AD-A096 042 SYRACUSE UNIV NY SCHOOL OF COMPUTER AND INFORMATION --ETC F/G 9/2
LOGIC PROGRAMMING IN LISP.(U)
JAN 81 J A ROBINSON, E E SIBERT F30602-77-C-0235
UNCLASSIFIED RADC-TR-80-379-VOL-1 NL

1 OF 2
AD
A096042

LEVEL II

12

RADC-TR-80-379, Vol I (of five)
Final Technical Report
January 1981

AD A096042

LOGIC PROGRAMMING IN LISP

LOGIC PROGRAMMING IN LISP

Syracuse University

J. A. Robinson
E. E. Sibert

DTIC
MAR 6 1981

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DDC FILE COPY

81 3 06 077

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Volumes IV and V of this report will be published at a later date.

RADC-TR-80-379, Vol I (of five) has been reviewed and is approved for publication.

APPROVED: Clement D Falzarano

CLEMENT D. FALZARANO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

⑲ REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER ⑱ RADC-TR-80-379, Vol 1 (of five)	2. GOVT ACCESSION NO. AD-A096 042	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ⑥ LOGIC PROGRAMMING IN LISP.		5. TYPE OF REPORT & PERIOD COVERED ⑨ Final Technical Report, 1 Oct 77 — 30 Sep 80
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) ⑩ J. A. Robinson E. E. Sibert		8. CONTRACT OR GRANT NUMBER(s) ⑮ F30602-77-C-0235
9. PERFORMING ORGANIZATION NAME AND ADDRESS Syracuse University School of Computer & Information Science Syracuse NY 13210		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F ⑰ 19 ⑯ 5581 1903
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE ⑪ January 1981
		13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same ⑫ 144		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Clement D. Falzarano (ISIS)

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

Programming Systems; Programming Languages; Programming Grammars; Proving Programs Correct; Computer Modeling; System Simulation; Scheduling Algorithm; Logic Programming.

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

The "Language Studies" contract is divided into four project areas, all of which are directed to the problems of effectively, reliably and efficiently using modern computers in a wide range of applications.

Three of the projects deal with methods of communicating with computers.

Task 1. Very High Level Programming Systems (P.I: J. A. Robinson). This group is working towards combining the features developed to support

(Cont'd)

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410196

Item 20 (Cont'd)

cont

work in the area of artificial intelligence and those used in general program development into a new conceptual framework that can be understood and used by a large community of users.

Task 2. Proving Program Correctness (P.I: J.C. Reynolds). This group is working towards programming language designs which increase the probability that specification errors will be detected by the compiler or interpreter and to provide the language facilities so that users will more nearly be able to prove that programs perform as they are specified than is currently possible.

Task 3. Grammars of Programming (P.I: E. F. Storm). This group is working towards the development of methods which will allow users to communicate with computer programs in terms more normal to their every day communication forms.

Task 4. Systems Studies (P.I: R.G. Sargent). This group is working towards developing more sophisticated and efficient models of computer systems which can predict system performance when given particular parameter values. The current efforts concern models of transaction processing systems (TPS).

Accession For
NTIS
DTIC
Unannou
Just
By
Dist
Dist

Preface

This report describes efforts completed in the Language Studies project at Syracuse University under RADC contract F30602-77-C-0235. The work covers the period October 1, 1977 through September 30, 1980.

The report is produced in five volumes to facilitate single volume distribution.

Volume 1. Report from the Very High Level Programming Systems task. Report title is "Logic Programming in Lisp".

Volume 2. Report from the Systems Studies task. Report title is "Multiple Finite Queueing Model with Fixed Priority Scheduling".

Volume 3. Report from the Systems Studies task. Report title is "An Algorithmic Solution for a Queueing Model of a Computer System with Interactive and Batch Jobs.

Volume 4. Report from the Grammars of Programming task. Report title is "Programming Control Structures in a High Level Language.

Volume 5. Report from the Proving Program Correctness task. Report title is "Realignment".

Acknowledgement

The authors wish to thank their colleagues Kenneth Bowen and Lockwood Morris for the stimulus of their intellectual friendship and for the numerous ideas and suggestions which they have contributed to the development of the system described in this manual.

CONTENTS

EVALUATION

This effort, being responsive to RADC TPO-R5A, Software Cost Reduction, was undertaken to perform research in specific areas of computing technology for affecting more reliable software, improved user communication with computer systems, and improved techniques for system performance evaluations. The applicable technology areas included: very high level programming system, grammar of programming, proving program correctness, and system performance evaluation techniques. Concepts have been advanced in these areas by this effort. Research in the very high level programming system area produced an experimental system combining logic programming with LISP for implementing intelligent data base query systems. Continued developments will allow for enhancements to be made to the system to provide a more powerful, user-friendly, larger-scale data base query capability than is possible with the current system. The system will provide a simpler, more natural, and more powerful formalism than is currently available.

Research in the system performance area produced solution procedures for finding values of system performance measures. These system performance measures consider several variables that have been combined into unique sets of numerical algorithms. These algorithms were verified using simulation techniques. The techniques developed in this area can be utilized to assist decision makers in choosing optimal system scheduling rules in specializing environments.

Clement D. Falzarano
CLEMENT D. FALZARANO
Project Engineer

CHAPTER 0

INTRODUCTION

Since Kowalski's 1974 paper "Predicate Logic as Programming Language" [Kowalski 1974] there has been a growing interest in the use of what he calls "logic programming" as a technique for specifying computations.

This technique consists of formulating computational specifications as a set of declarative sentences, each of which is a simple assertion of some truth - conditional or unconditional, general or particular - which one wishes to record in a "knowledge base".

A conditional assertion has the form

B if A

in which B is the <u>conclusion</u> and A is the <u>hypothesis</u>. The hypothesis is a list

(A1 ... An)

of <u>conditions</u> Ai all of which (the assertion is saying) must be true <u>in order</u> that the conclusion be true. As a special case, the list A may be empty. Such a hypothesis is always true, and so the assertion in this case is said to be unconditional.

Kowalski writes the conditional assertion

B if (A1 ... An)

as

B <- A1 ... An

and the unconditional assertion

B if ()

as

B <-

Each of the A's and the B is a sentence in subject-predicate form, or "predication"

(P S1 ... Sk)

in which some predicate P is ascribed to a subject (S1 ... Sk) which in general is a tuple of descriptive expressions each of which is either a proper name, or a variable, or an applicative construction

(F S1 ... Sn)

in which some operator F is applied to some operand (S1 ... Sn). The operand of a construction is in general a tuple of descriptive expressions of just the same kind as the subject of a predication.

An assertion containing one or more variables is general. It is also often called a rule.

An assertion containing no variable is particular. It is also often called a datum.

The variables in a general assertion are treated as if they were governed by universal quantifiers preceding the assertion. Thus, the assertion

(Odd (Product x y)) <- (Odd x) (Even (Sum x y))

should be understood as being preceded by "for all x and y". Once a knowledge base has been built the logic programmer can request answers to queries. It is these requests which invoke the "logic computations" or deductions which reveal the implicit content of the knowledge base.

A query is essentially a description

all (x1 ... xk) such that (A1 and ... and An)

of a set of tuples which satisfy a given conjunction (the constraint of the query).

The constraint of a query may contain variables in addition to those occurring in the answer template (x1 ... xk) of the query.

These are to be understood as being governed by existential quantifiers preceding the constraint.

The answer to such a query is then the set of all tuples whose satisfaction of the given constraint follows logically from the knowledge base.

Thus the answer may be the empty set, or a set containing just one tuple, or a set containing many - even infinitely many - tuples. If the answer set is infinite, then in practice some finite subset of it will be supplied, or some other description of the set will be given.

A logic computation, then, consists of the sequence of events necessary to construct the answer to some query from the information embodied in some knowledge base.

1.0 PROLOG

These ideas were incorporated into a programming language called PROLOG, designed and first implemented by a group at the University of Marseille. PROLOG has since been implemented at the Universities of Edinburgh, Leuven, London, Waterloo and Budapest.

PROLOG implementations of logic programming go beyond the "pure" version of it described by Kowalski. They provide certain "imperative" features by which the programmer can affect the deductive computation of the answer to a query, and indeed by which he can affect the meaning of the query and of the assertions in the knowledge base.

These "control constructs" of PROLOG have been found most useful in practical applications of logic programming and we are in no sense critical of them. However, we believe that it is one of the essential ideas of logic programming to make a clean distinction between the "logic" of one's program and its "control".

2.0 LOGIC

Accordingly we have implemented a programming language called LOGIC, which embodies our idea of the "pure" version of logic programming featured in Kowalski's writings. Those who are interested to experiment with "pure" logic programming can do so by working with LOGIC.

For those who may wish to avail themselves - while still in some sense working within a logic programming framework - of a greater degree of algorithmic control over events, we have embedded LOGIC within a system called LOGLISP.

3.0 LOGLISP = LOGIC + LISP

LOGLISP is a marriage of LOGIC with LISP.

A LOGLISP workspace contains everything one expects to find in a LISP workspace, and can be used purely as such by those who wish to ignore the presence of LOGIC in that workspace.

The same LOGLISP workspace can also be used as a "pure" LOGIC workspace, that is, as nothing but a basic logic programming environment, in which the assertion/query style of computing can be conducted in just the Kowalski manner. The logic programming facilities are invoked by making suitably-formed LISP calls on such LISP functions as ¦- ("assert") and the query functions ALL, ANY, THE, and SETOF. These LISP functions, together with further auxiliary and supplementary LISP functions, comprise the LOGIC system.

A major advantage of embodying logic programming within LISP in this way is that the LISP environment is available to the logic programmer as a convenient host facility in which LISP functions for editing, displaying, monitoring, debugging, inputting and outputting one's assertions, queries and deductions can be invoked interactively or under program control.

Since the putting of a query is just the submission of an appropriate LISP function call, this can be done either (as in the PROLOG systems) interactively from the terminal or internally from within an applications program.

Since the answer to a query is a LISP data object it can either (as in PROLOG) be displayed on the terminal as a stream or returned to an internal call as its result and subjected, if desired, to analysis and manipulation.

Both predicates and operators in logic expressions can be given a LISP meaning by suitable programmer-supplied definitions of them as LISP function names. Some proper names indeed have a LISP meaning which is present in every workspace as part of LISP itself.

By a benign extension of the "pure" logic programming paradigm, LOGLISP is capable of recognizing such predicates and operators during the deduction cycle of LOGIC. The predications and constructions in whose heads they occur are thereby treated as

LISP-meaningful function calls, and are replaced in situ by appropriate simplifications.

The effect of this LISP-simplification step, performed once in every iteration of LOGIC's deduction cycle, is to give the LOGIC programmer the means to invoke very nearly the full power of LISP from within logic expressions.

This fact, together with the previously mentioned fact that LOGIC calls are simply certain LISP calls, means that it is very easy to initiate subordinate deductions during a deduction, by making recursive calls on LOGIC from within LOGIC.

Thus LISP is not only a rich and convenient host environment for LOGIC programming, but also an intimately involved partner in the novel hybrid style of "LOGLISP" programming in which LISP and LOGIC call each other, and themselves, recursively.

The following chapters describe LOGLISP in full. The background ideas are explained in detail, and the design and implementation are presented both "top-down" and "bottom-up" . Examples of applications of LOGLISP are given which illustrate its novel capabilities.

LOGLISP runs on the DEC-10 under the TOPS-10 operating system using UCI LISP.

CHAPTER 1

EXPRESSIONS. NOTIONS AND NOTATIONS.

In this manual we are concerned with computations whose data are expressions. It will be useful to have the basic ideas and notational conventions available from the outset, and in this chapter we discuss the most important of these. The general framework is that of LISP , augmented in certain ways to accommodate the needs of LOGIC.

1.1 EXPRESSIONS.

LISP has two kinds of expression: atoms and dotted pairs. We further divide the atoms into two kinds: variables and proper names. Therefore we have three kinds of expression:

variables
proper names
dotted pairs

A variable is an atom which begins with a lower case letter. A proper name is any atom which is not a variable (in particular a numeral is a proper name). A dotted pair is a composite expression with two immediate constituents, called its head and its tail, both of which are expressions. We have three formal predicates, for use in writing algorithms, which correspond to the three kinds of expression.

```
(VAR A) = TRUE   if A is a variable, = FALSE otherwise
(NAME A) = TRUE  if A is a proper name, = FALSE otherwise
(CONSP A) = TRUE is A is a dotted pair, = FALSE otherwise
```

1.2 NOTATION FOR DOTTED PAIRS AND LISTS.

When A is a dotted pair whose head is B and whose tail is C, we write

B = hA
C = tA

using the decomposition functions h and t. To indicate the composition of A from B and C we write:

A = (B.C)

using the composition function . ("dot") written between its two arguments. In writing nested compositions with the infixed dot we may omit pairs of parentheses with the understanding that association is to the right. Thus

A.B.C.D.E.F.G

is short for

(A.(B.(C.(D.(E.(F.G))))))

A further notational economy is achieved by identifying certain expressions as lists and writing them without dots. All lists are dotted pairs except for one, which is the proper name: NIL . NIL is known as the empty list, and may also be written: () . Lists other than () are said to be nonempty . A nonempty list is any dotted pair whose tail is a list. A nonempty list may be written by writing its one or more components in order, with a left parenthesis before the first component and a right parenthesis after the last. The head of a list is its first component, and in general the (i + 1)st component of a list is the ith component of its tail . Thus the list

(0.(2.(4.(6.(8.(BINGO.NIL))))))

has six components and would be written

(0 2 4 6 8 BINGO)

Note that the tail of a nonempty list is just the list of its remaining components after the head has been removed. Certain formal notions are used for computing with lists. The result of concatenating two lists L and M is written L*M and is defined by

L*M = if L is () then M else (hL).((tL)*M)

Thus

(1 2 3)*(4 5 6) = (1 2 3 4 5 6)

The length of a list is the number of components it has:

(LENGTH L) = if L is () then 0 else 1 + (LENGTH tL)

1.3 PATHS. STRUCTURES. PRINTABLE EXPRESSIONS

The decomposition functions h and t are the two paths of length 1 . The functions hh, ht, th, tt are the four paths of length 2. In general the 2**(n+1) paths of length n+1 are all the functions hp, tp where p is a path of length n. The identity function is the (only) path of length 0. An expression is said to admit a path p if the result of applying p to it is defined. Thus, every expression admits I, and every dotted pair also admits h and t. Variables and proper names admit only I, and this fact is their characteristic structural property. In general the set of all paths admitted by an expression A is called the structure of A, and gives a rather direct portrayal of A's "shape". The printable expressions are those whose structure is finite. Not all expressions are printable. For example, the dotted pair whose head is 0 and whose tail is itself is not printable. Its structure is the infinite set of paths

{ I, t, tt, ttt, ..., ht, htt, httt, ..., }

It may be described as the expression which solves the equation

x = 0.x

and we may reason about it from this description. However, to attempt to print it would result in a nonterminating process.

1.4 ENVIRONMENTS

A dotted pair whose head is a variable is called a binding. A list whose components are bindings with distinct heads is called an environment. Intuitively an environment is a collection of replacement instructions coded as dotted pairs, each one saying that a certain variable (its head) is to be replaced by a certain expression (its tail). An environment which contains all the bindings of the environent E (and perhaps other bindings) is called an extension of E.

1.5 THE NOTION DEF

If E is an environment and A is an expression we say that A is defined in E if, and only if, there is a binding in E whose head is A. Accordingly we introduce the function DEF by the scheme

```
(DEF A E) = if E is () then FALSE
                else if hhE is A then TRUE
                else (DEF A tE)
```

which computes the truth value that A is defined in E. Note that if A is defined in E then A is a variable.

1.6 THE NOTIONS IMM AND ULT.

If A is defined in E we say that the immediate associate of A in E is the tail of the binding in E whose head is A, and we define the corresponding function IMM by

```
(IMM A E) = if hhE is A then thE else (IMM A tE)
```

with the understanding that IMM will never be invoked for an A and E such that A is not defined in E. The immediate associate in E of a variable A may itself be a variable defined in E. In such a case we may wish to track down the ultimate associate of A in E - namely the first expression in the series

```
A , (IMM A E), (IMM (IMM A E) E) ...,
```

which is not defined in E. Accordingly we define the function ULT by

```
(ULT A E) = if (DEF A E) then (ULT(IMM A E)E) else A
```

which computes, for any expression A and environment E, the ultimate associate of A in E. For example, if E is the environment

```
( x.y y.z z.(F A (B r s)) r.(G s) s.5 )
```

then the immediate associate of x in E is y, but the ultimate associate of x in E is (F A (B r s)) .

1.7 REALIZING EXPRESSIONS IN ENVIRONMENTS.

Given an expression A and an environment E, we consider the result of replacing each variable in A by its immediate associate in E. This expression is called the realization of A in E. To compute the realisation of A in E we use the function REAL, defined by:

```
(REAL A E) = if (CONSP A) then (REAL hA E).(REAL tA EA)
             else if (DEF A E) then (IMM A E)
             else A
```

We note that, for example, the realization of (PLUS x y) in the environment

```
( x.y y.z z.(F A (B r s)) r.(G s) s.5 )
```

is (PLUS y z) . We are also interested in recursive realizations. For example, if we start with (PLUS x y) we obtain each of the following expressions by repeatedly realizing the previous one in E:

```
(PLUS y z)
(PLUS z (F A (B r s)))
(PLUS (F A (B r s)) (F A (B (G s) 5)))
(PLUS (F A (B (G s) 5)) (F A (B (G 5) 5)))
(PLUS (F A (B (G 5) 5)) (F A (B (G 5) 5)))
```

Realizing the final expression in E merely reproduces it. This final expression is therefore by definition the recursive realization of (PLUS x y) in E. In general the recursive realization of an expression A in an environment E is defined by:

```
(RECREAL A E) = if (CONSP A) then (RECREAL hA E).(RECREAL tA E)
                else if (DEF A E) then (RECREAL (ULT A E) E)
                else A
```

1.8 UNPRINTABLE RECURSIVE REALIZATIONS OF PRINTABLE EXPRESSIONS.

It can happen that a printable expression may have an unprintable recursive realization in a printable environment. For example, in the environment

```
(  x.(0.x) )
```

the expression x has the recursive realization

```
(0.(0.(0.      ...      )))
```

which is the "infinite expression" whose head is 0 and whose tail is itself.

1.9 UNIFICATION

A fundamental notion in logic programming is the operation of unifying two expressions A and B relative to a given environment E. This operation yields a result, denoted by (UNIFY A B E) , which is either the message "IMPOSSIBLE" , indicating that A and B cannot be unified with respect to E, or else is an extension of E in which the recursive realizations of A and B are identical. In the latter case we say that the environment (UNIFY A B E) is the most general unifier ("mgu") of A and B with respect to E. By definition, we then have that

```
(RECREAL A (UNIFY A B E)) = (RECREAL B (UNIFY A B E))
```

The computation of (UNIFY A B E) is defined by

```
(UNIFY A B E) =

  if E is "IMPOSSIBLE"  then  "IMPOSSIBLE"
  else (EQUATE (ULT A E) (ULT B E) E)
```

where

```
(EQUATE A B E) =

  if A is B then E
  else if (VAR A) then  (A.B).E
  else if (VAR B) then  (B.A).E
  else if not (CONSP A) then "IMPOSSIBLE"
  else if not (CONSP B) then "IMPOSSIBLE"
  else  (UNIFY tA tB (UNIFY hA hB E))
```

The mgu of (P (G x y) x y) and (P a (H b) c) with respect to the empty environment () is

```
(  y.c x.(H b) a.(G x y) )
```

and in this environment both expressions are recursively realized as

```
(P (G (H b) c) (H b) c)
```

The mgu of A and B with respect to E is intuitively the most general way that E can be extended to an environment in which A and B can be recursively realized as identical expressions. It is possible that unifying A and B will make them unprintable. For example, the most general unifier of the expressions x and (0.x) with respect to the empty environment () is the environment (x.(0.x)) in which x is bound to (0.x) . This shows that in general it is possible for (UNIFY A B E) to be an environment in which the recursive realizations of A and B are identical but unprintable.

1.10 SUBSTITUTIONS

Some readers may be more familiar with the usual treatment of unification, which is developed in terms of the idea of substitutions. A substitution is a mapping from expressions to expressions which preserves proper names and the dotted pair structure. More precisely, a mapping S from expressions to expressions is a substitution if, and only if, it satisfies the two conditions:

$$\begin{aligned} XS &= X && \text{for all proper names } X \text{ ,} \\ (X.Y)S &= (XS).(YS) && \text{for all expressions } X \text{ and } Y \text{ .} \end{aligned}$$

We denote the result of applying a substitution S to an expression X by the postfix notation: XS , as illustrated above. An important property of a substitution is that its effect upon any expression is completely determined by its effect on the variables (if any) which it actually changes. By listing those variables, each equated to its image under the substitution, we therefore give a complete description of the substitution. But the information in such a list of equations is just what is provided by an environment. The list of equations

$$V1 = A1, \ldots, Vn = An$$

corresponds to the environment

$$(\ V1.A1 \quad \ldots \quad Vn.An\ )$$

and conversely. Indeed if S corresponds in this way to the environment E, then the image XS of any expression X under S is just the expression (REAL X E). We write [E] for the substitution corresponding in this way to the environment E. Thus we have

$$X[E] = (\text{REAL } X \ E)$$

for all expressions X and environments E. In this correspondence between environments and substitutions, the empty environment corresponds to the identity substitution (which transforms every expression into itself). Composition of two substitutions S and T yields the substitution ST, which sends each expression X into the expression X(ST) = (XS)T obtained by first applying S to X and then T to the result. If S is [E] and T is [F] , ST is [L], where L is the list of all distinct bindings

$$V.(V[E][F])$$

where V is defined in E or in F (or both).

An environment E may be taken as a description not only of [E] but also of the iterate of [E] . The iterate $S^{\sim}$ of a substitution S is the "limit" of the series

$$S, SS, SSS, \quad \ldots,$$

To find the image $X(S^{\sim})$ of an expression X under the iterate of S, we repeatedly apply S to X until no further changes occur. That is, $X(S^{\sim})$ is the first expression in the series

$$X,\ XS,\ XSS,\ XSSS,\ \ldots,$$

which is the same as its predecessor. It turns out that if S is E then $X(S^{\sim})$ is (RECREAL X E). If S is [E] then $S^{\sim}$ is denoted by {E}. So we have

$$X[E] = (\mathrm{REAL}\ X\ E)$$
$$X\{E\} = (\mathrm{RECREAL}\ X\ E)$$

Now in terms of substitution mappings, a unifier of two expressions A and B is a substitution S which maps A and B onto the same expression:

$$AS = BS$$

and a most general unifier of A and B is a unifier U of A and B with the property that

$$S = US$$

for all unifiers S of A and B.

