, Adele Goldberg, Ira Goldstein, Kenneth Kahn, Mark Miller, Neil Rowe, Albert Stevens, and Keith Wescourt.

Finally, the automatic-programming chapter was edited by Bob Elschlager and Jorge Phillips, working from original material supplied by David Barstow, Cordell Green, Neil Goldman, George Heidorn, Elaine Kant, Zohar Manna, Brian McCune, Gregory Ruth, Richard Waldinger, and Richard Waters.

The design and production of the volume were the responsibility of Dianne Kanerva, our professional editor, and José González. The book was typeset with TEX, Donald Knuth's system for mathematical typesetting, by David Eppstein, Janet Feigenbaum, José González, Jonni Kanerva, Dikran Karagueuzian, and Barbara Laddaga. Our publisher William Kaufmann and his staff have been generous with their patience, help, and willingness to experiment.

The Advanced Research Projects Agency of the Department of Defense and the Biotechnology Resources Program of the National Institutes of Health supported the *Handbook* project as part of their longstanding and continuing efforts to develop and disseminate the science and technology of AI. Earlier versions of material in these volumes were distributed as technical reports of the Department of Computer Science at Stanford University. The electronic text-preparation facilities available to Stanford computer scientists on the SAIL, SCORE, and SUMEX computers were used throughout the writing and production of the *Handbook*.

FUZZY. The most recently developed language included in this study illustrates current work in AI programming languages. The design of the FUZZY language (Le Faivre, 1977) was motivated by the theory of fuzzy sets (Zadeh, 1965; Gupta, Saridis, and Gaines, 1977), a generalization of Boolean set theory that allows for "graded" set membership (rather than all-or-none). For many natural-language concepts, for instance, there is no sharp boundary between situations for which the concept applies and situations for which it does not. Consider, for example, the concept *young*. We may say that people under 10 years of age are young and those above 60 years are not young. However, there is no particular day at which a person's age switches from "young" to "not young"; rather, this is a gradual transition. In fuzzy set theory, the concept of young in this context is expressed by a "membership function" representing the degree to which a person of a particular age can be considered to be young.

Many AI systems deal explicitly with fuzzy information (see, e.g., the *certainty factor* in MYCIN, Article VIII.B1), and FUZZY is designed to facilitate certain types of reasoning with fuzzy sets. It has been used for various AI projects, including the AIMDS/BELIEVER system at Rutgers University (Schmidt and Sridharan, 1977) and HAM-RPM, a knowledge-based conversationalist at the University of Hamburg (Wahlster, 1977).

Logic Programming

Two languages based on first-order predicate calculus are PROLOG and FOL. PROLOG programs consist of "axioms" in first-order logic together with a theorem to be proved. The axioms are restricted to implications with the left- and right-hand sides in *horn-clause* form. If the theorem contains existentially quantified variables, the system will return instantiations of these that make the theorem true (if such exist) using methods developed from those of QA3. The style of programming is similar to that demonstrated in QA3 and, to a lesser extent, PLANNER. Automatic backtracking is used, but the programmer may add annotation to control the order in which clauses and axioms are considered. A compiler has been implemented for PROLOG that allows programs in a variety of domains to be executed in about the same time as corresponding compiled LISP programs. (See Clocksin and Mellish, 1981; Warren, Pereira, and Pereira, 1977.)

Another direction of logic—the uses of meta-theory—has been explored in FOL (Weyrauch, 1979). This program is primarily a proof checker that accepts logic statements and proof-step commands that can be carried out and tested for correctness. However, it provides a powerful theory-manipulating structure that allows the building of arbitrary meta-theory. Thus, for example, a theorem may be proved not only within a theory but also with the help of the meta-theory. Proving a theorem by going through the meta-theory corresponds closely to executing a procedure to produce the theorem.

FUZZY

Much like PLANNER, the programming language FUZZY maintains a database of assertions. However, FUZZY assertions include a *Z-value* indicating the degree of certainty, for example, `((CHANCE OF RAIN) . 0.30)`. FUZZY maintains an associative network of assertions quite similar to that of PLANNER and CONNIVER. Any arbitrary LISP list structure may be entered into this net. In addition, an assertion may have a Z-value associated with it, if desired. The Z-value of the assertions can be used to control success and failure of retrieval or subsequent actions.