Thus if U is an mgu of A and B and S is any unifier of A and B we have

$$AS = AUS = BUS = BS \quad \text{and} \quad AU = BU$$

so that the common expression onto which S maps A and B is obtainable by applying S to the common expression onto which U maps A and B. The substitution {(UNIFY A B E)} is the mgu of the two expressions A{E} and B{E} . Thus UNIFY is given the two expressions to be unified in an indirect way.

1.11 IMPLICIT EXPRESSIONS

The way that the two expressions A{E} and B{E} are given to the UNIFY algorithm is indirect, in "unassembled" form. This idea of working with expressions not yet (or possibly never) fully assembled is used extensively in our system. It makes for computational economy and also for increased intelligibility. We think of the list (A E) as an "implicit" way of giving the expression A{E}. We say that A is the skeleton part, and E the environment part, of the implicit expression (A E). For many purposes it is more convenient, as well as more economical, to deal with such "implicit expressions" than with the actual expressions themselves. This is particularly the case when (A E) describes an unprintable expression even though both A and E are printable - as in the example previously mentioned when A is x and E is (x.(0.x)).

1.12 INSTANCES. VARIANTS. GROUND EXPRESSIONS. PATTERNS.

We often wish to consider, for some expression A, the various expressions AS, where S is a substitution. These are known as the instances of A. For example, the expressions

```
(Divides 17 85)
(Divides (Plus a b) (Times 3 c))
```

are both instances of the expression (Divides p q) . The first of them is in fact a ground instance, since it contains no variables. In general we say that expressions which contain no variables are ground expressions: so a ground instance of A is an instance of A which happens to be a ground expression. Expressions which contain one or more variables are known as patterns. We often think of a pattern as a way of representing all of its instances.

VARIANTS

In the role of a representative of all its instances a pattern is not unique. Other patterns - known as its variants - have exactly the same instances. For example, the expressions

```
(Divides p q)    (Divides x y)
```

have exactly the same instances. Each is a variant of the other. In general a variant of an expression A is an instance AS of A under a substitution which maps variables onto variables in one-to-one fashion. Such a substitution is called a variation, and is the only kind of substitution which has an inverse. If [E] is a variation then its inverse is [E'], where E' is obtained from E by interchanging the head and tail of each of its bindings. The compositions [E][E'] and [E'][E] are then both the identity substitution.

In view of the identity of the set of instances of an expression with that of any variant of the expression, we often treat mutual variants as merely different ways of writing the same thing. However, in some of the computations involving patterns (such as the unification computation) it is sometimes necessary to take suitable variants of one's data beforehand.

To see why this is so, consider the problem of finding a pattern whose instances are exactly those which are instances of two given expressions, A and B.

For example, if A and B are the expressions

```
      (Divides (Plus x y) z)       (Divides x (Times x y))
```

then among their common instances are the expressions

```
      (Divides (Plus 3 4) (Times (Plus 3 4) 6))
      (Divides (Plus 0 0) (Times (Plus 0 0)(Exp x y)))
```

and so on. We can get the first instance from A by the substitution

```
         x = 3, y = 4, z = (Times (Plus 3 4) 6)
```

We can get it from B by the substitution

```
         x = (Plus 3 4), y = 6 .
```

However, there is no single substitution S such that AS = BS =this common instance. The difficulty is the occurrence of the same variables in both A and B. If we take a variant of B which has no variables in common with those of A - say, the expression

```
       (Divides p (Times p q))
```

which we shall call C - then we can in fact find a pattern whose instances are exactly those common to A and B. To do this we need only compute the expression

```
       (REALREC A (UNIFY A C ()))
```

or (which is the same)

```
       (REALREC C (UNIFY A C ()))
```

which is the "most general common instance" of A and C - and therefore also of A and B.

Now the environment (UNIFY A C ()) is

```
       ( p.(Plus x y) z.(Times p q) )
```

and so the required expression is

```
(Divides (Plus x y) (Times (Plus x y) q) )
```

Every expression which is an instance both of A and of B is an instance of this expression - and conversely. This example illustrates the way in which the unification computation solves the general problem of constructing a pattern whose instances are precisely those which two given patterns have in common. Of course, when the two given patterns have no common instances, no such pattern exists. The UNIFY function detects all such cases by returning "IMPOSSIBLE" instead of an environment.

CHAPTER 2

LOGIC PROGRAMMING IN GENERAL

Logic programming is a technique for specifying computations by making assertions. No imperative constructs are used. The course of events during a logic computation is determined not by the programmer's control instructions (for there are none) but by the machine's pursuit of certain of the deductive consequences of the programmer's assertions. For example, the programmer might make the following assertions:

1 Drobny is a champion
2 Drobny is older than Rosewall
3 Rosewall is older than Goolagong
4 If x is older than y and y is older than z
then x is older than z
5 If x was born before y then x is older than y
6 Kelly is a child of Goolagong
7 If x is a child of y then y was born before x
8 Goolagong is female
9 Drobny is male
10 Rosewall is male
11 Rosewall is a champion
12 Goolagong is a champion
13 Connors is a champion
14 Borg is a champion
15 Connors is male
16 Borg is male
17 Borg was born before Connors
18 Connors was born before Kelly
19 Kelly is female
20 Evert is a champion
21 Evert is female
22 Evert was born before Connors

FIGURE 1

Some of these assertions are of particular facts; others are generalities involving the use of logical variables x, y, z. LOGIC is now capable of responding to queries about the "world" described by these assertions. In supplying answers to such queries it must in general deduce them from what it has been told (rather than merely look the answers up) . For example, the query:

> Which male champions are older
> than Kelly ?

would elicit the answer

> (Connors Borg Rosewall Drobny)

That these persons are male and champions is explicitly given among the assertions, but that each of them is older than Kelly must be deduced. The deductions involved can, if desired, be examined by the user. For example, one could request:

> * Explain the fourth answer

and LOGIC would respond with the following rationale:

```
*   To show: Drobny is a male
             Drobny is a champion
             Drobny is older than Kelly

*   it is enough, by assertion 9,

*   to show: Drobny is a champion
             Drobny is older than Kelly

*   But then it is enough, by assertion 1,

*   to show: Drobny is older than Kelly.

*   But then it is enough, by assertion 4,

*   to show: (there is a y:1 such that)
             Drobny is older than y:1
             y:1 is older than Kelly

*   But then it is enough, by assertion 2,

*   to show: Rosewall is older than Kelly.

*   But then it is enough, by assertion 4,

*   to show: (there is a y:2 such that)
             Rosewall is older than y:2
             y:2 is older than Kelly.

*   But then it is enough, by assertion 3,

*   to show: Goolagong is older than Kelly.

*   But then it is enough, by assertion 5,

*   to show: Goolagong was born before  Kelly.

*   But then it is enough, by assertion 7,

*   to show: Kelly is a child of Goolagong.

*   But then it is enough, by assertion 6
    to show:  nothing.

*   End of explanation.
```

FIGURE 2

In the LOGIC system implemented within LOGLISP, the language of the queries, assertions and explanations is formalized and artificial. We shall shortly discuss the details of its design. Meanwhile, note that an explanation is essentially a proof, which proceeds in steps all of the same kind. At each step there is a "constraint list" of simple propositions, all to be shown true. Any variables in these propositions are considered to be existentially quantified by quantifiers placed at the beginning of the constraint list, and the constraint list itself is considered to be the conjunction of its members. The empty constraint list (i.e. the empty conjunction) is by convention true, so that if at some step the list has become empty, the proof is complete - there is nothing left to show. In general, each inference step consists of three stages:

(1) The selection of a constraint A from the constraint list and of an assertion from the knowledge base whose conclusion B will unify with A.

(2) The replacement of A in the constraint list by the constraints comprising the hypothesis (if any) of the selected assertion.

(3) The application to the new constraint list of the most general unifier of A and B.

The notion of unification has been defined only for formal expressions, however, and so to make this account precise we must now recast it in terms of the formal language of LOGIC.

Let us now survey this formal language.

2.1 PREDICATIONS

The basic unit of the formal language is the predication. Predications are simple sentences of the subject-predicate form in which the predicate is written first and the subject second. The predicate may be any proper name P which is not a numeral. The subject is a list of expressions called terms. Ground terms are essentially noun-phrases which denote things . A list A = (A1 ... An) of n ground terms denotes the n-tuple of things denoted respectively by the component terms A1, ..., An . Predicates denote properties of tuples. (Properties of tuples are often also called relations). The intuitive meaning of a ground predication with predicate P and subject A is the proposition that the tuple denoted by A has the property denoted by P. We write this formally as the list whose head is P and whose tail is A .

Thus we might formally write:

Drobny is a champion	as	(Champion Drobny)
Drobny is male	as	(Male Drobny)
Drobny is older than Kelly	as	(Older Drobny Kelly)
Evert is female	as	(Female Evert)
Evert was born before Kelly	as	(Before Evert Kelly)
Kelly is a child of Goolagong	as	(Child Kelly Goolagong)

2.2 TERMS

A term may be either a variable, or a proper name, or a construction. Constructions have an operator-operand form. The operator (which may be any proper name which is not a numeral) denotes an operation, and the operand may be any list of terms . When the construction is a ground expression, its operand denotes a tuple of things, in just the same way as does the subject of a ground predication. Constructions are indeed syntactically indistinguishable from predications. Their common syntactic form reflects an underlying unity in their semantics as applicative expressions. Each ground construction or ground predication can be understood as representing the result of applying some function to some argument. In the case of a predication this means construing a property or relation as a truth function, namely a function which yields as its result one or other of the two truth values, TRUE, FALSE. We write the construction with operator F and operand A = (A1 ... An) as the list

(F A1 ... An)

whose head is F and whose tail is A.

Ground predications , then, express facts and denote truth values. Ground terms express applicative descriptions and denote things. Both ground terms and ground predications have the same simple, systematic denotational semantics based on the applicative principle.

2.3 WORLDS

A world is a collection of facts - "everything that is the case" in that world. In logic programming a world is represented by a collection of ground predications. Given a collection W of ground predications as such a world, we can ask for what substitutions, any, a given predication Q (whether ground or not) is "true in W" . If Q is a ground predication, this is simply the question whether Q is a member of W. If Q is in W, the answer is then: the identity substitution. If Q is a predication pattern, however, this is not quite so simple a question, and we construe it to mean: for which substitution operations E is the the instance of Q under E in W? For example, the world specified by the assertions in our earlier example is the set

(Male Drobny)
(Male Rosewall)
(Male Borg)
(Male Connors)

(Female Goolagong)
(Female Evert)
(Female Kelly)

(Champion Drobny)
(Champion Rosewall)
(Champion Borg)
(Champion Connors)
(Champion Goolagong)
(Champion Evert)

(Older Drobny Rosewall)
(Older Drobny Goolagong)
(Older Drobny Kelly)
(Older Rosewall Goolagong)
(Older Rosewall Kelly)
(Older Goolagong Kelly)
(Older Borg Connors)
(Older Borg Kelly)
(Older Evert Connors)
(Older Evert Kelly)
(Older Connors Kelly)

(Before Borg Connors)
(Before Connors Kelly)
(Before Evert Connors)
(Before Goolagong Kelly)

(Child Kelly Goolagong)

FIGURE 3

With this world as W, if we ask what are the substitutions for which the predication

(Male x)

is true in W, we get four "solutions", namely:

x = Drobny
x = Rosewall
x = Borg
x = Connors

there being four ground instances of "(Male x)" in W , namely those corresponding to these four substitutions. More generally we can ask a question involving a conjunction of predications. If Q1, ..., Qn are predications, we can ask of a world W

for what substitutions
is (Q1 & ... & Qn) true in W ?

or more briefly:

```
what substitutions satisfy (Q1 & ... &Qn) in W?
```

For example in the W of our example the question

```
what substitutions satisfy
 ( (Male x) & (Champion x) & (Older x Rosewall) )
in W?
```

has the answer

```
x = Drobny
```

since under this (but no other) substitution the conjunction becomes true in W.

2.4 QUERIES

It is useful to introduce the formal notion of a query, based on the preceding discussion. A query is an expression of the form

```
(ALL X Q1 ... Qn)
```

in which Q1 ... Qn are predications and X is an expression called the answer template of the query. The answer template may be any variable, any proper name, or any list of terms . The list Q = (Q1 ... Qn) is the constraint list of the query. For any world W, such a query has an answer, which is a list of expressions. Each expression in this answer list is the instance of the answer template under a substitution which satisfies the constraint list Q, that is, which transforms the conjunction (Q1 & ... & Qn) into one which is true in W. Thus the query

```
(ALL x (Male x)
       (Champion x)
       (Older x Rosewall) )
```

has the answer (in the world of our example)

```
(Drobny)
```

since the substitution x = Drobny is the only one which satisfies the given constraint, while the query

```
(ALL z (Female z)(Older z Drobny))
```

has the empty list

```
()
```

as its answer since there are no substitutions which satisfy the constraint

```
( (Female z) (Older z Drobny) )
```

2.5 SPECIFYING A WORLD BY ASSERTIONS

It is not expected that one should have to specify a world by explicitly listing, as in FIGURE 3, all of its predications (although this would in principle be possible for a finite world). A world is specified indirectly, by giving a collection of assertions. An assertion has two parts: a conclusion, which is a predication, and a hypothesis, which is a list of predications. The hypothesis of an assertion can be the empty list, in which case the assertion is said to be an unconditional assertion, whereas an assertion whose hypothesis is nonempty is said to be a conditional assertion. An unconditional assertion whose conclusion is B is written

```
B <-
```

while a conditional assertion with conclusion B and hypothesis (A1 ... An) is written

```
B <- A1 ... An
```

A collection of assertions is called a knowledge base . Any such collection determines a world.

An unconditional ground assertion B <- intuitively says that B is one of the facts in the world being described - "B is true" . A conditional ground assertion B <- A1 ... An says that B is one of the facts in the world being described provided that A1, ..., An all are - "if A1 and ... and An are true then B is true" . An assertion pattern - an assertion containing one or more variables - has the same descriptive effect as would the set of all its ground instances. In general this means that an assertion pattern is in effect a universally quantified statement. If its variables are x1, ..., xk (say) then the assertion B <- A1 ... An can be read

"for all x1, ..., xk: if A1 and ... and An then B"

Indeed, if some of the variables among the xi (say, z1, ..., zp) do not occur in the conclusion A while the rest (say, y1, ..., yt) do, the assertion B <- A1 ... An may be more intuitively (but equivalently) read

"for all y1, ..., yt:
    if there exist z1, ..., zp such that A1 and ... and An
    then B"

In the example of FIGURE 1 there are three such assertion patterns. All the other assertions in FIGURE 1 are unconditional ground assertions. FIGURE 4 shows the knowledge base of FIGURE 1 written in the formal notation.

```
 1   (Champion Drobny) <-
 2   (Older Drobny Rosewall) <-
 3   (Older Rosewall Goolagong) <-
 4   (Older x z) <- (Older x y)(Older y z)
 5   (Older x y) <- (Before x y)
 6   (Child Kelly Goolagong) <-
 7   (Before y x) <- (Child x y)
 8   (Female Goolagong) <-
 9   (Male Drobny) <-
10   (Male Rosewall) <-
11   (Champion Rosewall) <-
12   (Champion Goolagong) <-
13   (Champion Connors) <-
14   (Champion Borg) <-
15   (Male Connors) <-
16   (Male Borg) <-
17   (Before Borg Connors) <-
18   (Before Connors Kelly) <-
19   (Female Kelly) <-
20   (Champion Evert) <-
21   (Female Evert) <-
22   (Before Evert Connors) <-
```

FIGURE 4

The knowledge base of FIGURE 4 completely determines the world of FIGURE 3, according to the following general definition.

DEFINITION

The world determined by a knowledge base D is the smallest set W of ground predications which satisfies the two conditions:

(1) if D contains the unconditional ground assertion G <- , then G is in W

(2) if G is a ground instance of an assertion in D and the predications in the hypothesis of G are all in W, then the conclusion of G is in W.

END OF DEFINITION

In effect, this definition describes a process which infers W from D by a series of wholesale inference steps. First, by (1), the process constructs outright the set W0, which contains just those ground predications which are conclusions of unconditional assertions in D. Then by (2), in general, having constructed the set Wn, this process constructs Wn+1 by adding to Wn the *conclusion of every ground instance G of every conditional* assertion in D, provided that every predication in the hypothesis of G is in Wn. Thus the process constructs a series of bigger and bigger worlds

W0, W1, ..., Wn, ...

which either ends (with a world that is the same as its predecessor) or else continues indefinitely. The world W is then the "limit" of this series, i.e., the union of all of the sets in it , i.e. the smallest set which includes them all. Thus the world W is determined by a knowledge base D through a "bottom up" process of reasoning.

Given such a D, we wish to be able to answer queries about its world W. In doing so we wish to avoid the brute force method of generating W bottom up and searching it. It is much better, given a query about W, to reason "top down" about W's contents without actually constructing W. This turns out to be possible through the use of unification, built into a special inference principle called *LUSH* resolution. *This inference principle can* be applied very efficiently through the use of implicit expressions, as we shall now see.

2.6 IMPLICIT CONSTRAINTS AND THEIR SOLUTIONS

By an implicit constraint we mean a pair (Q E) in which E is an environment and Q is a list of predications. The expression Q{E} is the corresponding explicit constraint . Now let D be a knowledge base and let W be the world described by D. We denote by (SOL Q E D) the set of extensions A of E for which the predications in Q{A} are all true in W . We wish to calculate (SOL Q E D) from (Q E) and D . There are two cases to consider. The first case is when Q is empty. Then (SOL Q E D) is simply the set whose only member is E . Such a (Q E) is said to be solved.

The second case is when (Q E) is unsolved, i.e., when Q is nonempty. For this case we use LUSH resolution to represent the desired set as the union of one or more simpler sets.

2.7 LUSH RESOLUTION

For any unsolved constraint (Q E), any knowledge base D, and any positive integer K not greater than the length of Q, the set

(RES Q E D K)

is a set (possibly empty) of implicit constraints called the (D K)-resolvents of (Q E). The interest of this set lies in the fact that we have:

(SOL Q E D) = (SOL Q1 E1 D) U ... U (SOL Qn En D)

where (Q1 E1), ..., (Qn En) are the (D K)-resolvents (if any) of (Q E). This equation holds for all the admissible values of K (however, the (D K)-resolvents will in general be different, for each value). In particular for some choices of K it may be that there are no (D K)-resolvents of (Q E). This then means that (SOL Q E D) is the empty set, although other choices of K may delay the discovery of this by providing one or more (D K)-resolvents of (Q E).

2.8 THE CHOICE OF K.

The computation of the set (RES Q E D K) involves a choice of the number K. Accordingly we introduce a choice function SEL. For each unsolved implicit query (Q E) and knowledge base D the number (SEL Q E D) is a positive integer no larger than the length of Q. (In the LOGIC system as currently implemented in LOGLISP, we take (SEL Q E D) = 1 throughout).

In general SEL might be expected to take into account the evidence available in Q, E and D so as to make an informed choice with desirable pragmatic effects on the overall computation.

2.9 SPLITTING NONEMPTY LISTS

The purpose of the number K is to determine a decomposition of the list Q of the form: Q = L*(A)*R , where A is the Kth component of Q.

In general we say that, for any list X and any positive integer K no larger than (LENGTH X), the K-decomposition of X is the triple (L A R) such that A is the Kth component of X and X =L*(A)*R.

Thus when K = 1 we have L = (), A = hX, R = tX.

2.10 SEPARATION OF VARIABLES.

The computation of (RES Q E D K) involves a further choice, namely of a variant D' of the knowledge base D. D' must have the property that none of its assertions contains a variable which occurs in (Q E) . This "standardizing apart" of the variables in the constraint from those in the assertions is necessary for the theoretical completeness of the resolution transformation. In the current implementation D' is selected automatically and represented implicitly and economically by techniques explained in Chapter 13.

2.11 DEFINITION OF (RES Q E D K).

> The set (RES Q E D K) is the set of all implicit constraints of the form
>
> (L*B*R (UNIFY A H E))
>
> for which H <- B is an assertion in D' whose conclusion H unifies with A in E, and where (L A R) is the K-decomposition of Q.

2.12 THE DEDUCTION CYCLE

The heart of the LOGIC system is the basic deduction cycle, which computes the set (SOL Q E D) for a given implicit constraint (Q E) and a given knowledge base D.

The computation of (SOL Q E D) consists of the development of two sets of implicit constraints, SOLVED and WAITING. Initially, SOLVED is empty and WAITING contains the single constraint (Q E). These two sets are then subjected to an iterative transformation which corresponds intuitively to the construction of a "deduction tree" whose nodes are implicit constraints. The root of this tree is the implicit constraint (Q E). The successors (if any) of an unsolved node (X Y) are the (D K)-resolvents of (X Y), for some admissible value of K which is selected by the function SEL. The tips of the deduction tree are the solved nodes (if any) and the unsolved nodes (if any) which have no (D K)-resolvents for the particular value of K assigned to them by the function SEL. The output of the deduction cycle is the set of environment parts of the solved nodes of the tree.

As the tree develops, the solved nodes are collected into the set SOLVED, and the nodes which have not yet been processed are kept in the set WAITING. Thus the tree construction is finished when WAITING finally becomes empty.

The deduction cycle is the following three-step algorithm:

```
  IN:  let SOLVED be the empty set and
       let WAITING be the set containing only (Q E)

 RUN:  while  WAITING is nonempty

       do  1  remove some constraint C from WAITING
              and let (X Y) be C

           2  if    (X Y) is solved
              then  add (X Y) to SOLVED
              else  add the (D K)-resolvents of (X Y) to WAITING
                    where  K = (SEL X Y D)

 OUT:  return the set of environment parts of SOLVED
```

In general (SOL Q E D) is computed by executing the deduction cycle and taking its output as the required set.

Several points are worth noting about the deduction cycle.

2.12.1 Failure Nodes: Immediate And Ultimate.

An unsolved node of the deduction tree which has no solved nodes as descendants is known as a "failure". There are two kinds of failure. An immediate failure has no descendants at all - because it has no (D K)-resolvents for the particular value of K selected for it by SEL. An ultimate failure has one or more successors, but they too are failures - the entire subtree rooted in an ultimate failure consists of nothing but failures, and its tips are all immediate failures. It is an interesting problem to design implementations of the deduction cycle in which the subtrees rooted in ultimate failures are kept as small as possible without undue extra computation. Ideally, all failures would be immediate and would be recognised as such in constant (and short) time.

2.12.2 Nondeterminacy Of Deduction Cycle.

There are several sources of nondeterminacy in the RUN step of the deduction cycle.

The most obvious of these are the explicit choices called for in steps 1 and 2. In both cases, the choice can be made uniformly and cheaply according to default criteria which are built into the system design. For example, in our own system the default criterion for the choice of K is to choose always the value K = 1. In the PROLOG systems, the selection in step 1 is in effect ruled by a similar criterion - the first constraint (X Y) is selected from a WAITING which is represented in effect as a list. [We have to say "in effect" because in fact the PROLOG systems handle WAITING dynamically in a backtrack mode of working which never explicitly realises the whole list at once.]

The selection of the node C in step 1 can (as in the PROLOG systems) be made according to the "depth first" criterion in which the younger members of WAITING are chosen before the older members. This may sometimes lead to the "depth first runaway" situation in which one or more nodes in WAITING are never selected because they are never the youngest. In practice other considerations (see the discussion below of the Deduction Window) preclude an infinite depth first runaway, but even the finite versions of it which are allowed by the Deduction Window may be thought undesirable. Avoidance of depth first runaway can be economically achieved by letting the selection in step 1 depend upon a quantity which can be computed once for all for each node when it is first generated. This quantity is the "solution cost" of the node.

2.12.3 Definition Of Solution Cost

The solution cost of a node (X Y) is simply a heuristic estimate of the "cost" (in arbitrary units) of obtaining a solved descendent of (X Y). Ordinarily we estimate this cost as the sum of (LENGTH X) and the depth of (X Y), which is number of nodes preceding (X Y) on its branch of the deduction tree. The user may select any linear combination of these two quantities as the cost estimate (see chapter 8). The selected node C in step 1 is then always one whose solution cost is least. This method coincides with a breadth first development of the tree in the case when the solution cost of a node is taken to be its depth in the deduction tree.

We have also provided a "PROLOG" mode of operation in which the search is strictly depth first. (This mode does not incorporate any other special features of PROLOG.) Details concerning mode selection are also given in chapter 8.

2.12.4 Which Predication To Resolve Away? The Function SEL.

The selection of the value of K in step 2 may well affect the total cost of computing the set of solved descendents of the selected node (X Y) - including the particular case when this set is empty and (X Y) is therefore a failure. However, the potential benefit of lowering this cost is offset by the expense of making the choice. The least costly selection criterion is that used by the PROLOG systems (and by our own system as its default criterion), namely, K = 1. We have not provided the normal user with any means of overriding the default value for K. The present discussion is intended to highlight an opportunity for the system designer to add a further layer of sophistication to the deduction cycle by making the choice of "which predication to resolve away" depend upon particular features of (X Y), rather than making it independent of all such features.

2.13 THE DEDUCTION WINDOW.

Since in general the deduction tree can be infinite, it is necessary in some cases to truncate the deduction cycle and accept the resulting (perhaps incomplete) set of solutions as an approximation to the full set (which may be infinite).

It is desirable to manage this truncation gracefully and to provide the LOGIC user with some control over its details. This is the reason for the deduction window.

The Deduction Window is a collection of parameters which can be set in various ways by the user and which have default values which are used in the absence of user-provided alternatives.

Each parameter in the Deduction Window is used as an upper bound on an associated quantity measuring some feature of the deduction cycle. These quantities are TREESIZE, NODESIZE, ASSERTIONS, RULES and DATA.

At a given moment in the execution of the deduction cycle TREESIZE is the total number of nodes which have so far been generated. The RUN loop is terminated as soon as TREESIZE exceeds the bound set for it in the deduction window.

The implicit constraint (X Y) selected in step 1 of the body of the RUN loop is treated as an immediate failure (hence dropped from WAITING without progeny) if NODESIZE(X Y), ASSERTIONS(X Y), RULES(X Y) and DATA(X Y) are not all within the bounds specified for them in the deduction window.

NODESIZE(X Y) is (LENGTH X), the number of predications in the constraint list X of (X Y).

ASSERTIONS(X Y) is the number of nodes which precede (X Y) on the branch of the deduction tree of which it is the current tip. This number is the same as the number of assertions invoked in its deduction. It is 0 for the initial node, and is 1 greater than that of its predecessor for each derived node.

RULES(X Y) is a quantity similar to ASSERTIONS(X Y), but reflects the classification of assertions into rules and data.

An assertion which contains no variables is a datum - it records a single fact. An assertion containing one or more variables is a rule.

RULES(X Y) is then the number of times a rule was invoked in its deduction, and

DATA(X Y) is the number of times a datum was invoked in its deduction. We obviously have, for each (X Y) in WAITING, that:

DATA(X Y) + RULES(X Y) = ASSERTIONS(X Y) .

Thus the Deduction Window serves as a truncation device which ensures that each particular execution of the deduction cycle will terminate. It provides the user with both a global (TREESIZE) and a local (NODESIZE, ASSERTIONS, RULES and DATA) cutoff control. All the bounds in the deduction window are set to system-defined default values in the absence of user-defined alternatives.

2.14 RECORDING DEDUCTIVE HISTORIES FOR LATER EXPLANATIONS.

The system can be asked to explain the logical genesis of some or all of the members of SOLVED. The deduction cycle so far described does not preserve the information which is needed to provide such explanations. At the option of the user, the deduction cycle can be modified to include provision for keeping a record of the "history" of each solved node. Such a history is essentially the branch of the deduction tree whose tip is that solved node. However, each node in the branch (after the first) must be labelled with the assertion which was invoked in deducing it from its predecessor node. A request to explain a given solved node can then easily be met by constructing from its history a text of the sort illustrated earlier in FIGURE 2.

The extra time and space needed to operate the deduction cycle in the historical mode are not so small as to be negligible. The user therefore will probably decide to switch the deduction cycle into this mode only when the availability of explanations is worth the cost.

CHAPTER 3

LOGIC PROGRAMMING IN LISP

LOGIC is related to LISP in two different ways.

First, it is implemented in LISP - that is, the LOGIC system consists of a collection of LISP functions which live in a LISP workspace and provide all the logic programming facilities described in this manual.

Second, LOGIC in a certain sense contains LISP. This means that the LOGIC programmer can invoke LISP from within LOGIC calls, by incorporating in assertions and queries pieces of text which can be handed over to LISP for processing. To understand how this works we need to discuss the notion of LISP-simplification.

3.1 LISP-SIMPLIFICATION OF LOGIC EXPRESSIONS.

The expressions encountered by the LOGIC processor during the deduction cycle are terms and predications arising ultimately from the input constraint list and from the assertions used in constructing resolvents. However, some of these LOGIC expressions may also admit an interpretation as LISP programming constructs. In that case they may have a LISP value, or if not they may be capable of some simplification.

For example, the expression

```
(+ 3 (* 5 4))
```

is both a LOGIC term and a LISP construct. (We allow short names +, -, * and % for the LISP functions PLUS , DIFFERENCE, TIMES and QUOTIENT.) In the latter role, it is equivalent to, and can be replaced by, its "value", namely the numeral

```
23
```

within any normal expression A to produce an expression which has the same meaning as A. Such replacements of expressions by others which are their values are basic equivalence-preserving transformations of ordinary computation as normally conceived. The presence of free variables does not invalidate this idea. Thus even though "a" has no LISP value the LISP construct

```
(+ a (* 5 4))
```

can be simplified; it is LISP-equivalent to and can be replaced by the simpler expression

```
(+ a 20)
```

even though the latter is not its "value" as in the first case. In general, an expression may well "reduce" to another expression even when it will not, in the usual sense, "evaluate" to a "value".

We refer to this process of replacing a LOGIC expression by one which is LISP-equivalent to it as "LISP-simplification" . It can be done to any expression at any time and is always defined (but may be merely the identity transformation).

3.2 LISP DEFINITIONS.

Certain reduction rules are built into LISP itself and come with the system whenever one sets up a LISP workspace. That is, certain identifiers are defined as denoting built-in LISP functions (CAR, CDR, PLUS, etc.) or as the keywords of built-in special forms (COND, SETQ, PROGN, etc.).

In addition to these built-in LISP definitions, a LISP workspace may contain further definitions made by the user. A collection of such user-coined LISP definitions indeed constitutes a LISP program.

3.3 REDUCTIONS, VALUES AND SIMPLIFICATIONS.

The joint effect of the system- and user-imposed definitions in a LISP workspace is to determine a notion of "reduction".

Each LISP construct is either reducible or irreducible. If A is reducible, then there is another LISP construct called the reduction of A, to which A is LISP-equivalent and by which it may be replaced.

Accordingly we say that the simplification of an irreducible LISP construct A is A itself, while the simplification of a reducible construct A is the reduction of A. Thus simplification is always defined. It often coincides with evaluation - that is, the value of A and the simplification of A are often identical. But this is not always the case and the matter requires some care.

For example, the expression

```
(QUOTE (This is an S-expression))
```

is evaluable and has as its value the expression

```
(This is an S-expression)
```

but it is irreducible and hence is its own simplification.

The expression

```
(* (+ 3 4) (% 5 x))
```

has no value (since its second argument expression contains an occurrence of a variable) but simplifies to the expression

```
(* 7 (% 5 x))
```

In general if a LISP construct A has an atomic value V which is a numeral or a truth value or a constant identifier, its simplification is also V. However, if V is not atomic, or is a variable, the simplification of A is the expression (QUOTE V), rather than the expression V. This is at first a somewhat surprising feature of the simplification notion. A little reflection soon shows its naturalness.

The intuitive notion of simplification is that it always yields an expression which cannot be further simplified - which is irreducible. Moreover, an expression A must be LISP-equivalent to the simplification of A - and this means that if A has the value V so must the simplification of A. These two considerations together require that the simplification of A be (QUOTE V) - the value of which is V - whenever the expression V might itself be evaluable and have a value W distinct from V. Only when V is a constant is W necessarily identical with V.

3.4 REDUCTION AND EVALUATION

Generally speaking, we consider that the "applicative" expression e = (f e1 ... eN) is *evaluable* if f is the name of a function (defined in LISP) and e1, ..., en are themselves evaluable. In this case we also say that e is *reducible* and that the reduction of e is the value of e, quoted when necessary as explained above. The latter is obtained by applying f to the values of the argument expressions.

The reduction of an arbitrary applicative expression, in general, is obtained by replacing occurrences of its outermost reducible subexpressions by occurrences of their reductions.

We proceed now to precise definitions of the notions of evaluability, reducibility, and reduction. We shall speak of the expressions in question as though they were explicitly represented. In fact, in the LOGLISP system we compute the reduction of an expression directly from its implicit representation, as economically as we can. The resulting reduction is also represented implicitly, with the same environment component.

3.4.1 Evaluable Expressions.

Excepting certain special forms which are discussed below, we say that the expression e = (f e1 ... eN) is <u>evaluable</u> if

(1) f is an identifier with property EXPR, SUBR, LSUBR or MACRO and e1, ..., eN are evaluable, in which case the value of e is obtained by applying f to the values of e1, ..., eN.

(2) f is an identifier with property FEXPR or FSUBR, in which case the value of e is the result of applying f to the expression list (e1 ... eN). (This is just the standard notion of "application" for FEXPR's and FSUBR's.)

An atom which is not a variable is evaluable and its value is itself. Variables are not evaluable.

3.4.2 Reducible Expressions

Again excepting certain special forms, an applicative expression e = (f e1 ... eN) is <u>reducible</u> if

(a) e is evaluable as above, in which case the reduction of e is the value of e, v, if v is an atom which is not a (logic) variable, otherwise (QUOTE v)

or e is not evaluable, but

(b) f is a proper name and one or more of the expressions e1, ..., eN is reducible, in which case the reduction of e is (f e1' ... eN'), where ei' denotes the simplification of ei.

Note that atoms, whether variables or not, are irreducible. Note further that expressions (f e1 ... eN) in which f is a variable, a number, or, indeed, anything except a proper name, are neither evaluable nor reducible. This convention may be justified intuitively on the ground that one doesn't know what to do in such a case. We could, in fact, extend the definitions to allow f to be a lambda expression, say, but have chosen not to do so for the present. Such an extension would complicate matters significantly with no great advantage in flexibility.

3.5 SPECIAL FORMS.

In addition to the expressions just considered there are a number of special forms which are evaluable or reducible or both. Most of these are special forms of LISP.

Since the syntax of special forms is the same as that of applicative forms whose function designator is atomic, LISP users often slur over the distinction. It is, however, most important to remember that the LISP value of a special form is NOT obtained by "applying the function denoted by its head to the object denoted by its tail" - that being how the LISP value of an APPLICATIVE form is obtained.

There is a special process set up for obtaining the LISP value of each special form, to which a LISP interpreter switches on recognizing the keyword (COND, SETQ, PROGN, QUOTE, etc.) of that special form.

This little homily would not be necessary if the syntax of applicative forms were designed in the same way, and applicative forms were tagged as such by a keyword, say, APP. The high frequency of applicative forms in programs would make such a convention burdensome. No one wants to have to write

```
(APP + (APP * 3 4)(APP SIN 30))
```

instead of

```
(+ (* 3 4)(SIN 30))
```

3.5.1 Quotations.

(QUOTE v) or (FUNCTION v)

These forms are always evaluable and never reducible. The value of either form is v.

3.5.2 Listings.

(LIST e1 ... eN)

LIST is treated as though it were an ordinary function (an LSUBR, say) despite the fact that UCI LISP implements LIST by means of an FSUBR. This is just what one would naively expect.

3.5.3 Conjunctions.

(AND e1 ... eN)

(AND) is evaluable and reducible with value (and reduction) T. (AND e) is reducible and its value and reduction are the value and reduction, respectively, of e. If e1 is evaluable and its value is NIL then (AND e1 ... eN) is evaluable and reducible and its value and reduction are NIL. If e1 is evaluable with a non-NIL value .then the evaluability, reducibility, value, and reduction of (AND e1 ... eN) are those of (AND e2 ... eN). If e1 is not evaluable then (AND e1 ... eN) is reducible just in case e1 is reducible, and the reduction of it is (AND e1' e2 ... en), e1' being the reduction of e1. All of this corresponds to LISP usage, the conjuncts being evaluated in order and only as far as necessary to determine the result.

3.5.4 Disjunctions.

(OR e1 ... eN)

(OR) is evaluable and reducible with value (and reduction) NIL. (OR e) is reducible and its value and reduction are the value and reduction, respectively, of e. If e1 is evaluable and its value is non-NIL then (OR e1 ... eN) is evaluable and reducible and its value and reduction are the atom T. If e1 is evaluable with value NIL then the evaluability, reducibility, value, and reduction of (OR e1 ... eN) are those of (OR e2 ... eN). If e1 is not evaluable then (OR e1 ... eN) is reducible just in case e1 is reducible, and the reduction of it is (OR e1' e2 ... en), e1' being the reduction of e1. All of this corresponds to LISP usage, the disjuncts being evaluated in order and only as far as necessary to determine the result.

3.5.5 Conditionals.

(COND q1 ... qN)

(COND) is evaluable and reducible and its value and reduction are NIL. If q1 is (e0 ... eM) then (COND q1 ... qN) is reducible (and possibly evaluable) just in case e0 is reducible or evaluable. If e0 is reducible but not evaluable then (COND q1 ... qN) is reducible with reduction (COND (e0' ...eM) ... qN) and is not evaluable. If e0 is evaluable with non-NIL value v, then (COND q1 ... qN) is reducible and its evaluability, reduction and value are those of (PROGN (QUOTE v) e1 ... eM). If e0 is evaluable with value NIL then the evaluability, reduction, and value of (COND q1 ... qN) are those of (COND q2... qN). All of this conforms to customary LISP practice, since PROGN mimics the sequential evaluation of the expressions in a conditional "arm".

3.5.6 Sequential Compositions.

(PROGN e1 ... eN)

(PROGN) is evaluable and reducible with value and reduction T. (PROGN e) is reducible and its evaluability, reduction, and value are those of e. If e1 is reducible but not evaluable then (PROGN e1 ... eN) is reducible with reduction (PROGN e1' e2 ... eN), e1' being the reduction of e1. If e1 is evaluable then (PROGN e1 ... eN) is reducible and its evaluability, reduction, and value are those of (PROGN e2 ... eN).

(PROG1 e1 ... eN)

(PROG1) is evaluable and reducible with value and reduction T. (PROG1 e) is reducible and its evaluability, reduction, and value are those of e. If e1 is reducible but not evaluable then (PROG1 e1 ... eN) is reducible with reduction (PROG1 e1' e2 ... eN), e1' being the reduction of e1. If e1 is evaluable with value v then (PROG1 e1 ... eN) is reducible and its evaluability, reduction, and value are those of (PROGN e2 ... eN (QUOTE v)).

(PROG loc s1 ... sN)

PROGs are neither evaluable nor reducible. There is no reasonable way to carry out a reduction of a PROG analogous to the reduction of PROG1 or PROGN expressions, and the necessity of assignment to the local identifiers of the PROG would lead to limited utility of such a construct, even if we were to define some notion of reducibility for PROGs. PROG may, of course, be used freely in the definitions of functions invoked from LOGIC.

3.5.7 Assignments.

```
(SETQ ident e)
```

If e is evaluable with value v and ident is a constant identifier then (SETQ ident e) is evaluable with value v, and assigns v to ident. Otherwise (SETQ ident e) is reducible just in case e is reducible and its reduction is (SETQ ident e'), where e' is the reduction of e.

Note that assignment should be used with extreme caution in LOGIC, since the order in which assignments are performed is determined in part by the heuristic search methods, and thus is not readily predictable. Observe too that in order to obtain the LISP value of an identifier I one must write (EVAL I), not just I.

3.5.8 Selections.

```
(SELECTQ e (q1 e11 ... e1k1) ... (qN eN1 ... eNkN) u)
```

The evaluation and reduction of the SELECTQ are basically the same as the evaluation and reduction of

```
(COND ((EQ e q1) e11 ... e1k1)
        .
        .
        .
       ((EQ e qN) eN1 ... eNkN)
       (T u))
```

except that reductions are expressed with SELECTQ and e is evaluated just once at the beginning. If one of the selection keys qi is a list (i1 ... im) then the corresponding COND predicate is

```
(MEMQ e (LIST i1 ... im))
```

3.6 LOGLISP SPECIAL FORMS.

The remaining special forms do not correspond to anything in conventional LISP. They provide means by which the LOGIC programmer may control the interaction between LOGIC and LISP in order to deal with various unusual circumstances.

(LOGIC e)

Intuitively, the LOGIC form specifies that the result of evaluation is to be regarded as a logic expression rather than as an object, the effect most often being to suppress the normal quoting of non-atomic values.

More precisely, if e is evaluable with value v then (LOGIC e) is reducible, (LOGIC e) is evaluable according as v is evaluable, and the reduction and value of (LOGIC e) are the reduction and value of v. If e is not evaluable, (LOGIC e) is reducible according as e is reducible and the reduction of (LOGIC e) is (LOGIC e'), where e' is the reduction of e. Put differently, when e is evaluable, we reduce or evaluate (LOGIC e) by treating the value of e, v, as a logic expression and reducing or evaluating v. In practice it usually happens that v is neither evaluable nor reducible, in which case (LOGIC e) reduces to v.

(LISP e)

The form (LISP e) indicates that e is itself to be treated as the value of (LISP e). More precisely, (LISP e) is never reducible; it is always evaluable and its value is e.

(GROUND e)

The form (GROUND e) is similar to (LISP e), but is evaluable only if no variables occur in e. More precisely, (GROUND e) is never reducible; it is evaluable if no variable occurs in e, in which case its value is e.

(LOGIC-GR e)

(LOGIC-GR e) is precisely equivalent to (LOGIC (GROUND e)). It follows that if any variable occurs in e then (LOGIC-GR e) is neither evaluable nor reducible. If no variable occurs in e then (LOGIC-GR e) is reducible and its evaluability, reduction, and value are those of e. We shall illustrate a few applications for these forms. First, consider

```
(LOGIC (SUBST (GROUND x) (GROUND y) (GROUND z)))
```

which, as it stands, is neither evaluable nor reducible. Suppose now we instantiate to obtain

```
(LOGIC (SUBST (GROUND (+ (VAR A) 3))
              (GROUND (VAR Q))
              (GROUND (<= (VAR Q) 10))))
```

where VAR is not the name of a LISP function. Since no variables occur in the inner expressions these are evaluable, the expression (SUBST ...) is evaluable, hence the whole reduces to

```
(<= (+ (VAR A) 3) 10)
```

The abbreviation LOGIC-GR is sometimes useful in connection with FEXPR's. If f is the name of a FEXPR or FSUBR then (LOGIC-GR (f e1 ... en)) is evaluable and reducible just in case no variable occurs in any of the e's, in which case the value and reduction of (LOGIC-GR (f e1 ... eN)) are the value and reduction of (f e1 ... eN). This treatment of FEXPR's is sometimes a useful alternative to the customary procedure described earlier.

3.7 SIMPLIFYING IMPLICIT CONSTRAINTS. THE FUNCTION SIMPLER.

If C = (Q E) is an implicit constraint and D is a knowledge base, then (SIMPLER C D) is the implicit constraint which results from simplifying one or more of the predications in C and dropping them if they simplify to "true". Specifically, (SIMPLER C D) is the result of the following three-step algorithm:

```
1  let (Q E) = C

2  while  Q is nonempty

   and    the simplification of A{E}
          is evaluable and not NIL

   where  (L A R) is the (SEL Q E D)-decomposition of Q

   do     replace Q by L*R

3  return (Q E)
```

3.8 THE EXTENDED DEDUCTION CYCLE.

In the actual LOGIC cycle of our LOGLISP system we include a step of LISP-simplification in step 1 of the RUN loop. The full description of the loop is then:

```
RUN:  while   WAITING is nonempty

      do  1   remove some C from WAITING
              and let (X Y) be (SIMPLER C D)

          2   if   (X Y) is solved
              then add (X Y)  to SOLVED
              else add the (D K)-resolvents of (X Y) to WAITING
              where K = (SEL X Y D)
```

Note that the K selected in step 2 will be the same as that selected in the final iteration of SIMPLER. (Indeed, in LOGLISP this is obviously so since K = 1 uniformly; but it is true for every SEL function).