FUZZY has a context mechanism that activates and deactivates associative nets of assertions. It is also possible to save the state of the entire system in order to allow for later restoration. Functions are available to compute differences between states and to add differences to a state. State changes can be set, if desired, so that they cannot be undone by a subsequent restoration. This feature is useful to control backtracking. Several FUZZY primitives exist in backtrackable and in finalizing versions to give the programmer easy control over the global control mechanisms.

Summary

Data types. The languages vary considerably in the number and kinds of data types they offer. Basic LISP is at one extreme: It began with exactly one data type, with a few supplementary ones added later. Advantages of a sparse set of data types accrue mostly to the writers of LISP compilers and interpreters, whose job is simplified because there are fewer operations and less need for type conversion. Also, the relatively small compilers and interpreters that are produced help conserve available core space. From the *user's* point of view, however, there is little to recommend such a small set of data types, except that it (almost) removes the need for type declarations.

Later versions of LISP, such as INTERLISP and MACLISP, and to an even greater extent the languages QLISP and POP-2, have provided rich sets of data types and the access functions that go with them. Programming is easier because the data structures can more closely mirror the programmer's ideas, and type checking becomes available. Efficiency is improved because the standard data types can be implemented closer to the machine level than equivalent structures built of more primitive units. For example, a SAIL or POP-2 record uses fixed-offset fields and avoids the overhead of the pointers needed in an equivalent LISP list structure.

A related issue is whether to allow user-defined data types. The advantages are similar to those of having many data types, but when user-defined data types have been allowed, as in POP-2, they have not found great utility. Probably the main reason is simply the extra effort required from the user

database update is done. Demons that are waiting for such a message will be activated, simulating a direct activation of demons by the database update.

Coroutining is a special case of multiprocessing, in which only one process is active at any time. The coroutining primitives CREATE, TERMINATE, ACTIVATE, and SUSPEND can all be implemented using SAIL's message-passing mechanism.

POP-2

The POP–2 control structure is much like that of LISP. The basic language is very simple, partly in the interests of fast compilation, and contains none of the specialized AI control structures found in PLANNER, CONNIVER, and QLISP. Some of these features, including coroutines and multiprocessing primitives, are available in POP–2 libraries. The POPLER library (Davies et al., 1973) implemented the spaghetti stack, backtracking, database demons, and pattern matching in the manner of PLANNER.

The basic POP–2 language does provide for the use of generators to construct *dynamic lists*. The programmer defines a function of no arguments, say, F, and applies the POP–2 function FNTOLIST to F. The result is the list LF of values that F produces on successive calls. Of course, F has to read and change a free variable, representing the state of the generation, or else every element of LF would be the same. Now comes the interesting part. The program can operate on LF just as on ordinary (static) lists. But, in fact, LF is not all there; it starts empty and new elements are added onto the back of it only as needed. This means that LF can be conceptually very large or even infinite, and it does not matter so long as only the initial part of it is used.

Dynamic lists allow the programmer to abstract away from the details of how a list is produced, whether it is computed once and for all or is extended when needed. Similarly, the *memo function* allows abstraction from the details of how a function is computed. Memo functions are provided by one of the POP–2 libraries. The name comes from the notion that a function, if it is called several times with the same argument, would do better to "make a memo of it" (the answer) and store it in a lookup table.

FUZZY

FUZZY procedures have a more general global-control mechanism than PLANNER theorems have. The procedure demons are given control not only upon failure of an expression (as in MICRO–PLANNER) but also after successful termination of a top-level expression. This makes it possible to evaluate globally the results returned by the expressions of a procedure. With each procedure a variable is associated that maintains an "accumulated Z-value" for the demon's calculations. The procedure demon is called a last time when the procedure is exited in order to make any necessary final computations (e.g., concluding statistics).