This means that the predication resolved away is the one which was just processed by SIMPLER and that it is therefore a LISP-irreducible expression. In particular it may be the expression NIL (i.e. the LISP representation of FALSE). In this case, there will be no resolvents forthcoming and (X Y) will therefore be a failure.

3.9 UNIFICATION IN LOGLISP

There are a few points worth noting about the LOGLISP implementation of unification.

First of all, there is no check performed to see if a unification has created any cycles. Such a check would, if routinely made, be time-consuming. It appears that in normal LOGIC programming the check is unnecessary. Since unification is confined to the cases where the input expressions do not have variables in common, cycles can arise only if assertions or queries are formulated in certain abnormal ways.

The use of implicit representations throughout in any case makes it possible to work with infinite (cyclic) expressions as though they were finite (which in a suitable sense they are). It is only when a sophisticated user wishes to exclude such expressions from the domain of discourse that their detection becomes necessary.

Of course, any process (such as a naive recursive realization) which seeks to traverse every path in such an expression will run on indefinitely, and the user will want to avoid this situation. In designing LOGLISP we have assumed that any user deliberately creating such expressions will be sophisticated enough to use LISP to protect himself without being lectured at by us. We have further assumed that any user inadvertently creating such expressions will prefer to take the error messages or other indications of his mistake which LISP will provide - in place of the expensive LOGLISP overhead which would be needed to protect him from them.

3.9.1 Atoms

Two atoms, say a1 and a2, neither of them variables, are considered to be unifiable iff (EQUAL a1 a2) or both are strings and have the same characters. Thus the condition for unifiability can be expressed as

```
(OR (EQUAL a1 a2)
    (AND (STRINGP a1) (STRINGP a2)
         (EQUAL (EXPLODE a1) (EXPLODE a2)))) .
```

This produces just the effect one wants, but note that distinct identifiers with the same PNAME are not unifiable (it cannot be the case that both are INTERNed). The integer 1 unifies with the floating-point number 1.0, on the other hand, and distinct occurrences of the same floating-point value are unifiable.

3.9.2 Special Forms

Expressions in QUOTE and FUNCTION are treated specially. (QUOTE e1) unifies with (QUOTE e2) iff (EQUAL (QUOTE e1) (QUOTE e2)), and similarly for (FUNCTION f1) with (FUNCTION f2). In addition, expressions of the form (CONS e1 e2) may unify with expressions (QUOTE (a . d)). In attempting to unify two such expressions any logic variables appearing in (a . d) will be treated as constants. Let us denote by q[e] the expression e if e is an atomic constant, and (QUOTE e) otherwise. The unifier proceeds by attempting to unify e1 with q[a], then, if successful, unifying e2 with q[d]. Variables in e1 and e2 will be bound to subexpressions of a and d, QUOTEd when appropriate. Some examples will make things clear. The expression

(CONS x y)

unifies with

(QUOTE (A B C))

with mgu x = A, y = (QUOTE (B C)). To take a more complicated case,

(CONS (CONS F x) (CONS u v))

unifies with

(QUOTE ((F (A B)) C D))

with mgu

x = (QUOTE ((A B))), u = C, v = (QUOTE (D)) .

Expressions in QUOTE and FUNCTION are not otherwise unifiable. It should be remarked that an expression like (F A QUOTE (B)) does not contain an expression in QUOTE, merely an occurrence of the constant QUOTE.

3.9.3 Variables As Tails

Ordinarily, an expression is either an atom or a list, but one may, in fact, introduce expressions which are composite but not lists. The only useful expressions of this class are those for which repeated CDR's eventually yield a variable, an example being (P (F x) . y). We remark that the definitions of unification and resolution given in chapters 1 and 2 do not actually require that non-atomic expressions be lists. In a sense, there is really nothing special about a composite expression which is nOt a list, but such expressions are sufficiently unusual that a bit more discussion may be in order. Expressions of this sort are particularly useful in dealing with operators which take a variable number of arguments. To illustrate, the expression

(+ x . y)

unifies with

(+ u 7)

with mgu

x = u, y = (7)

and also unifies with

```
(+ (F u 3) 7 (G A B))
```

with mgu

```
x = (F u 3), y =(7 (G A B)) .
```

Thus a simple, but still rather flexible, rule for solving equations involving sums may be asserted by

```
(|- (= (+ x . y) z) <- (= x (- z (+ . y)))) .
```

3.10 REDUCTION OF EXPRESSIONS ENDING IN VARIABLES.

The reduction of an expression (f e1 ... eN . v) will now be explained. Such an expression is evaluable if and only if f is the name of a FEXPR or FSUBR, in which case the value is the result of applying f to (e1 ... en . v), an argument "list" with which few FEXPRs are prepared to cope. If f is a proper name, but not the name of a FEXPR or FSUBR, then the expression is not evaluable, but is reducible if any of e1, ..., eN are reducible, in which case the reduction is (f e1' ... eN' . v).

The sequentially evaluated LISP forms, those formed with AND, OR, COND, PROGN, PROG1 and SELECTQ, may also involve variable tails. Reduction proceeds as described before, stopping when a variable tail is encountered. Such expressions may be evaluable if the "evaluation path" avoids variable tails entirely.

3.11 SPECIAL RULES FOR RESOLUTION.

The system "automatically" incorporates a number of special rules applicable to certain predicate symbols. In most cases these rules are just economical implementations of computations that could be achieved with ordinary assertions, but the rule for CONDitional expressions constitutes a fundamental extension of the system, as it introduces a form of "negation as failure". Application of any of the rules can be enabled or disabled at will by the user.

3.11.1 The Rules.

Each of the rules is introduced by an informal, assertion-like description, followed by discussion and, in some instances, a nearly equivalent formulation with actual assertions.

3.11.1.1 Equations -

(= e1 e2) <- "e1 and e2 are unified"

The rule is just the reflexive law of equality, and amounts to

(¦- (= x x)) .

This rule is not applicable to expressions of the form (EQUAL e1 e2), even though = and EQUAL denote the same function for purposes of simplification.

3.11.1.2 Conjunctions -

(AND p1 ... pN) <- p1 & ... & pN

Bearing in mind that (AND) simplifies to T, the rule for AND amounts to

(¦- (AND x . y) <- x & (AND . y)) .

3.11.1.3 Disjunctions -

(OR p1 ... pN) <- pi, for i = 1 ... N

Again, bear in mind that (OR) simplifies to NIL. The rule for OR is practically equivalent to

(¦- (OR x . y) <- x)
(¦- (OR x . y) <- (OR . y))

except that resolvents for all of the disjuncts are obtained in one step.

3.11.1.4 Conditionals -

(COND (p1 q1) ... (pN qN)) <- pk & qk, for the first k such that pk is provable

Let us refer to the constraint from which (COND ...) was selected for resolution as the "original constraint". The control mechanism, in fact, begins by attempting to prove p1. If it succeeds in doing so, it introduces a new resolvent consisting of qk and the other predications of the original constraint in the environment which proved p1. (Such a resolvent will eventually be produced for each proof of p1, if the search continues so long.) If all attempts to prove p1 terminate in failure then then control mechanism attempts to prove p2, and so on. All of these searches are carried out within the heuristic limitations imposed on the problem at the beginning. These searches are, moreover, carried out "in parallel" with searches for other solutions to the initial problem, in accordance with the standard heuristics, so that depth-first runaway will be avoided to the extent possible.

The "arms" of the CONDitional expression need not have exactly two expressions. An arm of the form (pk) is, for purposes of resolution, equivalent to (pk T), while an arm of the form (pk qk1 ... qkm) is equivalent to (pk (PROGN qk1 ... qkm)).

This treatment of conditionals depends on a feature of the system not hitherto mentioned, namely the ability to associate a "continuation" with a node. The continuation is itself just a node of a somewhat special nature which is not itself available for computing resolvents. We write a node C with continuation K as "C Continuation: K". The resolvents of C Continuation: K are exactly the nodes R Continuation: K such that R is a resolvent of C.

Let (X Y) be a node whose resolvents are desired, let the selected component of X be P, and suppose that P{Y} has the form (COND (p1 q1) ... (pN qN)). We obtain a "resolvent" which is

```
((p1)) Y) Continuation: ((#COND (q1) (p2 q2) ... ))*X' Y)
```

where X' consists of the unselected predications of X. Each proof of p1 generates a resolvent (NIL Z) with the same continuation, from which we "pop up" the continuation to obtain a resolvent ((q1)*X' Z). If and when all attempts to prove p1 fail, we pop up the continuation to obtain

```
((COND (p2 q2) ... (pN qN))*X' Y)
```

which is added to WAITING.

Continuations are not usually printed when explaining answers or monitoring deductions, rather the fact that a node has a continuation is indicated by printing "[CONTINUED]". Users can instruct the system to print continuations in full by invoking the command (CONTINUATIONS ON). (CONTINUATIONS OFF) returns the system to the normal mode.

3.11.2 Controlling The Special Resolution Rules.

All of the rules may be enabled or disabled by invoking functions of the form (AUTOx "flag") where flag may be either ON or OFF. The complete set of control functions for the resolutions is

```
(AUTO= "flag")
(AUTOAND "flag")
(AUTO-OR "flag")
(AUTOCOND "flag)
```

Each function returns its argument. T or NIL may be used instead of ON or OFF. While these function behave like FEXPRs for atomic arguments, they evaluate non-atomic arguments, so one could, for example, type

```
*(AUTOAND (AUTO-OR OFF))
```

to disable both the AND rule and the OR rule. All of the rules are enabled by system initialization, hence by RESTORE (see the chapter on filing knowledge bases).

3.12 RESOLUTION WITH DATA PROCEDURES.

A procedure is said to be a <u>data procedure</u> if (and only if) the heads of its assertions are <u>all</u> ground predications (predications containing no variables). In practice such assertions are invariably unconditional, hence <u>data</u>, in the sense in which we earlier used the term. Data procedures often contain a great many assertions, so that a straightforward approach to computing resolvents from them would usually result in numerous futile attempts at unification.

To avoid this source of inefficiency LOGLISP automatically performs secondary indexing of data procedures. This indexing depends on the fact that if the predication P unifies with the head H of a datum, and if the proper name C occurs in P, then C must also occur in H. The secondary indexing enables the system to retrieve promptly just those assertions in which C occurs, and thus obtain all possible resolvents without an exhaustive examination of the entire procedure.

Since the secondary indexing applies only to data procedures, users concerned with efficiency will want to avoid mixing rules with data in the same procedure, at least when the amount of data is significant. This seems, however, to be a natural and convenient way to organize knowledge bases.

CHAPTER 4

CREATING KNOWLEDGE BASES

To create a knowledge base one begins with the empty knowledge base and adds assertions to it one at a time as explained below. Or one can extend an already existing knowledge base by installing it in a LOGLISP workspace and adding more assertions to it. The empty knowledge base is created by executing the command

```
(START)
```

which discards any assertions already present and initializes the LOGIC part of the workspace (without affecting the LISP definitions, if any, which the user may have set up).

4.1 ADDING AN ASSERTION TO THE KNOWLEDGE BASE.

The assertion command

```
(|- B <- A1 ... An)
```

causes the assertion

```
B <- A1 ... An
```

to be added to the current knowledge base. The symbol |- is the assertion symbol. (It is pronounced "assert").

The arrow may be omitted. We shall often omit it in the examples in this manual.

4.1.1 Naming An Assertion.

An assertion may be given a user-coined name. This is most conveniently done at the time the assertion is added to the knowledge base, using an extended assertion command. Execution of the extended assertion command

```
(|- N B A1 ... An)
```

adds the assertion B <- A1 ... An to the current knowledge base, as before, but also ascribes to it the name N. The user-coined name N may be any identifier beginning with an upper-case letter. For example, the following four transactions:

```
*(|-(Born Herbrand 12 February 1908))

ASSERTED
*(|-(Died Herbrand 27 July 1931))

ASSERTED
*(|- TURING1 (Born Turing 23 June 1912))

ASSERTED
*(|- TURING2 (Died Turing 7 June 1954))

ASSERTED
*
```

add four assertions to the knowledge base, the first two of which are anonymous, and the second two of which have been named respectively TURING1 and TURING2. Note that each assertion transaction is terminated by the message ASSERTED. If the assertion is ill-formed the message returned will be ERROR-Ignored, in which case the knowledge base is not altered by the transaction.

The assertions making up a knowledge base are organized into groups called procedures. All assertions in the knowledge base whose conclusions have the same predicate P are grouped together into a procedure which is called "the procedure P". It is thought of, intuitively, as the portion of the knowledge base which is relevant to establishing those facts in the world whose predicate is P.

Assuming that the knowledge base was empty before the above four assertions were added, the contents of the knowledge base now consists of two procedures, each containing two assertions.

By invoking the PRINTFACTS command [see the following Chapter on Displaying Knowledge Bases] the contents of the knowledge base can be displayed, its clauses organised into procedures. Thus:

```
*(PRINTFACTS)

FACTS-ASSERTED

(PROCEDURE Born)

(¦- (Born Herbrand 12 February 1908))

(¦- TURING1 (Born Turing 23 June 1912))

(PROCEDURE Died)

(¦- (Died Herbrand 27 July 1931))

(¦- TURING2 (Died Turing 7 June 1954))

END

*
```

4.2 THE FACTS MODE

A somewhat more convenient way of adding a succession of assertions is provided by the FACTS mode. By executing the command (FACTS) the user puts the system into the FACTS mode. This is simply a wait-read-assert cycle which expects successive assertions to be typed in. The prompt-message ASSERT: is printed by the system to signify its readiness to receive the next assertion. Thus the four assertions of our example could have been added by means of the following excursion through the FACTS mode:

```
*(FACTS)

ASSERT: ((Born Herbrand 12 February 1908))

ASSERT: ((Died Herbrand 27 July 1931))

ASSERT: (TURING1 (Born Turing 23 June 1912))

ASSERT: (TURING2 (Died Turing 7 June 1954))

ASSERT: ;

DONE

*
```

Such a FACTS session is terminated by typing a semicolon in response to the ASSERT: prompt. It should be noted that the format in which an assertion B <- A1 ... An is typed for input to the FACTS mode is the list (B A1 ... An) . The first item on this list may be the optional user-coined name, as illustrated above. The list format enables the system to accept inputs which are too large to fit all on one line. As in the standard LISP convention, the system reads line after line of typed input until a syntactically complete object has been formed. Thus in the following FACTS transaction the three-component assertion AGE-FORMULA is asserted on several lines, each of which after the first is prompted by a colon:

```
*(FACTS)

ASSERT: (AGE-FORMULA
:                  (Age person given-year a)
:                  (Born person day month birth-year)
:                  (= a (- given-year birth-year)))

ASSERT: ;

DONE
```

The assertion AGE-FORMULA is now installed as the sole component of a procedure Age which computes a person's age in a given year by looking up the year in which that person was born and subtracting it from the given year. The contents of the knowledge base may again be viewed by executing (PRINTFACTS) :

```
*(PRINTFACTS)

FACTS-ASSERTED

(PROCEDURE Born)

(¦- (Born Herbrand 12 February 1908))

(¦- TURING1 (Born Turing 23 June 1912))

(PROCEDURE Died)

(¦- (Died Herbrand 27 July 1931))

(¦- TURING2 (Died Turing 7 June 1954))

(PROCEDURE Age)

(¦- AGE-FORMULA (Age person given-year a)
                <- (Born person day month birth-year)
                 & (= a (- given-year birth-year)))

END

*
```

The "<-" and "&" appearing in AGE-FORMULA are simply "syntactic sugar" intended to assist the reader in perusing complex assertions. These may also be typed in assertions given to ¦- or FACTS, but we usually don't bother to do so.

An ill-formed assertion typed to FACTS will be ignored, and a message will be typed to inform the user.

4.3 ADDING ASSERTIONS FROM LISP FUNCTIONS.

The assertion function ¦- is just a LISP FEXPR, and as such may be invoked by any LISP function. LISP programmers will usually find it more convenient, however, to use the SUBR-type function ASSERTCLS of one argument, whose value should be a list as might be typed to FACTS (or appear as the tail of an invocation of ¦-). If the assertion is well-formed it will be added to the knowledge base and ASSERTCLS will return NIL. If the assertion is ill-formed it is ignored and ASSERTCLS returns ERROR.

CHAPTER 5

DISPLAYING KNOWLEDGE BASES

Various commands are provided for viewing the contents of a knowledge base.

5.1 DISPLAYING THE ENTIRE CONTENTS OF A KNOWLEDGE BASE.

The command (PRINTFACTS) causes the system to print out a display of the entire current knowledge base. The display is organised into groups of assertions preceded by the message FACTS-ASSERTED. Each group of assertions constitutes a (logical) procedure. That is to say, the header of every assertion in the group has the same predicate symbol (say, P) . The predicate symbol P is used as the name of the procedure, and the group of assertions is accordingly preceded by the line: (PROCEDURE P) . The constituent assertions of the procedure P are then displayed in the form of assertion commands. The order in which the assertions appear in the display is the chronological order in which they were originally asserted. The display is terminated by the message END.

5.2 DISPLAYING A PROCEDURE.

The command (PRINTFACTSOF P) displays the procedure P in the same style as that of the (PRINTFACTS) display. If one wishes to print several procedures one types (PRINTFACTSOF P1 ... PN). Further, the standard function GRINDEF has been altered to print logic procedures in addition to the properties which it ordinarily prints. These too are in PRINTFACTS format.

The command (PRLENGTH P) returns the number of assertions in the procedure P:

```
*(PRLENGTH Born)

2.
*
```

5.3 DISPLAYING THE SET OF DEFINED PREDICATES.

The command (PREDICATES) returns a list of the predicates for which logic procedures are defined in the current knowledge base. With the example of the preceding chapter we have:

```
*(PREDICATES)

(Born Died Age)
*
```

5.4 DISPLAYING DATA ASSERTIONS IN WHICH A GIVEN PROPER NAME OCCURS.

It is often convenient to be able to retrieve and display the set of data assertions in a given knowledge base in which a given notion occurs explicitly. Such a set in some sense corresponds to what the knowledge base says about that notion in a direct way. The command (PRINTCREFSOF C) displays all assertions belonging to data procedures (in the sense of chapter 3) in whose header the constant C appears somewhere. These assertions are organised into groups by their procedure name, but the entire procedure is not necessarily shown (only those of its assertions whose headers actually contain C).

5.5 RETRIEVING A PROCEDURE AS A LIST

The procedure P may be obtained as a LISP data object, namely, as the list of its constituent assertions. This list is returned as the value of the command

```
(ASSERTIONSOF P)
```

Each assertion B <- A1 ... An in the procedure is represented as the list (B A1 ... An) . If the assertion has the user-coined name N then it is represented as the list (N B A1 ... An) . For example, (ASSERTIONSOF Born) returns the list

```
(((Born Herbrand 12. February 1908.))
 (TURING1 (Born Turing 23. June 1912.)))
```

The result of ASSERTIONSOF shares no list structure with the internal representation of the knowledge base, thus list-altering operations such as RPLACA and RPLACD performed on this list will have no effect on the knowledge base.

CHAPTER 6

EDITING KNOWLEDGE BASES

The resident editor of the LISP system has been extended so as to allow the editing of knowledge bases in essentially the same style as is used to edit LISP functions and data objects. The edit command EDITA is used to enter the editor when LOGIC editing is to be done. The following editing session will illustrate the way this works. We will use the editor to attach names to the (at present) anonymous assertions in the current knowledge base, and to change the name AGE-FORMULA to AGE-RULE.

```
*(EDITA Born)

EDIT
#P

((& ) (TURING1 &  ))
#PP

(((Born Herbrand 12 February 1908))
 (TURING1 (Born Turing 23 June 1912)))
#1 PP
((Born Herbrand 12 February 1908))
#(-1 HERBRAND1) PP
(HERBRAND1 (BORN Herbrand 12 February 1908))
#OK

Born
*(EDITA Died 1 (-1 HERBRAND2) PP)

(HERBRAND2 (Died 27 July 1931))

Died
*(EDITA Age 1 (1 AGE-RULE) PP)

(AGE-RULE (Age person given-year a)
          (Born person day month birth-year)
          (= a (- given-year birth-year)))

Age
*
```

This editing has produced the desired changes, as may be seen if we display the resulting knowledge base:

```
*(PRINTFACTS)

FACTS-ASSERTED

(PROCEDURE Born)

(¦- HERBRAND1 (Born Herbrand 12 February 1908))

(¦- TURING1 (Born Turing 23 June 1912))

(PROCEDURE Died)

(¦- HERBRAND2 (Died Herbrand 27 July 1931))

(¦- TURING2 (Died Turing 7 June 1954))

(PROCEDURE Age)

(AGE-RULE (Age person given-year a)
          <- (Born person day month birth-year)
           & (= a (- given-year birth-year)))

END
*
```

which is what we wanted. If one wishes to edit several procedures simultaneously one types (EDITA (P1 ... PN)). In this case one edits the assertions for all of the procedures as a single list. For example:

```
*(EDITA (Born Died))

EDIT
#P

((HERBRAND1 &) (TURING1 &) (HERBRAND2 &) (TURING2 &))
#OK

(Born Died)
*
```

As with the LISP edit functions EDITF and EDITV, one may also specify one or more editor commands after the predicate (or list of predicates) to be edited. In such cases the commands are performed and the editor returns without further interaction with the user. For this reason it is very important that one remember the parentheses when specifying several procedures. Performing (EDITA P1 ... PN) will usually result in an editor error, in which case P1 will be erased from the knowledge base.

6.1 REMOVING PROCEDURES FROM THE KNOWLEDGE BASE.

If one wishes to remove one or more procedures P1, ..., PN from the current knowledge base one invokes the command (ERASEP P1 ... PN).

CHAPTER 7

FILING KNOWLEDGE BASES

The current knowledge base may be preserved in a file by the LOGIC primitive SAVE. The command (SAVE N) creates on the disk a file named N [N is a user-coined name of no more than six characters of which the first is an upper-case letter.] When this work is completed the message DONE is printed.

The file created by SAVE is written on the user's file structure DSK: as conventional text, though not so prettily formatted as with PRINTFACTS. An extension may be specified with the file name, in which case the form is (SAVE (NAME . EXT)), following the usual LISP convention. An extension is supplied only when explicitly specified.

7.1 RESTORE AND LOADLOGIC

A file which has been created by a command (SAVE N) may be later read into primary storage by means of a (RESTORE N) or a (LOADLOGIC N) command. The command (RESTORE N) restores the knowledge base to its contents as of the time the (SAVE N) command which created the file was executed. The command (LOADLOGIC N) adds the procedure clauses in the saved knowledge base N to the procedure clauses in the current knowledge base. RESTORE first clears out the current knowledge base whereas LOADLOGIC does not.

File names with extensions are specified just as for SAVE. The file in question must be found on file structure DSK:, but the project-programmer number of the area from which the file is to be read may be specified separately. The most convenient form is (RESTORE [proj,prog] file). Note that LOGLISP normally expects numeric input in decimal, while ppn's are usually written in octal. One way around this small difficulty is to use a sequence such as:

```
*(SETQ IBASE 8.) (RESTORE [733,21] PLACES) (SETQ IBASE 10.)

8.

DONE

10.
*
```

7.2 ADDTO, BUILD AND DSKIN

The existing primitives of LISP's filing system have been adapted appropriately for use in LOGLISP. In LISP the command (ADDTO N P1 ... Pk) adds the LISP objects named P1, ..., Pk to the file named N (and opens a new file named N if one does not already exist). In LOGLISP the effect of ADDTO is extended so that the objects Pi can also be LOGIC objects, namely logical procedures. Thus the command

```
(ADDTO BIOG Born Died)
```

creates (assuming it does not already exist) a file named BIOG and records that its members are the logical procedures Born and Died. The command

```
(ADDTO BIOG Age)
```

then extends the description of the file BIOG by recording that the logical procedure Age is also a member. Once a file has been opened and described by one or more ADDTO commands, it may be constructed and written on the disk by means of the BUILD command. The command (BUILD N) writes out onto disk storage the current members of the file N in their current condition.

7.3 DSKIN

Files created and stored using the ADDTO and BUILD primitives may be read into primary storage from the disk by means of the DSKIN primitive. Thus if the file BIOG had been previously written on the disk by execution of the command (BUILD BIOG) , it would be read into primary storage by execution of the command (DSKIN BIOG) . DSKIN is analogous to LOADLOGIC in that no prior clearing of the knowledge base currently in primary storage takes place before the asserting of the procedure clauses in the file. Thus by "disking in" several files of logical procedures one may build up a knowledge base containing them all. The format of files created by BUILD is a series of executable commands. Thus under the current assumptions as to the description of BIOG and the contents of its members, the command (BUILD BIOG) would create the file:

```
(SETQ IBASE 10.)

(SETQ DSKINATOM (QUOTE BIOG))

(DEFPROP BIOG (BIOG . LSP) FILENAME)

(DEFPROP BIOG (Born Died Age) MEMBERS)

(PROCEDURE Born)

(¦- HERBRAND1 (Born Herbrand 12. February 1908.))

(¦- TURING1 (Born Turing 23. June 1912.))

(PROCEDURE Died)

(¦- HERBRAND2 (Died Herbrand 27. July 1931.))

(¦- TURING2 (Died Turing 7. June 1954.))

(PROCEDURE Age)

(¦- AGE-RULE (Age person given-year a)
             <- (Born person day month birth-year)
              & (= a (- given-year birth-year)))
```

The command (PROCEDURE P) declares P to be a logical procedure and may be used to record further pragmatic information about P as explained in the Chapter on Interacting with LOGLISP . The DEFPROP commands are LISP acts of definition, recording on the property list of "BIOG" the information describing BIOG's properties as a filename. The two SETQ commands create the appropriate LISP environment for the execution of the rest of the commands in the file.

CHAPTER 8

DEDUCING ANSWERS TO QUERIES

The deduction machinery of LOGIC is invoked by the deduction commands: ALL, ANY, THE, and SETOF . The first three are LISP FSUBR's which may conveniently be invoked from the terminal or within assertions. SETOF is a SUBR intended for use by LISP programs.

8.1 ALL

The command (ALL X C1 ... Cn) returns a list of simplifications of the instances of the answer template X with respect to all of the environments within which the query clause (C1 ... Cn) is true with respect to the current knowledge base. [These environments are called the answer environments for the query-clause (C1 ... Cn) .]

The answer template X may be a variable, an atom not a variable, or a list of expressions. We emphasize that the answers returned are the expressions (or lists of expressions) obtained by simplifying the instances of the answer template in the solution environments, not the values of those expressions, which need not, after all, be evaluable.

8.2 ANY

The command (ANY K X C1 ... Cn) behaves in a similar manner, except that no more than K instances of X are returned from among those which the corresponding ALL command would return. K is expected to be a nonnegative integer.

8.3 THE

The command (THE X C1 ... Cn) returns the sole member of the list (ANY 1 X C1 ... Cn) , if there is one, and is intended for use only in contexts where it is known that exactly one answer environment exists. If no answer environment exists for the stated query THE returns the identifier No-solutions-found.

8.4 SPECIFYING THE SEARCH WINDOW.

The constraints appearing in invocations of ALL, ANY and THE need not all be predications. They may include limit specifications which determine the search window to be used. The form of a limit specification is

Limit: Value

where "Limit:" is one of TREESIZE:, NODESIZE:, ASSERTIONS:, RULES:, DATA:, and "Value" is a number, the identifier INF (denoting infinity) or a non-atomic expression whose LISP value is a number or INF. These values determine bounds for the corresponding parameters of the search window. Thus one might, in the context of the "tennis" example of chapter 2, ask for

```
(ALL x (Champion x) (Male x) (Older x Pete) RULES: 5)
```

to obtain the set of all those who can be deduced to be male champions older than Pete with no more than five applications of rules.

In the absence of any specification the limits are all taken to be INF, except for RULES, which is never allowed to exceed a limit determined by the implementation, normally 500. See the discussion of initialization in the chapter on interacting with LOGLISP for further details on this point.

8.5 SETOF

The preceding commands are special adaptations of the basic general deduction primitive, SETOF. SETOF takes three arguments. In the command (SETOF S X C) the arguments S, X and C are (LISP) evaluated before the SETOF procedure is entered (SETOF is an EXPR). The first argument S (the "scope indicator") is an expression which evaluates either to a nonnegative integer or else to the identifier ALL . The second argument X is an expression which evaluates to an answer template. The third argument C is an expression which evaluates to a query clause. The command (SETOF S X C) returns a list of the instances of the answer template [which is the value of] X corresponding to the answer environments in which the query clause [which is the value of] C is true with respect to the current knowledge base. If the value of S is ALL , then all such instances are in the list returned. If the value of S is the integer K , then no more than K such instances are returned. Thus the command (ALL (x y) (Age x 1928 y)) is equivalent to the command

```
(SETOF (QUOTE ALL) (QUOTE (x y)) (QUOTE (Age x 1928 y)))
```

and both return the list

```
((Turing 16) (Herbrand 20))
```

as their result, if the current knowledge base contains only the procedure clauses HERBRAND1, HERBRAND2, TURING1, TURING2 and AGE-RULE. The command (The logician (Born logician something February 1908)) returns the result: Herbrand .

8.6 NONDETERMINACY OF DEDUCTIVE PROCESSES

The order of the items in the lists returned by ALL, ANY and SETOF is not defined, nor is there defined any rule for selecting a subset of all instances when less than all are requested.

This non-determinacy is accompanied by a measure of "concurrency", in that the order in which LISP evaluations will be performed in the course of various simplifications is also not specified. The evaluation of a single evaluable expression is, however, carried out "indivisibly". It is for this reason that assignment and other effect-producing operations must be used with caution in logic.

8.7 CONTROLLING THE DEDUCTION PROCESSS.

Having emphasized the non-determinacy of the deduction process, we should now point out that the user can, in fact, exercise a considerable degree of control over the deduction, even to the point of making it fully determined.

8.7.1 The Heuristic Solution Cost.

Recall from chapter 2 that the node selected for further progress is always one whose heuristically estimated solution cost is least. This cost is computed as

```
*DEPTHCxASSERTIONS(X Y) + *LENGTHCxNODESIZE(X Y)
```

where *DEPTHC and *LENGTHC are (global) LISP identifiers, both set initially to 1. These coefficients may, however, be set to any integer values one likes, so long as the magnitudes of the resulting costs are not so large as 2**18.

The standard settings give a reasonable heuristic search sheme, but other setiings may prove useful. If one puts *DEPTHC = 1, *LENGTHC = 0, for example, the resulting search is breadth first, while setting *DEPTHC = -1, *LENGTHC = 0 gives depth-first search. In neither of the latter cases is the order in which the deduction tree is explored specified.

8.7.2 The PROLOG Mode.

As mentioned earlier, the user can obtain a strictly determined depth first search by placing the system in the "PROLOG" mode. This is accomplished by the command (PROLOG ON). In this mode the search is, first of all, depth first. The resolvents of a particular node will, moreover, be explored in the order in which the corresponding assertions appear in the knowledge base (this is the order in which the assertions are printed by PRINTFACTS). It is this ordering of the search that distinguishes the PROLOG mode from the depth first search produced by adjusting the solution cost coefficients. If a special rule is in effect for a predicate which also has assertions, the special rule is considered to come before any assertion.

The heuristic search mode is selected by (PROLOG OFF). One is not allowed to change modes while a search is in progress. Any attempt to do so will be met by the response "(Not while searching)". To inquire about the current search mode use the command (SEARCHMODE). In heuristic mode this returns [the value of] (LIST 'HEURISTIC *DEPTHC *LENGTHC), while in PROLOG mode the response is DEPTH-FIRST.

8.8 "ONE RESOLVENT" PROCEDURES.

It sometimes happens that the programmer can determine that on every call of a particular procedure at most one resolvent can lead to success. Such a determination usually depends both on the nature of the queries that can be expected and on the nature of the assertions which constitute the procedure. If it can further be arranged that this resolvent always results from the first assertion which yields a resolvent, then one may inform the system of these facts by specifying the predicate in question to be a "ONERES" procedure. This is done with the command (PROCEDURE Pred ONERES), "Pred" being the predicate of the procedure. If a special rule is in effect for "Pred", the special rule is considered to precede any assertion.

The conditions under which one may appropriately specify a procedure to be "ONERES" may seem rather restrictive, but they are not unusual in practice. An inappropriate ONERES attribution will, of course, have a drastic effect on the meaning of a procedure, since the system will in any case compute only the one resolvent for each call.

CHAPTER 9

MONITORING DEDUCTIONS

Provision has been made for the optional "viewing" of a deduction process as it is happening. Ideally such a facility would show the tree of constraints growing during the execution of the deduction cycle. This is however somewhat extravagant of display space, and LOGIC has a more modest version of this idea.

9.1 THE MONITOR FACILITY

Execution of the command (MONITOR ON) enables the system to display, during the deduction process, the query component of each successive selected constraint. In order to give the user time to reflect, the system pauses once each cycle, and resumes on receiving a suitable input (normally, a semicolon). The predications comprising the query component of (say) the selected constraint (Q E D) are displayed as they appear with the environment E, before any simplification is performed. It should be noted that when viewing a developing deduction process in this way one may observe some discontinuity in the display. This is because the selection mechanism may not always choose a successor of the previous selected constraint, but rather "resume" some older constraint whose turn has arrived for some more "progress". Even though the genetic thread remains unbroken, there may be rather drastic changes in the state owing to the LISP-simplification step of the cycle. The user will soon become accustomed to the realities of the MONITOR display , however , and will find it an enlightening tool when sparingly used to slow down and observe the deductive action. The command (MONITOR OFF) disables the MONITOR facility.

One need not simply continue from the MONITOR pause. The commands one can give are as follows (the prompt is "?*"):

```
?*E expr     -  Evaluate expr and print the result
?*EXPLAIN    -  Explain the current state
?*QUIT       -  Abandon the search
?*HELP       -  Print brief instructions
```

Any other input is taken as a command to proceed. E, EXPLAIN and HELP leave the system in the MONITOR pause. EXPLAIN may be followed by qualifiers to specify the mode of explanation (see the next chapter).

9.2 THE PURR FACILITY

It is often desirable to be able to see in some direct way that the deduction process is taking place, without necessarily slowing it down to the extent that the MONITOR facility entails. The command (PURR ON) enables just such a facility, the PURR facility. The PURR facility consists of a running display accompanying the deduction process. It involves the printing of a few single characters per cycle. No line feed is given after printing (except at the physical end of a line) so that the characters form a continuous string. The meaning of each character is as follows:

Character	Meaning
- (hyphen)	Start of a new cycle
P	Selected constraint a success
U	Selected predication is NIL (false)
R	Resolvents of selected predication obtained
X	Selected predication failed for lack of resolvents
C	A continuation popped up
L	Selected predication failed due to window limit

The PURR facility is disabled by the command (PURR OFF). Thus with the PURR facility on the following transaction would occur:

```
*(ALL (x y) (Age x 1920 y))
-R-R-R-P-R-P
((Turing 8.) (Herbrand 12.))
*
```

The "PURR string" shows that the deduction took six cycles, invoked four procedures and found two answer environments.

CHAPTER 10

EXPLAINING DEDUCTIONS

Once a deduction has been completed and its answer list obtained, one may call for an explanation of the reasoning by which some or all of the answers were deduced. For instance, the following transaction consists of first constructing the answer list for the query (ALL (x y) (Age x 1920 y)) and then requesting an explanation for the second item.

```
*(ALL (x y) (Age x 1920 y))

((Turing 8.) (Herbrand 12.))
*(EXPLAIN 2)

To show:
((Age x 1920. y))

it is enough, by
(¦- AGE-RULE (Age x 1920. y)
               <- (Born x day:1 month:1 birth-year:1)
                & (= y (- 1920. birth-year:1)))

to show:
((Born x day:1 month:1)(= y (- 1920. birth-year:1)))

then it is enough, by
(¦- HERBRAND1 (Born Herbrand 12. February 1908.))

to show:
((= y 12.))

then it is enough, by
(¦- REFLEXIVE-LAW (= Reflexive Law))

to show:
NIL

(End of explanation)
```

The (EXPLAIN 2) command causes an explanation of the answer (Herbrand 12.) to be printed. The successive appearances of the query component of the active proof states leading to the answer are exhibited, and the procedure clause activated to cause the transition is shown. The query component of each state is shown with respect to the environment component of that state. The activated procedure clause is shown with respect to the environment of the resulting state (i.e. after the activation has extended the environment). Various further inflections are provided with the EXPLAIN command. (EXPLAIN ALL) provides explanations of all answers. (EXPLAIN N1 ... Nk) provides explanations of the N1st, ..., Nk'th answers. (EXPLAIN) is the same as (EXPLAIN 1).

Explanations can be produced only when the history facility is enabled, which normally it is not. The history facility is enabled by (HISTORIES ON), disabled by (HISTORIES OFF). Enabling the history facility can impose significant overhead on the system, particularly when the deduction tree must be searched to great depth.

The answers which one can have explained are those produced by the most recently completed invocation of ALL, ANY, THE or SETOF. If there are no such answers EXPLAIN will simply respond "(Nothing to explain)". An attempt to select a non-existent answer will be ignored, except that a note to that effect is typed.

10.1 ALTERNATIVE EXPLANATION MODES.

The EXPLAIN facility is considerably more flexible than indicated by the example just discussed, which only illustrates the normal mode of explanation. One can obtain explanations in a variety of styles. The variations are specified by typing qualifiers in the command following the selection of the answers to be explained. To illustrate, the command (EXPLAIN 2 NAMES FINAL) would print a similar sort of explanation, except that only the names of the assertions would be printed, and the constraints would all be shown in the solution environment.

10.1.1 Specifying Items To Be Included.

Besides constraints and assertions, one may also instruct the system to print answer templates at each stage of the explanation, instantiated and simplified. One may also print names of assertions rather than printing assertions in full.

When names of assertions are to be printed the system will construct names for assertions for which the user has not specified names. These "manufactured" names have the form (Pred k), where "Pred" is the principal predicate symbol of the assertion and the integer k gives the sequence number of the assertion in the list produced by PRINTFACTS. User-supplied names are usually taken just as specified, but one can request "long" names, in which case the name given by the user is combined with the principal predicate symbol to form a list "(Pred Name)". Manufactured names are always in the long format.

The qualifiers which control all this are the following:

ASSERTIONS	Print assertions in full	[Default]
NAMES	Print names of assertions	
UNNAMED	Print assertions which lack user-supplied names, print names where available	
LONG	Print all names in long format	
SHORT	Print user-supplied names in short format	[Default]
CONSTRAINTS	Print constraints	[Default]
NOCONSTRAINTS	Omit constraints	
ANSWERS	Print answer templates	
NOANSWERS	Omit answer templates	[Default]
CONTINUATIONS	Print continuations with constraints	
NOCONTINUATIONS	Omit continuations	[Default]

If NOASSERTIONS is specified the format of the explanation is adjusted accordingly. If NOASSERTIONS, NOANSWERS and NAMES are all specified the explanation is simply a list of the names of the assertions used, with no ornamentation. The default selection between CONTINUATIONS and NOCONTINUATIONS can be changed by (CONTINUATIONS ON) or (CONTINUATIONS OFF).

10.1.2 Specifying Environments To Be Used.

We remarked earlier that the normal explanation shows each step of the derivation in the environment current at that step. One can, however, specify other choices as follows:

INITIAL	Use initial (empty) environment	
CURRENT	Use current environment	[Default]
FINAL	Use final (solution) environment	

When the INITIAL environment is specified constraints are shown in the current environment, as nothing earlier makes any sense, while assertions are shown in the form in which they appear in the knowledge base. Note that the ANSWERS option is useful only in conjunction with CURRENT, though other combinations are allowed.

Anything other than a qualifier appearing in the command will be ignored, with a warning message to that effect typed to the user.

10.2 OBTAINING EXPLANATIONS IN LISP.

The system contains a number of SUBR-type functions which allow the LISP programmer to get at the basic material of the explanations. The programmer can then format explanatory material in whatever way he finds convenient. The first argument to each of these functions is an "answer number", which is the number of the answer to be explained, just as might be typed to EXPLAIN.

(EXPLNAMES ANSNMB)

returns a list of the names, in long format, of the assertions used to derive the answer, in the order used.

(EXPLASSERTIONS ANSNMB ENV)

returns a list of the assertions used to derive the answer, in the order used. Here ENV should be one of the atoms INITIAL, CURRENT, FINAL, to specify the environment in which the assertions will be shown. Each assertions is represented by a list

(Pred Name/Number Head Tl ... Tl)

where "Pred" is the principal predicate symbol, "Name/Number" is the user-supplied name or system-manufactured number, and the remaining entries are the predications of the assertion.

(EXPLCONSTRAINTS ANSNMB ENV CONTNS)

returns a list of the constraints arising in the derivation, beginning with the original query and ending with NIL. Here ENV specifies the environment as before, except that INITIAL is treated the same as CURRENT. CONTNS should be T if continuations are desired, NIL otherwise. The entries of the list returned by EXPLCONSTRAINTS are themselves lists of some complexity. If the constraint in question has no continuation, the corresponding entry has the form:

((q1 ... qN))

where qi is a predication. If the constraint has a continuation, but CONTN is NIL, the entry will have the form

```
((q1 ... qN) CONTINUED)
```

while if the constraint has a continuation and CONTN is T, the entry has the form

```
((q1 ... qN) (p1 ... pM) ...)
```

where pi is a predication of the continuation, which may itself be followed by another continuation, and so on.

```
(EXPLTEMPLATES ANSNMB)
```

returns a list of answer templates shown in the successive CURRENT environments, beginning with the original template and ending with the actual answer.

All of these functions follow a common convention regarding exceptions. If the answer number specified does not correspond to an existing derivation the result is the atom NO-EXPLANATION. If the most recent search was performed with the history facility disabled the result is NIL.

CHAPTER 11

INTERACTING WITH LOGLISP

In the present chapter we discuss the mechanics of running LOGLISP, obtaining information, controlling the operating modes and default settings, and some points dealing with errors. Before doing so we emphasize one convention:

> RESERVED IDENTIFIERS
>
> Identifiers beginning with the character "#" are reserved for use by the system. Users should generally avoid such identifiers. Under no circumstances should a user assign a value to such an identifier.

11.1 RUNNING LOGLISP.

We suppose that the user has logged in and obtained access to the disk area containing the LOGLISP system. The precise method for doing so will vary from one installation to another, but should be explained in a note accompanying this guide.

To run the LOGLISP system simply type the monitor "command"

```
./LOGLSP core
```

where "core" is an optional core argument in the form one would give for the RUN command. If the core argument is omitted the system will have a rather small working area. A good medium allocation is 60K. For large programs one may wish to specify the LISP storage allocations. To do so, use a command like

```
./LOGLSP 140,10000 1000 1000 1000
```

in which the core argument is followed by a comma and the LISP allocations, separated by spaces. The order of these is FULL WORD SPACE, BINARY PROGRAM SPACE, REGULAR PDL, SPECIAL PDL, with the allocations being interpreted in octal, just as in LISP.

LOGLISP will prompt with "*" when ready, being at the top level of LISP. At this point one may enter assertions, queries, and the like as described in the earlier chapters. The system includes the optional numeric functions (SQRT, SIN, SIND, etc.) loaded in expanded core.

11.1.1 An Alternate Method.

The method of running LOGLISP to be described now is useful only for those who desire non-standard initialization or wish to minimize the core requirements of the system. One may also run the system using the monitor command

```
.RUN LOGLSP core
```

in which case LISP will ask for allocations. The system will be run uninitialized and without tne extra numeric functions. Before assertons can be entered or queries processed the system must be initialized using START or SYSINIT (see below).

11.2 INITIALIZATION.

When run in the ususal way the system starts out properly initialized with an empty knowledge base. One may re-initialize the LOGIC part of the system at any time by invoking the function START.

```
(START)
```

leaves an empty knowledge base and resets the operating mode controls and system defaults to their standard values. LISP function definitions, file descriptions, and identifier values are not changed, except for those values which are used in system control.