There are several levels at which information is accessed in a knowledge base:

1. Explicitly available information is requested.
2. An explicit procedure is invoked to retrieve the desired information.
3. A goal is specified and the system is left to decide how to achieve it.

All three methods are possible in FUZZY:

1. The FETCH statement retrieves assertions that are explicitly stored in the associative net by pattern matching.
2. FUZZY procedures can be called by name.
3. If FUZZY procedures have been stored in the associative net, they can be invoked by pattern matching through the DEDUCE statement. This relieves the programmer of keeping track of which particular procedures may be used to achieve a certain task and allows the easy addition of new procedures to the associative net that can be utilized automatically by existing programs without change.
4. The GOAL statement combines the FETCH and DEDUCE statements. It first checks whether the desired information is explicitly available in the net of assertions. If it fails, it then attempts to deduce the goal by invoking DEDUCE procedures that match the specified pattern.

In addition, FUZZY supports ASSERT and ERASE procedures, which are invoked automatically when assertions of corresponding patterns are added and removed, respectively, from the associative net.

The following program illustrates how FUZZY may deal with both vague and incomplete information. The vagueness is expressed here by Z-values associated with assertions. Incomplete information in this example is manifested by the absence of useful assertions. This "missing information" does not force the procedure into failure, but rather lowers the confidence in the result obtained by the procedure:

```
=== NET ===
((CHANCE OF RAIN) . 0.8)
((DRYNESS DESIRED) . 0.7)
((BLUE SKY) . 0.4)

=== DEDUCE ===
(PROC NAME: UMBRELLA DEMON: CONFIDENCE (RAIN PROTECTION)
      (BIND ?SK (FETCH ((*OR CLEAR BLUE) SKY)))
      (BIND ?BU (FETCH (BURDENSOME UMBRELLA)))
      (BIND ?DD (FETCH (? DESIRED)))
      (IF (ZAND THRESH: 0.9 (ZNOT !SK) !DD !BU)
    THEN: (SUCCEED @"STAY HOME!")
    ELSE: T)
      (BIND ?CR (FETCH (CHANCE ??)))
```

```
      (IF (MINUSP (DIFFERENCE (PLUS (ZVAL !CR) (ZVAL !DD))
                              (PLUS (ZVAL !SK) (ZVAL !BU))))
    THEN: (SUCCEED @"DON'T TAKE UMBRELLA" ZACCUM)
    ELSE: (SUCCEED @"TAKE UMBRELLA" ZACCUM)))

==========
(DEFPROP CONFIDENCE
 (LAMBDA (RESULT THRESHOLD C-LEVEL)
  (COND [(EQ RESULT FAIL) (COND [(*GREAT C-LEVEL 0.25)
                                 (DIFFERENCE C-LEVEL 0.25)]
                                [T (FAIL)])]
        [(EQ RESULT DONE) C-LEVEL]
        [(*LESS (ZVAL RESULT) THRESHOLD) (FAIL)]
        [T C-LEVEL]))
EXPR)
        (;   RESULT    = result of last top-level expression
             THRESHOLD = criterion for forcing procedure to fail
             C-LEVEL   = current confidence level)
```

The program listing includes the associative net, containing some declarative knowledge about a potential umbrella carrier and his situation. Next is the procedural associative net containing a DEDUCE procedure to give advice whether or not to carry an umbrella in a given situation. Finally, there is a LISP procedure that is used by the DEDUCE procedure UMBRELLA as a procedure demon. The procedure UMBRELLA uses assertions and their modifiers to calculate the projected payoff for carrying an umbrella. The demon CONFIDENCE watches this calculation and determines a confidence measure for the result obtained by UMBRELLA. This is done as follows. UMBRELLA looks for four types of assertions in the associative net:

1. information about the blueness or the clearness of the sky,
2. information about the burden of carrying an umbrella,
3. information about a desired goal that can be satisfied with an umbrella, and
4. information about the chance that an event would occur that would make an umbrella desirable.