An alternate initialization function is available for those with special requirements.

```
(SYSINIT ENVL HEAPL)
```

initializes with the specified limits. ENVL is the maximum number of rules allowed in any one deduction. HEAPL is the maximum number of nodes which can be WAITING at any instant in a heuristically guided search. On re-initialization SYSINIT will retain a previous limit which is larger than the one specified, since there is nothing to be gained by reducing one of the limits. START simply performs (SYSINIT 500 300).

The limit values determine the sizes of certain arrays allocated in binary program space. The storage consumed by these is approximately

(1/2)ENVL + HEAPL

words. If the system is re-initialized with increased limits the storage occupied by the old arrays is lost, as binary program space is not garbage-collected.

11.3 INFORMATION.

When prompted for input at any of the main interaction points the user can obtain brief instructions by simply typing HELP. Assistance is thus available at the top level of LISP, in the montior pause, and in FACTS, as well as when the deduction machinery asks for instructions (see below). HELP is not available while editing, but the editor is just the standard LISP editor, so no special difficulties should be encountered.

Abbreviated instructions for using any of the LOGIC interface functions (¦-, THE, ALL, ANY, etc.) can be obtained by invoking the command (DOC fn), where "fn" is the name of the function in question. These instructions were developed using the on-line documentation package described in [???]. The documentation package itself is included in LOGLISP for the convenience of users.

11.4 CONTROL.

The earlier chapters of this report mention a number of functions used to control various operating modes, as well as several defaults used by the system. In this section we shall summarize the control functions and explain the treatment of defaults in somewhat greater detail.

11.4.1 Control Functions.

All of the control functions take one argument, which should be ON or OFF (T or NIL may be used as well), and return the argument after altering the system state appropriately. These functions will, however, evaluate a non-atomic argument expression, so that calls of the functions may be nested. To illustrate, the command (PURR (MONITOR ON)) enable both the PURR facility and the MONITOR facility.

Several of these functions operate simply by setting the value of a LISP identifier, in which case NIL represents OFF, while anything else represents ON. The identifiers so used may changed directly by LISP programs, or accessed by them as may seem useful. The table which follows lists the names of the control functions, the initial settings, and, where applicable, the identifier set by the function.

Function	Initial Setting	Identifier
PURR	OFF	*PURR
MONITOR	OFF	*MONITOR
CONTINUATIONS	OFF	*CONTINUATIONS
HISTORIES	OFF	*HISTORIES
ASK	ON	*ASK
AUTO=	ON	[None]
AUTOAND	ON	[None]
AUTO-OR	ON	[None]
AUTOCOND	ON	[None]
PROLOG	OFF	[None]

The facility controlled by ASK is described below in the discussion of errors.

11.4.2 Defaults.

Both in specifying search windows and in requesting explanations the user normally relies on a good many defaults. These are not, in fact, determined rigidly by the sytem, but may be adjusted by the user. The standard default settings are, however, restored by (START).

11.4.2.1 Search Window Defaults - The defaults for search window are the values of the LISP identifiers listed below, along with their initial values.

Identifier	Initial Value
*TREESIZE	INF
*NODESIZE	INF
*ASSERTIONS	INF
*RULES	500
*DATA	INF

Each of these gives the default value for the corresponding window limit. Though 500 is the normal value for *RULES, a different initial value will be used if the system has been specially initialized using SYSINIT. The implementation constraint on the number of rules in a single deduction will be rigorously enforced, even if *RULES is made larger than this limit.

The values which one may assign to these identifiers are the atom INF or any non-negative integer.

11.4.2.2 EXPLAIN Defaults - The default qualifiers for EXPLAIN are similarly controlled by a collection of LISP identifiers. The table below shows the identifiers, the set of values each is allowed to take, and the initial value.

Identifier	Value Set	Initial Value
*ASSERTIONS	{ALL, SOME, NIL}	ALL
*CONSTRAINTS	{T, NIL}	T
*LONGNAMES	{T, NIL}	NIL
*ANSWERS	{T, NIL}	NIL
*CONTINUATIONS	{T, NIL}	NIL
*ENVIRONMENT	{FINAL, CURRENT, INITIAL}	CURRENT

Note that *CONTINUATIONS is controlled by the function CONTINUATIONS, and affects the monitoring facility as well as EXPLAIN.

11.5 ERRORS.

Errors can arise either in LOGIC or in LISP, though aside from minor syntax errors LOGIC is more likely to fail than to detect an error.

11.5.1 LISP Errors.

Errors detected by LISP will result in entry to the LISP break package in the usual way. If the error arose during simplification a backtrace will show none of the workings of the reduction machinery, which is probably the best course the system could take. All of the LISP facilities for recovery and analysis are available.

Note that misspelled function names in LOGIC terms will not lead to undefined function errors, simply to expressions which are not evaluable.

11.5.2 LOGIC Errors.

Earlier chapters explained how syntax errors are handled by |- and FACTS. There is one other type of error which can be detected by LOGIC -- the undefined predicate error.

A predicate is considered to be undefined if it has neither a LISP definition (as a function) nor a LOGIC definition (as a procedure of one or more assertions). If such a predicate is encountered during a search, and if the ASK facility is enabled (as it is initially), the system will ask the user for instructions, after first printing a message specifying the undefined predicate.

The prompt for instructions is "ASK*". Responses are as follows:

ASK*;	Continue search
ASK*F	Execute FACTS
ASK*S	Correct spelling automatically, if possible
ASK*S pred	Correct spelling to pred
ASK*E expr	Evaluate expr and print the result
ASK*P	Print the current constraints (as when monitoring)
ASK*HELP	Print instructions

Anything other than ";" causes the system to remain in the ASK state. If the user does anything which might conceivably alter matters, the system will try again to aimplify and obtain resolvents.

The automatic spelling correction attempts to find a predicate (defined by LOGIC) which closely matches the undefined predicate. If successful it informs the user of the chosen predicate, if not successful it informs the user of that fact. Spelling corrections are accomplished with RPLACA, so the effect may reach beyond the immediate situation. When the undefined predicate occurs as an instance of some variable, spelling corrections are probably unwise, and the user is warned of such circumstances.

11.5.3 WAITING Heap Overflow.

If the heap used to store WAITING nodes is full and a new node needs to be entered, the system will discard the new node and print a message to that effect. No provision is made for user intervention upon such an occurrence. The only recourse is to re-initialize with a larger allocation (see SYSINIT).

11.5.4 Exhaustion Of Free Storage.

If free storage is exhausted during a search there may be a great many nodes WAITING to be processed, and there may not be enough storage to start another search until these nodes are erased. To do so, one may give the command (#INITHEAP 1). This invokes an internal function which accomplishes the desired result, setting the search mode to (HEURISTIC 1 1) as it does so. Afterwards it may help to run with (HISTORIES OFF).

11.6 ADDITIONAL LISP FUNCTIONS.

LOGLISP includes a number of functions not provided by standard LISP. Some of these have been mentioned earlier.

11.6.1 Short Names For Arithmetic.

The short arithmetic operators are as follows:

(+ e1 ... eN)	[MACRO]
(- e1 ... eN)	[MACRO]
(* e1 ... eN)	[MACRO]
(% e1 ... eN)	[MACRO]

These are the same as PLUS, DIFFERENCE, TIMES, QUOTIENT, except for being more defined. (+) = (-) = 0, while (*) = (%) = 1.

11.6.2 Arithmetic Relations.

The following arithmetic relations are provided, in addition to those included in LISP:

(< e1 e2)	[SUBR]
(<= e1 e2)	[SUBR]
(>= e1 e2)	[SUBR]
(> e1 e2)	[SUBR]

Of course "=" is defined on numbers as well as other objects.

11.6.3 Miscellaneous Arithmetic.

Two other special arithmetic functions are provided:

(** X N) [SUBR]

returns X**N for integer N.

(ODD N) [SUBR]

returns T if the integer N is odd, NIL otherwise.

11.6.4 LOGLISP Utilities.

Some of the LOGLISP system utility functions may be of use to programmers. The names of these functions are not reserved.

(VARIABLE e) [SUBR]

returns T if e is a LOGIC variable, NIL otherwise. In an assertion one might write (VARIABLE (LISP x)) to determine whether the instantiation of x is or is not a variable.

(CONSM e1 ... eN-1 eN) [MACRO]

returns the object (v1 ... vN-1 . vN), where vi denotes the value of ei.

(XFERPROP "DST" "SRC" "KEY") [FSUBR]

makes property KEY of SRC also be property KEY of DST. The property value is not copied.

CHAPTER 12

EXAMPLES OF APPLICATIONS OF LOGLISP

Applications of logic programming are described by [Kowalski 1979], [Clark 1979], [van Emden 1977], [Colmerauer 1973], and [Warren 1977], to name only the principal references.

In this chapter we describe two non-trivial examples of logic programming in which the special features of LOGLISP are exploited.

12.1 PLACES - AN "INTELLIGENT" DATABASE.

PLACES is a knowledge base containing several thousand assertions most of which are *data*, i.e., unconditional ground assertions.

Some representative data of PLACES are shown in Figure 1.

```
(POPULATION BURMA 32200000) <-
(LATITUDE WARSAW 52.25) <-
(LONGITUDE PYONG-YANG -125.8) <-
(ADJOINS LAOS VIETNAM) <-
(COUNTRY VIENNA AUSTRIA) <-
(PRODUCES USSR OIL 491.0 1975) <-
(BELONGS IRAN OPEC) <-
(REGION ISRAEL MIDDLE-EAST) <-
(AREA ETHIOPIA 471778) <-
(GNP-PER-CAPITA NEW-ZEALAND 4250) <-
(OPEN-WATER BALTIC-SEA) <-
(NARROW DARDANELLES) <-
```

Figure 1.

For each predicate appearing in Figure 1, PLACES has a collection of such unconditional ground assertions - a *data procedure* . All these data procedures are comprehensive (they average several hundred assertions each) and some are in a sense complete.

The procedures POPULATION, AREA, REGION, GNP-PER-CAPITA are complete in the sense that every country in the world is covered.

The GNP-PER-CAPITA procedure gives (in US dollars) the gnp-per-capita for each country in the world for a particular year (1976).

The procedure ADJOINS provides data for a procedure BORDERS, which is a pair of rules:

```
(BORDERS x y) <- (ADJOINS x y)
(BORDERS x y) <- (ADJOINS y x)
```

which give PLACES the ability to determine which countries (or bodies of open water) border upon which others. Since ADJOINS is a symmetric relation we need not assert it in both directions, and BORDERS uses ADJOINS accordingly.

The procedure PRODUCES gives (in millions of metric tons) the quantities of various basic commodities (oil, steel, wheat, rice) produced by most of the world's countries in two particular years (1970 and 1975). This procedure could well have covered more years and more commodities, but for the purposes of an example a few hundred assertions seemed enough to illustrate the possibilities.

While the countries of the world form (at any given time) a rather definite set, it is less clear what are the bodies of water which should be named and treated as entities in a database such as PLACES. We took the arbitrary course of naming those bodies of water found on the maps of various parts of the world in the Rand McNally Cosmopolitan World Atlas. We ignored those bodies of water which seemed too small to be of much significance but we strove for some sort of comprehensive description of the boundary of each country. For example, the query

```
(ALL x (BORDERS x IRAN))
```

gets the answer

```
(STRAITS-OF-HORMUZ GULF-OF-OMAN TURKEY USSR PAKISTAN IRAQ
 CASPIAN-SEA AFGHANISTAN PERSIAN-GULF)
```

in which each of the bodies of water STRAITS-OF-HORMUZ, GULF-OF-OMAN, CASPIAN-SEA and PERSIAN-GULF is listed as having a portion of its boundary in common with that of the country IRAN.

12.1.1 RULES.

PLACES contains, in addition to these large "data procedures", a number of rules defining predicates useful in formulating queries.

For example there is a procedure DISTANCE, which consists of the following four rules:

```
   (DISTANCE (POSITION la1 lo1) (POSITION la2 lo2) d)

   <- (= d (SPHDST la1 lo1 la2 lo2))


   (DISTANCE (POSITION la1 lo1) (PLACE q) d)

   <- (LATITUDE q la2)
    & (LONGITUDE q lo2)
    & (= d (SPHDST la1 lo1 la2 lo2))


   (DISTANCE (PLACE p) (POSITION la2 lo2) d)

   <- (LATITUDE p la1)
    & (LONGITUDE p lo1)
    & (= d (SPHDST la1 lo1 la2 lo2))


   (DISTANCE (PLACE p) (PLACE q) d)

   <- (LATITUDE p la1)
    & (LATITUDE q la2)
    & (LONGITUDE p lo1)
    & (LONGITUDE q lo2)
    & (= d (SPHDST la1 lo1 la2 lo2))
```

This procedure can be used to obtain the great-circle distance between any two cities whose latitudes and longitudes are in the data tables, or between one such city and an arbitrary position on the earth's surface (given by its latitude and longitude) or between two such arbitrary positions.

The procedure DISTANCE illustrates the ability to call user-defined LISP functions by forming constructions using their names as operators. The LISP function SPHDST returns the great circle distance (in nautical miles) between any two points on the earth's surface (given by their respective latitudes and longitudes).

Thus the query:

```
(THE d (DISTANCE (PLACE SAN-FRANCISCO)(PLACE OSLO) d))
```

gets the answer:

```
5197.5394
```

There is a rule which serves to define the predicate LANDLOCKED. Intuitively, a country or body of water is landlocked if it borders upon only land. The PLACES rule which formalizes this meaning is

```
(LANDLOCKED x)
<- (IS-COUNTRY x)
 & (NULL (ANY 1 T (BORDERS x z)(OPEN-WATER z)))
```

This rule contains two features worthy of comment.

The predicate IS-COUNTRY, defined by the rule

```
(IS-COUNTRY x)
<- (COND ((VARIABLE (LISP x))(COUNTRY y x))
         ((ANY 1 T (COUNTRY z x))))
```

shows how one can use to advantage the LISP conditional form within a LOGIC predication. The effect of the conditional is to avoid redundancy in proving that a given country is a country - by finding all the various cities in it - via a check to see if the argument x is a variable or not. If it is not, then we need find only one datum from the COUNTRY data procedure which has the given country as its second argument.

The second thing worth noting about the rule for LANDLOCKED is the embedded deduction. The list returned by the call

```
(ANY 1 T (BORDERS x z)(OPEN-WATER z))
```

will be empty if and only if x is landlocked.

A similarly structured rule defines the predicate DOMINATES. We wish to say that a country x dominates a "narrow" waterway y if x borders y but no other country does. Thus:

```
(DOMINATES x y)
<- (NARROW y)
 & (IS-COUNTRY x)
 & (BORDERS x y)
 & (NULL (ANY 1 T (BORDERS y w)
                  (NOT (OPEN-WATER w))
                  (NOT (= x w))))
```

12.1.2 NEGATION AS FAILURE.

The use of the predicate NOT in the precedure DOMINATES raises an interesting general point.

NOT is of course a LISP-defined notion and will therefore receive appropriate treatment during the deduction cycle in the manner explained in Chapter 3.

However, it is possible to include in one's knowledge base the rule

```
(NOT p) <- (NULL (ANY 1 T p))
```

which is known as the "negation as failure" rule. PLACES has the negation as failure rule as one of its assertions. The effect of its presence in a knowledge base is to declare that the knowledge base is complete - that inability to deduce p is to be treated as tantamount to the ability to deduce the negation of p.

The version of the negation as failure rule shown above is undiscriminating as between the various predications - it is in effect the declaration that all of the data procedures are complete and that all of the general procedures are "definitions" of their predicates. It would be possible to assert more specialised negation as failure rules, which declare that the knowledge base is complete with respect to a particular predication-pattern. For example, we might assert

```
(NOT (BELONGS x y)) <- (NULL (ANY 1 T (BELONGS x y)))
```

in order to declare that BELONGS is complete, even though we are not willing to assert the negation as failure rule for all predications p. In general, one would expect that users of LOGLISP would wish to be selective in their appeal to negation as failure, in just this fashion. These data and rules are invoked by the following queries, which illustrate some of the possibilities.

12.1.3 Some Sample Queries For PLACES.

The following examples consist of some specimen queries which one can make of PLACES, together with the answers that they get. In each case we first state the query in ordinary English, and then restate it in formal LOGLISP.

We are not claiming that there is a uniform procedure, known to us, by which one may translate queries from English to LOGLISP in this manner. At present, in order to express queries (and indeed, assertions) in LOGLISP, one must know the language and be able to express one's intentions in it. In this respect LOGLISP is like any other programming language. It is in fact quite easy to learn enough LOGLISP to construct and operate one's own "intelligent database" in the style of PLACES.

Query 1.

What are the oil production figures for the
non-Arab OPEC countries in the year 1975?

```
(ALL (x y)
     (BELONGS x OPEC)
     (NOT (BELONGS x ARAB-LEAGUE))
     (PRODUCES x OIL y 1975.))
```

Answer 1.

```
((IRAN 267.59999) (NIGERIA 88.399991)
                  (VENEZUELA 122.19999)
                  (INDONESIA 64.100000)
                  (ECUADOR 8.2000000))
```

This answer is shown just as the LISP "prettyprint" command SPRINT types it out. It is of course possible to dress up one's output in any way one pleases. Note that ALL returns a <u>list</u> of (in this case) tuples.

Query 2.

Of all the countries which are poorer than Turkey,
which two produced the most steel in the year 1975?
How much steel was that? What are the populations
of those countries?

AD-A096 042 SYRACUSE UNIV NY SCHOOL OF COMPUTER AND INFORMATION --ETC F/G 9/2
LOGIC PROGRAMMING IN LISP.(U)
JAN 81 J A ROBINSON, E E SIBERT F30602-77-C-0235
UNCLASSIFIED RADC-TR-80-379-VOL-1 NL

2 OF 2
AD
A096042

END
DATE
FILMED
4-81
DTIC

```
(FIRST 2.
       (QUICKSORT
        (ALL (x y w)
             (GNP-PER-CAPITA TURKEY v)
             (GNP-PER-CAPITA x u)
             (LESSP u v)
             (PRODUCES x STEEL y 1975.)
             (POPULATION x w))
        (DECREASING)
        2.))
```

Answer 2.

```
((CHINA 29.0 880000000.) (INDIA 7.8999999 643000000.))
```

This example illustrates the fact that ALL (like ANY, THE, and SETOF) returns a LISP data-object which can be handed as an argument to a LISP function. In this case QUICKSORT and FIRST are user-defined LISP functions which were created in order to serve as useful tools in posing inquiries to PLACES.

(QUICKSORT list relation k) returns the given list of tuples ordered on the kth component with respect to the given relation. (FIRST n list) returns the (list of the) first n components of the given list. (DECREASING) returns the LISP relation GREATERP (and we also have (INCREASING), which returns the relation LESSP, and (ALPHABETICALLY), which returns the relation LEXORDER).

Query 3.

Which of France's neighbors produced most wheat (in metric tons) per capita in the year 1975? How much wheat per capita was that?

```
(EARLIEST
 (ALL (x y)
      (BORDERS x FRANCE)
      (PRODUCES x WHEAT z 1975.)
      (POPULATION x u)
      (= y (QUOTIENT (TIMES z 1000000.) u)))
 (DECREASING)
 2.)
```

Answer 3.

```
(ITALY 0.16956329)
```

(EARLIEST *list* *relation* k) returns the first tuple in *list* after it has been re-ordered on the *k*th component of each of its tuples with respect to the given *relation*. Note that arithmetical terms formed with LISP's arithmetic operations are evaluated by the simplification step of the deduction cycle, as explained in Chapter 3.

Query 4.

Which of the NATO countries is landlocked?

```
(ALL x (BELONGS x NATO) (LANDLOCKED x))
```

Answer 4.

```
(LUXEMBOURG)
```

Query 5.

Which waterway is dominated by Panama?

```
(THE x (DOMINATES PANAMA x))
```

Answer 5.

```
PANAMA-CANAL
```

Note that THE returns PANAMA-CANAL and not (PANAMA-CANAL).

Query 6.

Describe the boundary of the USSR by giving all its neighbors in alphabetical order.

```
(ORDER (ALL x (BORDERS x USSR)) (ALPHABETICALLY))
```

Answer 6.

```
(AFGHANISTAN ARCTIC-OCEAN BALTIC-SEA BERING-SEA BLACK-SEA
 BULGARIA CHINA FINLAND HUNGARY IRAN MONGOLIA NORWAY
 POLAND ROMANIA TURKEY)
```

(ORDER <u>list</u> <u>relation</u>) returns the given <u>list</u> after ordering it with respect to the given <u>relation</u>.

Query 7.

Are there any landlocked countries in the Far East? If so, give an example.

```
(ANY 1. x (REGION x FAR-EAST) (LANDLOCKED x))
```

Answer 7.

```
(MONGOLIA)
```

Query 8.

Is there an African country which dominates an international waterway? Which country? Which waterway?

```
(ANY 1. (x y) (REGION x AFRICA) (DOMINATES x y))
```

Answer 8.

```
((EGYPT SUEZ-CANAL))
```

Query 9.

What is the average distance from London of cities in countries which have a Mediterranean coastline and which are no more densely populated than Ireland? List those countries, together with their population densities, from least crowded to most crowded.

```
(PROGN (SETQ COUNTRIES-AND-DENSITIES
             (QUICKSORT
              (ALL (x x-density)
                   (POPULATION IRELAND irish-population)
                   (AREA IRELAND irish-area)
                   (= irish-density
                      (% irish-population irish-area))
                   (BORDERS x MEDITERRANEAN-SEA)
                   (NOT (OPEN-WATER x))
                   (POPULATION x x-population)
                   (AREA x x-area)
                   (= x-density (% x-population x-area))
                   (NOT (> x-density irish-density)))
              (INCREASING)
              2.))
       (SETQ AVERAGE-DISTANCE
             (AVERAGE
              (ALL distance
                   (MEMBER pair
                           (EVAL COUNTRIES-AND-DENSITIES))
                   (= country (CAR pair))
                   (COUNTRY city country)
                   (DISTANCE (PLACE city)
                             (PLACE LONDON)
                             distance))))
       (GIVE AVERAGE-DISTANCE)
       (GIVE COUNTRIES-AND-DENSITIES)
       (QUOTE *))
```

Answer 9.

```
  AVERAGE-DISTANCE is

    1491.1892

  COUNTRIES-AND-DENSITIES is
```

```
((LIBYA 3.)
 (ALGERIA 20.)
 (ALBANIA 24.)
 (TUNISIA 101.)
 (EGYPT 102.)
 (MOROCCO 108.))
*
```

This example shows at somewhat more length what a LISP programmer might make of an inquiry which calls for a more involved investigation. Assignment to the LISP variable COUNTRIES-AND-DENSITIES of the answer to one LOGIC call for later use within another (as well as for output) illustrates one more way in which the LOGLISP programmer can fruitfully exploit the interface between LOGIC and LISP. GIVE is just a dressed-up PRINT command which not only prints the value of its argument expression but also prints the expression.

12.2 A COMPILER.

We shall now present a compiler for a subset of PASCAL. The compiler parses the "source" program, checks types, and generates "object" code which can be executed by LISP (with a few "run-time" utility functions). In order to keep the example small we have confined ourselves to a few statement forms, provided only the types INTEGER and BOOLEAN, with no data structures, and made no provision for declarations, simply incorporating a handful of variable identifiers directly into the language. There are no procedures, no functions, no labels, no jumps. Expressions are treated rather fully, however, given the other limitations.

Even though the language is quite limited, we feel that the example is sufficient to show that we can easily write compilers which, though slow, are entirely adequate for experiments in language design. We point out that the compiler is readily modified to produce an abstract representation of the program, rather than an executable form, as could be used for program analysis or verification.

12.2.1 Organization Of The Compiler.

The "source" program will be represented as a list of tokens, which are simply LISP atoms denoting reserved words, identifiers, constants, operator symbols, and the like. An example is

```
(BEGIN K := K - 1 ; Y := Y * Z END)
```

which will be QUOTEd when it appears as an expression in LOGIC.

Such lists read nicely enough, and the lexical analyzer required to produce such a list from a character string or text file is easily written.

Corresponding to each syntactic category (nonterminal) of the language we introduce a relation which "compiles" phrases of that category. For example, the relation for the category <statement> has the form (STATEMENT tokens rep rest), where 'tokens' is a list of tokens, as above, 'rep' is the object representation of the statement which begins 'tokens', if there is one, and 'rest' is the token list obtained by removing the initial statement from 'tokens'. We do recursive descent parsing, working from left to right without backtracking.

In some cases we wish to "parametrize" categories. The relation for <expression>, for example, has the form (EXPRESSION type tokens rep rest), 'type' being the result type of the expression. On some calls the procedure will be used to discover the type of the expression which begins 'tokens', while on others it will check that the expression in question has the proper type.

To see how this works, consider the assertion for the WHILE statement, which is

```
(|- (STATEMENT (CONS WHILE tl)
               (PROG NIL LOOP: (COND (l s (GO LOOP:))))
               c)
    <- (PARSE tl ((EXPRESSION BOOLEAN l) DO (STATEMENT s)) c))
```

The rule applies only to non-empty lists which begin with WHILE. Using CONS expressions to unify with lists in this fashion we avoid explicit tests for empty lists, but there is no possibility that we will attempt to take the CAR or CDR of an atom. The "object" representation is a PROG construct incorporating the components of the WHILE in an obvious way.

Recall that the syntax for the WHILE statment is

WHILE <Boolean expression> DO <statement> .

Although we could express this directly in terms of EXPRESSION and STATEMENT, it is more convenient to use the auxiliary relation PARSE. PARSE has the form (PARSE tokens items rest). The arguments 'tokens' and 'rest' are used as before, but 'items' is a list of expressions (itself an expression) which defines a sequence of items to be parsed. Each item has one of the forms: token, (syncat var), (syncat parm var). An item of the form 'token' simply specifies that the indicated token should be found. The form (syncat var) specifies that a phrase of the category 'syncat' should be found, its representation to be denoted by 'var'. The form (syncat parm var) is similar, except that 'syncat' is to be parameterized with 'parm'.

The assertions defining PARSE are

```
(|- (PARSE x NIL x))

(|- (PARSE x (hd . tl) c)
    <- (COND ((ATOM (LISP hd)) (= x (CONS hd tx)))
             ((= hd (syncat var)) (syncat x var tx))
             ((= hd (syncat parm var)) (syncat parm x var tx)))
     & (PARSE tx tl c))
```

Observe the use of the variable "tl" to deal with the unpredictable expression 'entries'. The expression (ATOM (LISP hd)) is always evaluable, having the value T just when 'hd' is a token.

12.2.2 The Compiler.

At this point we shall list the compiler, including the interactive documentation which has been provided for the logic procedures. Following the listing we remark further upon the techniques used. The variables "built in" to the compiler correspond to the declarations

```
var  I,J,K,X,Y,Z:INTEGER;
         B,P,Q,R:BOOLEAN;


(DEFPROP STATEMENT
 ((STATEMENT tokens rep rest))
DOC)

(PROCEDURE STATEMENT ONERES)

(|- (STATEMENT (CONS IF tl) (COND (1 s1) . s) c)
    <- (PARSE tl ((EXPRESSION BOOLEAN 1) THEN (STATEMENT s1)) tx)
     & (COND ((= tx (CONS ELSE txx))
              (AND (STATEMENT txx s2 c) (= s ((s2)))))
             ((= s NIL) (= c tx))))

(|- (STATEMENT (CONS WHILE tl)
               (PROG NIL LOOP: (COND (1 s (GO LOOP:))))
               c)
    <- (PARSE tl ((EXPRESSION BOOLEAN 1) DO (STATEMENT s)) c))
```

```
(¦- (STATEMENT (CONS BEGIN tl) (PROGN . ss) c)
    <- (PARSE tl ((REPEATO (STATEMENT ;) ss) END) c))

(¦- (STATEMENT (CONS v tx) (:= v e) c)
    <- (VAR-IDENT ty v)
     & (PARSE tx (:= (EXPRESSION ty e)) c))


(DEFPROP EXPRESSION
 ((EXPRESSION type tokens rep rest))
DOC)

(PROCEDURE EXPRESSION)

(¦- (EXPRESSION ty x r c)
    <- (SIMPLE-EXPR ty1 x se cc)
     & (COND ((REL-OPR ty1 cc rel ccc)
              (AND (SIMPLE-EXPR ty1 ccc se2 c)
                   (= ty BOOLEAN)
                   (= r (rel se se2))))
             (T (AND (= ty ty1) (= r se) (= c cc)))))


(DEFPROP REL-OPR
 ((REL-OPR arg-type tokens rep rest))
DOC)

(PROCEDURE REL-OPR ONERES)

(¦- (REL-OPR INTEGER (CONS r c) r c)
    <- (MEMQ r (QUOTE (< <= = >= > <>))))

(¦- (REL-OPR BOOLEAN (CONS r c) fr c)
    <- (= fr
          (SELECTQ r
                   (< B<)
                   (<= B<=)
                   (= =)
                   (>= B>=)
                   (> B>)
                   (<> <>)
                   NIL))
     & (NEQ fr NIL))


(DEFPROP SIMPLE-EXPR
 ((SIMPLE-EXPR type tokens rep rest))
DOC)
```

```
(PROCEDURE SIMPLE-EXPR ONERES)

(¦- (SIMPLE-EXPR INTEGER (CONS + tl) r c)
    <- (TERM INTEGER tl trm cc)
     & (SIMPLE-TAIL (INTEGER trm) cc r c))

(¦- (SIMPLE-EXPR INTEGER (CONS - tl) r c)
    <- (TERM INTEGER tl trm cc)
     & (SIMPLE-TAIL (INTEGER (MINUS trm)) cc r c))

(¦- (SIMPLE-EXPR ty x r c)
    <- (TERM ty x trm cc)
     & (SIMPLE-TAIL (ty trm) cc r c))


(DEFPROP SIMPLE-TAIL
 ((SIMPLE-TAIL (type prev) tokens rep rest))
DOC)

(PROCEDURE SIMPLE-TAIL)

(¦- (SIMPLE-TAIL (ty u) x r c)
    <- (COND ((ADD-OPR ty x opr cc)
              (AND (TERM ty cc trm ccc)
                   (SIMPLE-TAIL (ty (opr u trm)) ccc r c)))
             ((= r u) (= c x))))


(DEFPROP ADD-OPR
 ((ADD-OPR type tokens rep rest))
DOC)

(PROCEDURE ADD-OPR ONERES)

(¦- (ADD-OPR INTEGER (CONS + c) + c))

(¦- (ADD-OPR INTEGER (CONS - c) - c))

(¦- (ADD-OPR BOOLEAN (CONS OR c) OR c))


(DEFPROP TERM
 ((TERM type tokens rep rest))
DOC)

(PROCEDURE TERM)

(¦- (TERM ty x r c) <- (FACTOR ty x f cc)
                     & (TERM-TAIL (ty f) cc r c))
```

```
(DEFPROP TERM-TAIL
 ((TERM-TAIL (type prev) tokens rep rest))
DOC)

(PROCEDURE TERM-TAIL)

(¦- (TERM-TAIL (ty u) x r c)
    <- (COND ((MUL-OPR ty x opr cc)
              (AND (FACTOR ty cc trm ccc)
                   (TERM-TAIL (ty (opr u trm)) ccc r c)))
             ((= r u) (= c x))))


(DEFPROP MUL-OPR
 ((MUL-OPR type tokens rep rest))
DOC)

(PROCEDURE MUL-OPR ONERES)

(¦- (MUL-OPR INTEGER (CONS * c) * c))

(¦- (MUL-OPR INTEGER (CONS DIV c) % c))

(¦- (MUL-OPR INTEGER (CONS MOD c) REMAINDER c))

(¦- (MUL-OPR BOOLEAN (CONS AND c) AND c))


(DEFPROP FACTOR
 ((FACTOR type tokens rep rest))
DOC)

(PROCEDURE FACTOR ONERES)

(¦- (FACTOR BOOLEAN (CONS TRUE c) T c))

(¦- (FACTOR BOOLEAN (CONS FALSE c) NIL c))

(¦- (FACTOR BOOLEAN (CONS ODD tx) (ODD e) c)
    <- (PARSE tx (/( (EXPRESSION INTEGER e) /)) c))

(¦- (FACTOR BOOLEAN (CONS NOT tx) (NOT f) c)
    <- (FACTOR BOOLEAN tx f c))

(¦- (FACTOR ty (CONS /( tx) e c)
    <- (PARSE tx ((EXPRESSION ty e) /)) c))
```

```
(¦- (FACTOR ty (CONS u c) r c)
    <- (COND ((NUMBERP u) (AND (= ty INTEGER) (= r u)))
             ((VAR-IDENT ty u) (= r (VAR u)))))


(DEFPROP VAR-IDENT
 ((VAR-IDENT ty var))
DOC)

(PROCEDURE VAR-IDENT)

(¦- (VAR-IDENT ty v) <- (COND ((MEMQ v (QUOTE (I J K X Y Z)))
                               (= ty INTEGER))
                              ((MEMQ v (QUOTE (B P Q R)))
                               (= ty BOOLEAN))))


(DEFPROP PARSE
 ((PARSE tokens items rest)
  (An item may be a token or (syncat var) or (syncat parm var)))
DOC)

(PROCEDURE PARSE ONERES)

(¦- (PARSE x NIL x))

(¦- (PARSE x (hd . tl) c)
    <- (COND ((ATOM (LISP hd)) (= x (CONS hd tx)))
             ((= hd (syncat var)) (syncat x var tx))
             ((= hd (syncat parm var)) (syncat parm x var tx)))
     & (PARSE tx tl c))


(DEFPROP REPEATO
 ((REPEATO cntrl tokens rep rest)
  (cntrl is (syncat sep) or (syncat parm sep))
  (Yields {<syncat><sep>}*{<syncat>¦ }))
DOC)

(PROCEDURE REPEATO ONERES)

(¦- (REPEATO (syncat sep) x r c)
    <- (COND ((syncat x r1 tx)
              (COND ((= tx (CONS sep txx))
                     (AND (REPEATO (syncat sep) txx rr c)
                          (= r (r1 . rr))))
                    ((= r (r1)) (= c tx))))
             ((= r NIL) (= c x))))
```

```
(¦- (REPEATO (syncat parm sep) x r c)
    <- (COND ((syncat parm x r1 tx)
              (COND ((= tx (CONS sep txx))
                     (AND (REPEATO (syncat parm sep) txx rr c)
                          (= r (r1 . rr))))
                    ((= r (r1)) (= c tx))))
             ((= r NIL) (= c x))))
```

Note that procedures with more than one assertion are specified to be ONERES. That this is appropriate depends upon two circumstances. First, the grammar is unambiguous (we made it that way). Second, we expect that 'tokens' really will be a list of atoms. For the auxiliary procedures PARSE and REPEATO the appropriateness of ONERES follows from the fact that the argument expressions in calls of these procedures are always specified in sufficient detail that only one assertion will apply.

REPEATO, which has the appearance of a parameterized category, handles constructs of the form "zero or more occurrences of 'syncat' (possibly with parameter) separated by 'sep'". The "representation" it produces is an expression having the form of a list of representations, and is used as the tail of some larger expression. The assertion dealing with compound statements illustrates the use of REPEATO.

The treatment of simple expressions (relation SIMPLE-EXPR) is complicated by the necessity of associating unparenthesized expressions to the left. "X + Y - Z", for example, means "(X + Y) - Z". To accomplish this we introduce the auxiliary relation SIMPLE-TAIL, whose parameter includes the representation of the previous portion of the expression being compiled. The object representation of "X + Y - Z" is

```
(- (+ (VAR X) (VAR Y)) (VAR Z))
```

VAR being the run-time function which evaluates variables. TERM is handled similarly.

We have implemented AND and OR using the corresponding LISP functions, which is not entirely proper, as LISP uses "short-cut" evaluation, unlike PASCAL. No real harm results, though, since our restricted language admits no expressions with side effects.

12.2.3 Using The Compiler.

One can use the compiler by simply invoking a query, as

```
*(THE (r c) (STATEMENT '(X := Y * Z + I ;) r c))
```

```
((:= X (+ (* (VAR Y) (VAR Z)) (VAR I)) (QUOTE (;)))
*
```

Note that we can compile a single expression as easily as an entire program -- a useful feature for the language experimenter.

To make matters a little easier we have written some simple LISP programs to manage administrative chores, these being included with the run-time support programs. For our purposes a program is just a statement, followed perhaps by a terminating character such as ".". What follows is an example involving a less trivial program, the fast exponentiation algorithm.

```
*(QPRINTC FASTEXP)

(BEGIN
        Y := 1. ;
        Z := X ;
        K := I ;
        WHILE K > 0. DO
                IF ODD ( K ) THEN
                  BEGIN Y := Y * Z ;  K := K - 1. END
                ELSE
                  BEGIN Z := Z * Z ;  K := K DIV 2. END
  END .)
NIL
*(COMPILE FASTEXP)

COMPILED
*(SPRINT OBJECT 1)

(PROGN (:= Y 1.)
       (:= Z (VAR X))
       (:= K (VAR I))
       (PROG NIL
        LOOP:(COND
              ((> (VAR K) 0.)
               (COND ((ODD (VAR K))
                      (PROGN (:= Y (* (VAR Y) (VAR Z)))
                             (:= K (- (VAR K) 1.))))
                     ((PROGN (:= Z (* (VAR Z) (VAR Z)))
                             (:= K (% (VAR K) 2.)))))
              (GO LOOP:)))))
NIL
*(DEP X 2 I 13)

Deposited
*(RUN OBJECT)
```

```
NIL
*(EXM X I Y Z K)

X 2.
I 13.
Y 8192.
Z 256.
K 0.
***
*
```

The object program which results is left as the value of the identifier OBJECT, which we find to be a LISP version of the algorithm. The function DEP (for DEPosit) is used to preset the necessary variables, RUN executes the object program, and EXM is used afterwards to EXaMine the outcome. Values of program variables are actually stored as PVAL properties of the variable identifiers, thus avoiding any possibility of collision with LISP identifier values, which are VALUE properties.

The techniques used in the run-time system are too primitive to serve for the implementation of more sophisticated languages, particularly those including procedures, but the compiler itself suffers no such defect. One point regarding the compiler bears further discussion. We have chosen, at the expense of some complication in the logic, to write the compiler so as to avoid backtracking. That this is the case is apparent from the use of ONERES. The result is a faster compiler than would be obtained with backtracking, but beyond this, when enlarging the language to encompass declarations we have the option of using imperative techniques for symbol table management. With backtracking, hence "concurrent" exploration of the deduction tree, this option would be lost.

CHAPTER 13

IMPLEMENTATION OF LOGIC IN LISP

13.1 GENERAL CONSIDERATIONS.

The present implementation is written entirely in UCI Lisp as run at Syracuse. It should be possible to transport this implementation to other LISP systems based on LISP 1.6 with only trivial modifications, if any. Our goal throughout has been to devise an implementation which could be extended and adapted as we gained experience with the system, without making fundamental sacrifices in efficiency.

Certain choices regarding the behavior of the overall system have had a major impact on the implementation. One of these choices is the decision that the user should be able to add to or modify his logic procedures at will, much as he adds or modifies LISP function definitions. This decision leads us to adopt a syntax in which variables are always recognizable as such. In particular, no action of the user can cause an expression to be regarded as a variable. The user is not, in fact, given direct access to the internal representations of procedures, states, and the like, but the interfaces provided come pretty close to achieving our goal. Internal representations are not completely hidden, of course, but the "#" convention for system identifiers makes it easy enough for any but the malicious user to avoid entanglement with the system.

A second fundamental choice is that inference-making should not be confined to depth-first search. We are thus able to obtain answers to some queries even in the presence of "depth-first runaway" and also to use heuristic techniques to obtain a few answers more rapidly than might otherwise have been the case. This choice precludes, however, the very economical stack-oriented techniques pioneered by Warren in Edinburgh Prolog [Warren 1977]. We have instead used structure sharing techniques of the sort described by Boyer and Moore [Boyer-Moore 1972], specialized for Horn-clause resolution. The result is not greatly slower than Warren's method, the time needed to access a variable binding being, for practical purposes, bounded by a constant.

At present the search is guided by a very simple heuristic function which is a linear combination of the depth of the state and the number of predications it contains. This function could easily be made more sophisticated. States awaiting selection are kept in a priority queue represented by a heap (in the sense of Floyd) which allows for rapid storage and retrieval.

In order to avoid numerous futile attempts at resolution we have incorporated a secondary indexing scheme in the representation of the procedure base, and we expect to do more such indexing in the future. As now implemented, the secondary indexing applies only to procedures whose clauses all have heads which are ground predications, but it works very well for those procedures. The present indexing is entirely automatic, but future extensions may be able to take into account advice supplied by the user.

The implementation of reduction uses a sort of "conditional evaluation" technique which works directly with implicit representations and is able, generally, to preserve the structure sharing inherent in the Boyer-Moore method, thus avoiding the exponential growth which results from explicit representation of instantiations. Any evaluations which are performed during reduction are carried out by the underlying LISP system.

13.2 THE SYSTEM CODE.

We shall now discuss the code for the system, with particular attention to data representation. The functions which make up the system have been informally organized into "modules", and this presentation follows that organization. The modules are discussed in "bottom-up" order, more or less, although the organization of each module is generally "top-down", except that MACROs appear first, for technical reasons. Initialization functions which are naturally associated with a particular module are placed in such modules. There is also a "system initialization" module for initialization functions of broader scope.

The functions which constitute the system are accompanied by on-line documentation which is managed with the aid of the documentation package described in appendix A. (The documentation package is included in LOGLISP.) The system ordinarily includes documentation for those functions considered to be user interface functions, chiefly those whose names do not begin with "#". On-line documentation for the other functions is provided in auxiliary files, however. Details concerning these matters will be found in the section on system building placed at the end of this chapter.

The file LOGFNX.PRG, distributed with the system, contains a full listing of the system code, including interactive documentation as well as function definitions, and is organized by modules, paralleling the following discussion. A few functions with special requirements are found in files other than LOGFNX.PRG, but these are explained when they arise.

13.2.1 Utilities.

These are a number of small utility functions used at various points in the system. The on-line documentation together with the function definitions should be adequate explanation. The reader should also refer to the file CMPUTL.LSP which contains routines which aid in MACRO definition.

13.2.2 Environments.

The environment in use at a given moment (called the "current environment") is stored in the array #ENV and several global variables. Other environments are stored as list structures. The system keeps a record of which list-environment (if any) is the current environment so as to avoid needless effort converting between the two representations. The representation of the current environment is as follows:

#ENVK - Largest index (subscript) appearing
#ENVS - List environment (if any) from which obtained
#ENV - Array of association lists; indexed 0 :: #ENVLIMIT
#ENVDIFF - T if #ENVS differs from current environment, NIL otherwise

For $0 \leq i \leq$ #ENVK, (#ENV i) is an association list giving bindings for variables with index i. Each entry of such a list has the form (variable index . expr), indicating that the variable is bound to the specified expression viewed at the specified index.

The list representation of the current environment is

```
(#ENVK (#ENV #ENVK) ... (#ENV 0)) .
```

Conversion between list representation and array representation requires time proportional to #ENVK when actually performed, but because of the optimization this is not very significant. On account of the way environments are used, the number of entries in one association list can never exceed the number of distinct variables appearing in the tail of a single assertion. In practice this number hardly ever exceeds six, and the average is usually less.

We emphasize that variables are uniquely stored. Besides the atomic variables, which begin with lower-case letters, the system sometimes constructs explicitly subscripted variables having the form

(#VAR# atomic-variable . subscript).

Both kinds are detected by the function VARIABLE, which is available for public use.

13.2.3 Unification.

An expression to be unified is, in fact, presented as an expression to be viewed with a given index in the current environment, the index being, in effect, the subscript to be applied to each variable occurring in the expression. In the language introduced earlier, we are dealing with the recursive realization of the expression. Ordinarily the index specified is to be used throughout the expression, but the reduction machinery (discussed below) may sometimes construct explicitly indexed expressions of the form

(#INDEX# k . e)

which represents the expression e viewed with index k.

The unification algorithm is essentially that described in chapter 1, adapted to take indexing into account and to deal properly with the various kinds of atoms as well as CONS, QUOTE and FUNCTION (as described in chapter 3). The function ULT is implemented by #ULTX which is available in two compatible versions: a MACRO used in compiled code and a FEXPR used when interpreting. The MACRO is rather fast when compiled, but abysmally slow when interpreted. The two versions are recorded in the files ULTMAC.LSP (the MACRO) and ULTFXP.LSP (the FEXPR).