The most reliable advice can be given by UMBRELLA if all four pieces of information can be found. If a piece of information cannot be found (i.e., if the corresponding FETCH returns FAIL), the demon reduces the confidence level ZACCUM, which is returned as the Z-value of UMBRELLA. Observe that the Z-value can be used in a single program to do different kinds of qualifications.

Summary

In the chronological sequence PLANNER, CONNIVER, QLISP, we observe an increase in the variety of control structures and in the programmer's access

These interact in a manner best shown by example:

```
FOREACH X, Y SUCH THAT X IN AnimalSet AND Gregarious(X) AND
                    Desert(Y) AND Range ⊗ X ≡ Y DO PRINT(X);
```

The conjunctive conditions in the FOREACH statement are processed left to right. In this example, a set-membership pattern is leftmost, so the system chooses some X in the set **AnimalSet**. Then X is tested to determine whether it satisfies the predicate GREGARIOUS (just a Boolean function). If GREGARIOUS returns FALSE, another X in **AnimalSet** is chosen, but if it returns TRUE, the process continues: The matching procedure DESERT generates a Y and the database is checked to determine whether **Range** ⊗ X ≡ Y. (If not, another DESERT is generated by the matching procedure.) X-Y pairs that meet all the conditions are passed on to the action part of the FOREACH statement (following the DO), which in this case consists merely of printing X. The net effect of this FOREACH, then, is to print out all gregarious animals that live in a desert.

In general, a FOREACH statement can include any number of variables and can have any number of conjoined conditions. The ordering of the conditions is critical for efficiency; for instance, if we had put the triple **Range** ⊗ X ≡ Y first in our example, the FOREACH statement would first find all appropriate triples in the associative database and then try each X-Y pair on the other conditions. (A matching procedure, when its variable is already bound, behaves like a Boolean function.) If the database includes a large amount of information concerning the ranges of animals, this would be highly inefficient. (Note that consequent theorems in PLANNER have a similar property, namely, that the order in which subgoals are listed can drastically affect the number of alternatives examined.)

POP-2

The basic POP-2 programming language does not have pattern matching. As with POP-2's control-structure features (Article VI.C3), the pattern-matching facilities are provided by various library packages that implement facilities similar to those in PLANNER, CONNIVER, and SAIL.

FUZZY

As in PLANNER and CONNIVER, a FUZZY variable is assigned a value through pattern matching. For example,

```
(MATCH (?X ??Y) ((A B) C D))
```

binds the FUZZY variable `!X` to `(A B)` and `!Y` to `(C D)`. A greater selection of functions than in PLANNER and CONNIVER is available for restricting the structure of the pattern or the set of items that can match successfully; for example,

```
(*R ?OBJECT (FETCH (RED !OBJECT)))
```

will only match an object that is known to be red.

Summary

Once again, in the chronological sequence PLANNER, CONNIVER, QLISP, we see a general increase in sophistication of pattern matching and the range of its applications. PLANNER patterns are used implicitly to fetch assertions from the database and to choose a function (theorem) to invoke, but they are limited in their structure and cannot be used explicitly to analyze the structure of data items. QLISP patterns are, by comparison, very general, and much of the language's power depends on them. They serve as a major method for analyzing data, not just in the sense of extracting parts, but of performing quite complicated tests and searches as well.

Not surprisingly, pattern matching is expensive. In almost any particular case, the pattern-match algorithm will be more general than is really required, implying that replacing it with ad hoc code would yield a speedup. When the pattern contains segment variables (as in QLISP), the slowdown is especially severe. In this connection, it is interesting to note that QLISP is termed by its designers (Sacerdoti et al., 1976) a "language for the interactive *development* of complex systems" with the explicit intention that once a QLISP program works, the user can convert it to pure INTERLISP. This can even be accomplished in stages in the QLISP environment, because QLISP and INTERLISP can be freely mixed.

Some pattern-matching capabilities are not offered in any AI programming language. For one thing, all current languages do exact structural matches. A very desirable feature would be "best match" capability: Instead of matching exactly or failing, the matcher would do the best it could and return information on the points it could not find a match on (see Bobrow and Winograd, 1977, for speculations along this line).