13.2.4 Subscripts.

When expressions are explicitly represented it is necessary to construct explicitly subscripted variables, which are represented internally as list structures of the form

(#VAR# atomic-variable . subscript)

as mentioned before. The subscript is represented either by an integer or a structure of the form

(subscript . integer).

In the latter case the subscript should be read from left to right, ignoring parentheses and dots, to obtain the sequence of subscripts applied to the original variable. The implementation requires that the integers appearing in subscripts have the INUM representation.

In order to assure unique representations we maintain a record of all explicitly subscripted variables constructed since initialization as the value of the global variable *SUBVAR. The data structure used for this purpose is arranged to allow quick access and compact storage. We are able to accomplish this in part because an explicitly subscripted variable can be constructed only by "subscripting" a previously defined variable, either atomic or explicitly subscripted. Variables which share subscripts, or initial portions thereof, share the corresponding list structures.

*SUBVAR is a list of the form

```
(NIL (ix1 . box1) ... (ixN . boxN))
```

which is an association list with a header. The "keys" of the a-list are integers (subscripts in fact) arranged in increasing order, but not necessarily consecutive. Each "box" is a pair of the form

```
(var-list . next-list)
```

where var-list is a list of explicitly subscripted variables with subscript ixj (for boxj) and next-list is an association list with header of the sort just described, containing entries for variables with subscripts (ixj . ixk), which in turn contain lists for variables with subscripts ((ixj . ixk) . ixl), and so on, as far as necessary.

13.2.5 Showing.

The functions in this module form explicit representations (or, in some cases, more explicit representations) of implicitly represented expressions. Variables with implicit subscripts are given the explicit form described above, except that an atomic variable with implicit subscript 0 is represented by itself. A moment's reflection shows that implicitly distinct variables will still yield distinct explicit representations.

To "show" (i.e. to realize recursively) arbitrary expressions (or lists of such) we use the function #SHOW and its subsidiaries. In some instances we wish to show an implicit expression on the condition that the result should contain no variables. This is accomplished by #SHOWABLE and associated functions.

For convenience in resolution and reduction we sometimes wish to make sure that the top level of a list is explicitly represented. It is rare in practice that any other kind is encountered, but to deal with the possibility we provide the functions #TSHOW and #TSHOWX.

13.2.6 Reduction.

Evaluation and reduction are accomplished by interpretive routines which perform a sort of "conditional evaluation", conditioned on evaluability of the expressions in question, that is. In this way the evaluability, reducibility, value and reduction (if defined) of some expression can all be determined with one "pass" over the expression. Note that the reduction of a reducible, evaluable expression is always expressed in terms of the value, so there is no need to compute the reduction separately when a value can be obtained.

The reduction of a reducible expression is always represented as an expression to be viewed in the same environment with the same index as the original expression, using an explicitly subscripted expression when necessary.

The special treatment of LOGIC is built in to the reduction and evaluation routines. The other special forms are treated by functions whose names are recorded as #VALOGIC properties of the keywords for the forms. LISP is an exception, being simply the identity FSUBR.

13.2.7 Knowledge Base.

The assertions for a predicate p are stored as the property #ASSERTIONS of p, whose value has the form

((length . flags) . (first . last))

Here 'length' is the number of assertions for p, 'flags' is the bit-wise "and" of the flag bits of the individual assertions, and 'first' and 'last' are the first and last links in the list of internal representations of the assertions. TCONC is used to attach new assertions to this list.

The internal representation of an assertion has the form:

```
      --->*--->*---------
          |    |        |
          |   \|/       |
          |  flags      |
          |             |
      -------------------
     |    |
    \|/  \|/
H--->*--->*-/->*--> ... ->*--/
     |    |    |          |
    \|/  \|/  \|/        \|/
    n/n   hd   tl         tl
```

Here 'n/n' denotes the user name or system-supplied sequence number of the assertion, 'hd' is the head of the assertion, and 'tl' is a predication. The pointer marked "H" shows the part of the representation recoreded in histories. The item 'flags' is an integer which is interpreted bit-by-bit as follows:

2**0: This is a ground assertion
2**1: The head is a ground predication

The bit is '1' to indicate that the assertion has the indicated attribute. Other bits are not used at present.

Secondary indices are generated for each data procedure and each constant identifier appearing in a head of the procedure. If id is such an identifier then (GET id '#ASSERTCREF) is an association list whose entries have the form

(pred count . (first . last))

Here 'pred' is the predicate of a data procedure, 'count' is the number of assertions of this procedure in whose head id occurs, and (first . last) is the TCONC representation of the list of those predications, in the same order that they occur in the procedure. If p is the predicate of a data procedure then (GET p '#CONSTANTS) is a list of the identifiers occurring in heads of assertions of p. This allows the system to erase the secondary index structure when necessary.

13.2.8 Representing Nodes.

This section of the discussion does not correspond to a system module, but presents material on representation needed for forthcoming modules. Nodes of the deduction tree are represented by states, sometimes referred to as clauses. (The nomenclature in these programs has evolved over a period of at least three years.) Besides the full state, various partial states are used in parts of the system. A state has the form:

```
DLCLS  LCLS    DERIV
  ¦      ¦       ¦
  --->*------>*------>*------>*------>*------>*------>env
      ¦       ¦       ¦       ¦       ¦       ¦
      ¦       ¦       \       \       ¦       ¦
      ¦       ¦       ¦       ¦       ¦       ¦
     \¦/     \¦/      ¦      \¦/     \¦/     \¦/
    depth   length    ¦     contn    segl     mi
                     \¦/
                      *--->deriv
                      ¦
                     \¦/
                  assertion
```

Here 'depth' is the depth of the node in the deduction tree, 'length' is the number of predications in the node, including the continuation, if any, 'contn' is the continuation (possibly NIL), 'segl' is a segment list giving the predications in the node, 'mi' is the maximum index used in the node, and 'env' is the environment of the node. If histories are being recorded 'deriv' is the node (DERIV pointer) from which this was obtained, using 'assertion'. The segment list is a list of segments, each of which has the form

```
(index pred ... pred)
```

where the predications 'pred' are to be viewed with the specified index.

If the continuation is not NIL it will have the form

```
((sibling-count . success) . dlcls)
```

where 'sibling-count' is the number of waiting nodes sharing this continuation, 'success' is T if any siblings have been proved, NIL otherwise, and 'dlcls' is the continuation node.

13.2.9 Printing States.

The functions in this module are used to print states, both when monitoring and explaining. The caller is expected to control printing of continuations and the environment to be used. Variables are given in the form v:i (more subscripts if needed).

13.2.10 Resolution.

The special resolution rules are implemented by so-called #RESLOGIC functions which refer directly to the locals of the principal resolution function, #RESOLVENTS. The special rules are used before assertions in the knowledge base (if any). When the procedure involved is a data procedure the secondary index structure is used to obtain the shortest list of assertions possible, which may be empty.

13.2.11 Simplification.

Simplification of states is performed by #SIMPLIFY. Note that it is not assumed that the length given is actually the number of predications represented in the segment list.

13.2.12 Heap Management.

Waiting states are recorded in a priority queue represented as a heap. This is stored in two arrays #HPWT and #HPCLS, each indexed 1 to #HPLIMIT. (#HPWT i) is the weight of the state which is (#HPCLS i). The highest position in use is entry number #HPN, the root being empty when #HPROOTEMPTY. The technique of leaving the root position open for insertion of a new node is due to McCormack [Ph.D. dissertation, Syracuse University, 1978].

When operating in PROLOG mode the waiting states are kept in the global list #STACK.

13.2.13 Undefined Predicates.

The function #ASK conducts the dialogue which ensues when an undefined predicate is encountered. Automatic spelling corrections are based on a very simple measure of the "distance" between words, being essentially Hamming distance.

13.2.14 Continuations.

The routines which manage proved and failed states with continuations are intimately linked to #SETOF. In their present form they are intended only for continuations arising from CONDitionals, with no particular attempt at generality.

13.2.15 Searching.

The function which implements the search for solutions is #SETOF, some of whose parameters are packaged in a list in order to stay under the limit of six arguments for SUBRs imposed by the compiler.

13.2.16 Search-interface.

The functions in this module implement the user's interface to #SETOF. The function FIND which is included here is available for users, but not considered to be part of the "official" interface.

13.2.17 Printing.

The functions in this module print assertions in various forms. Some of these provide for a "quick" mode of printing which may be implemented in future versions of the system. The "public form" of an assertion (clause) is a list

({name} hd tl1 ... tlN)

A special version of GRINDEF is provided. This always prints logic procedures in addition to the properties specified in GRINPROPS. The function which accomplishes this is ##GRINDEF, and is found in the file GRINFN.LSP.

13.2.18 Explaining.

These functions implement the explanation facility. Recall that the derivations of the most recent results are recorded by #SETOF in #DERIVATIONS, the template used in *TEMPLATE.

13.2.19 Editing.

The technique used to edit the knowledge base is very simple. Having gathered up the assertions to be edited, the procedures involved are erased from the knowledge base. When editing is finished the remaining assertions are re-inserted into the knowledge base. While this is inefficient in many respects, it seems to work well enough for the present.

13.2.20 Saving & Restoring.

The files written by SAVE and read by RESTORE (or LOADLOGIC) have the form

FACTS-ASSERTED assertions END-OF-FILE

where 'assertions' consists of assertions and declarations in the form typed to FACTS, without sugar. These programs include some provision for a "quick" mode and for the inclusion of denials -- facilities which have not thus far been implemented.

13.2.21 Control.

Those control functions which simply toggle the value of a global identifier are implemented using #FLAGONOFF. Those which control the special #RESLOGIC functions use #FUNONOFF.

13.2.22 System Initialization.

This module contains those initialization functions which were not included in earlier modules. Note that the character tables are adjusted so that "}" is treated as a letter, rather than escape.

13.2.23 Miscellaneous.

The remaining functions implement some further means for access to the knowledge base together with a few arithmetic operations. Note that the top-level HELP message is just the value of the identifier HELP, and can thus be erased by negligent users.

13.3 SYSTEM BUILDING.

In this section we shall explain the way in which LOGLISP is represented as a collection of files and discuss the broader issues of system building. Details may be obtained by reading the MIC (Macro Interpreted Command) files which are used to compile, build, and run the system.

13.3.1 Files.

The files which constitute the "sources" for LOGLISP are

LOGFNX.LSP	The bulk of the system functions
LOGFNX.DOC	On-line documentation for these
LOGFNX.PRG	Combined listing of functions and documentation
LOGFNX.LAP	Compiled version of LOGFNX.LSP
IFCFNX.DOC	On-line documentation for interface functions
ULTMAC.LSP	MACRO version of #ULTX
ULTFXP.LSP	FEXPR version of #ULTX
GRINFN.LSP	Special version of GRINDEF (##GRINDEF)
LOGMAC.LSP	MACROs from LOGFNX which are included at run time

The user interface functions are documented in LOGFNX.DOC as well as IFCFNX.DOC. Besides these there are some utility files and the documentation package.

DOC.LSP	Documentation package (with documentation)
DOC.LAP	Compiled form
INIT.LSP	Initialization file for LISP

CMPUTL.LSP	MACRO definition utilities
DEVFNX.LSP	System development aids

The documentation package and CMPUTL.LSP are included in the run-time system, which is

LOGLSP.SHR	Sharable high-segment (includes LISP)
LOGLSP.LOW	Low segment

A number of MIC procedures are provided with the system. These are

LOGLSP.MIC	Runs system
LOGLSI.MIC	Runs interpreted form of system
LOGLSC.MIC	Runs compiled form from LAP file
LOGBLD.MIC	Builds image files
LOGLST.MIC	Prints source listing of LOGLISP
LSPCMP.MIC	Compiles LISP programs

LOGLSP.MIC is installation dependent, in that it contains the full file specification for the image files LOGLSP.SHR, LOGLSP.LOW. The others expect all relevant files to be on DSK:, except the LISP system is presumed to reside on SYS:. We should emphasize that these procedures are intended for use by system developers, except, of course, LOGLSP.MIC.

13.3.2 The Manual.

This manual is provided in the files

FRONT.MEM	Title page - contents
LOG1.MEM	Chapter 1
.	.
.	.
.	.
LOG13.MEM	Chapter 13
LOGA.MEM	Appendix A

A complete copy of the manual can be printed with LOGMAN.MIC, also included.

CHAPTER 14

REFERENCES

[Boyer-Moore 1972]

Boyer, R. S., Moore,J. S., The sharing of structure in theorem proving programs. Machine Intelligence 7, Edinburgh University Press, 1972.

[Bruynooghe 1976]

Bruynooghe, M., An interpreter for predicate logic programs, Part I. Report CW 10, Applied Mathematics and Programming Division, Katholieke Universiteit, Leuven, Belgium.

[Clark 1979]

Clark, K. L., McCabe, F., Programmers' Guide to IC-PROLOG. CCD Report 79/7, Imperial College, University of London.

[Colmerauer 1973]

Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P. Un Systeme de Communication Homme-machine en Francais. Rapport, Groupe Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France, 1973.

[Kowalski 1974]

Kowalski, R. A., Predicate Logic as Programming Language. Proceedings IFIP Congress, 1974.

[Kowalski 1979]

Kowalski, R. A., Logic for problem solving. Elsevier North Holland, 1979.

[Roberts 1977]

Roberts, G., An implementation of PROLOG. M.Sc. Thesis, University of Waterloo, 1977.

[Robinson 1965]

Robinson, J. A., A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery 12, 1965, pp. 23-41.

[Robinson 1979]

Robinson, J. A., Logic: Form and Function. Edinburgh University Press and Elsevier North Holland, 1979.

[Roussel 1975]

Roussel, P., PROLOG: manuel de reference et d'utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1975.

[van Emden 1977]

van Emden, M. H. Programming in Resolution Logic. Machine Intelligence 8, 1977, pp. 266 - 299.

[Warren 1977]

Warren, D.H.D., Periera, L.M., Pereira, F., PROLOG - the language and its implementation compared with LISP. Proceedings of a symposium on AI and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8, and SIGART Newsletters 64, August 1977, pp. 109-115.

APPENDIX A

AN INTERACTIVE DOCUMENTATION FACILITY

The file DSKC:DOC.LSP[2461,21] (compiled version DOC.LAP) contains a collection of LISP programs which define a simple facility for documenting LISP systems on-line. The method used is to associate "documentation" with identifiers under the property DOC. This documentation may be any list structure whatever, although the package supports certain elementary conventions regarding function documentation. There are, in addition, a few functions which assist in the preparation of nicely formatted files with function definitions and documentation.

A.1 FUNCTIONS FOR DEFINING DOCUMENTATION.

The package contains functions for defining documentation properties in general, as well as special functions for function documentation.

(DD "I" "DOC")   [FEXPR]

Inserts documentation (DOC) on the property list of the identifier I. DOC may consist of several items, as in (DD FOO (This is a) (silly example)), which results in ((This is a) (silly example)) as the documentation of FOO.

(DFD "F" "DOC")   [FEXPR]

Inserts documentation for a previously defined function F. DFD automatically inserts the argument list and function type (EXPR, FEXPR or MACRO) at the front of the documentation so as to produce standard function documentation. As with DD, DOC may consist of several items. DFD returns the function name F. If F is not, in fact, a function, DFD acts like DD, but returns the list (DD F) to inform the user of its action.

The style of documentation produced by DFD is considered standard in the system, and three functions are provided for defining functions and documentation simultaneously.

(DDE "FN" "ARGS" ("DOC") "BODY")   [FEXPR]

Defines an EXPR named FN with argument list ARGS, documentation DOC and body BODY, which is typed just as for DE. The documentation must be a single item (usually a list) as signified by the parentheses above, to which the argument list and type (EXPR) will be added automatically. DDE actually uses DE to insert the definition, so newly defined functions are added to the file SAVE. DDE returns FN, or (FN REDEFINED), if FN was previously defined to be a function.

(DDF "FN" "ARGS" ("DOC") "BODY")   [FEXPR]

Is like DDE, except that the function thus defined is of type FEXPR.

(DDM "FN" "ARGS" ("DOC") "BODY")   [FEXPR]

Is like DDE, except that the functions thus defined is of type MACRO.

A.2 FUNCTIONS FOR PRINTING DOCUMENTATION.

(DOC "I")   [FEXPR]

Prints the documentation for I (if any) in DEFPROP format, using SPRINT to obtain nice layout and indentation. DOC returns NIL.

(GRINDOC X)   [EXPR]

Controls the printing of documentation by GRINDEF. (GRINDOC T) enables printing of documentation, while (GRINDOC NIL) disables printing of documentation. GRINDOC returns the new value of GRINPROPS, which is the list of properties printed by GRINDEF.

A.3 FUNCTIONS FOR EDITING DOCUMENTATION.

The editing functions are akin to EDITF in that editing commands may be included in the function call or typed outside the call on the same line, if one wishes.

(EDITD "I" "COMS")   [FEXPR]

Edits the documentation of I. COMS is an optional sequence of editing commands. EDITD returns I.

(EDITFD "FN" "COMS")   [FEXPR]

Edits the documentation and definition of the function FN simultaneously as a list having the form

(DOC documentation type definition)

which looks like a segment of the property list of FN. Although this list can be edited any way one likes, the type of the function (as specified in its property list) can not actually be changed, nor can the documentation property be removed. Upon exit from the editor the function definition and documentation are checked to insure that arguments and type agree and, if not, a message to that effect is printed and one is returned to the editor. If FN is not documented EDITFD prints =EDITF and runs EDITF. If FN is documented but not a function, it prints =EDITD and runs that function. EDITFD returns FN.

A.4 FUNCTIONS FOR GENERATING FILES.

The functions described below provide means for writing files with linelengths specified at the terminal. This is particularly useful when generating files of LISP function definitions which are to be incorporated in papers typed on pages of normal size. These functions follow certain common conventions. In each case, the file generated by the function is written on device DSK:, and the name of the file is the first argument. The name should be either an atom or a dotted pair. The linelength with which the file is written is the second argument and should be an integer, usually in the range 60 to 124.

(WRITEPROGS "FILE" "LENGTH" "FLNM")   [FEXPR]

GRINDEFs the identifiers which are MEMBERS of FLNM on FILE with linelength LENGTH. MEMBERS of FLNM which are not identifiers are simply PRINTed. FLNM should be a file name defined for BUILD, as might be constructed with ADDTO. The properties recorded in the file are determined by the current value of GRINPROPS, as always. The resulting file can be read with DSKIN, but does not include the sort of file definition information written by BUILD. WRITEPROGS returns FILE.

(WRITEDOC "FILE" "LENGTH" "FLNM")   [FEXPR]

Generates a file much like WRITEPROGS, except that only DOC properties are recorded in the file.

(WRITEANY "FILE" "LENGTH" "OPERATION") [FEXPR]

Generates FILE with linelength LEN_TH, the contents of the file being written by the evaluation of OPERATION. OPERATION may be any expression whatever whose evaluation results in printing. The local variables of WRITEANY all have the form #...#, so these are unlikely to interfere with global variables appearing in OPERATION. WRITEANY returns the value of OPERATION.

A.5 HINTS ON USING THE PACKAGE.

Since SPRINT does not work particularly well on long lists of atoms, it is usually wise to divide narrative sections of the documentation into lists of manageable length. The next section gives a number of examples.

When developing a collection of LISP programs it seems most convenient to perform (GRINDOC T), so that BUILD will write documentation in the files it creates, which documentation will then be retrieved when the resulting files are read with DSKIN.

One may not wish to include all the documentation in a "production" system, since this could require a good deal of storage. In such circumstances one should BUILD after (GRINDOC NIL) and write a separate documentation file using WRITEDOC. Large linelengths are suggested for files not intended for publication. One can imagine other ways of using the package as well.

Files containing documentation may be compiled ir the usual way, in which case the documentation will appear in the LAP file, where it can be read by DSKIN. Files written by WRITEPROGS or WRITEDOC are likely to be incorporated in RUNOFF source files. RUNOFF will produce the expected result if the text from the program file is preceeded by the command

```
.nf.ts 8,16,24,32,40,48,56
```

One will usually need to insert skip commands in place of the blank lines appearing in files written by these programs. Beware too of characters such as '#' which are of special significance to RUNOFF, and also the ^Y which PRINT generates when atoms cross the end of a line.

A.6 THE PACKAGE.

We give now a listing of the functions in the file DOC.LSP, along with the documentation which is associated with those functions (and included in DOC.LSP). This listing was generated by performing

```
*(GRINDOC T)

(DOC EXPR FEXPR MACRO VALUE SPECIAL)
*(WRITEPROGS (DOC.TXT) 60 DOC)

(DOC . TXT)
*
```

```
(FSUBR (DD DFD DDE DDF DDM DOC EDITD EDITFD WRITEPROGS WRITE^Y
DOC WRITEANY))




(DEFPROP DD
 (LAMBDA(L)
  FEXPR
  (DD "I" "DOC")
  (Define documentation for I))
DOC)

(DEFPROP DD
 (LAMBDA (L) (PUTPROP (CAR L) (CDR L) (QUOTE DOC)) (CAR L))
FEXPR)


(DEFPROP DFD
 (LAMBDA(L)
  FEXPR
  (DFD "F" "Documentation")
  (Attaches documentation to function F under property DOC)
  (Args and type (EXPR FEXPR MACRO) included automatically)
  (Same as DD if F not a function))
DOC)

(DEFPROP DFD
 (LAMBDA(L)
  (PROG (FD FT)
        (SETQ FD (GETL (CAR L) (QUOTE (EXPR FEXPR MACRO))))
        (COND
         ((NULL FD)
```

```
             (RETURN
              (LIST (QUOTE DD) (APPLY# (FUNCTION DD) L)))))
           (SETQ FT (CAR FD))
           (SETQ FD (CADR FD))
           (PUTPROP
            (CAR L)
            (CONS (QUOTE LAMBDA)
                  (CONS (CADR FD) (CONS FT (CDR L))))
            (QUOTE DOC))
           (RETURN (CAR L))))
FEXPR)

(DEFPROP DDE
 (LAMBDA(L)
  FEXPR
  (DDE "FN" "ARGS" ("DOC") "BODY")
  (Define EXPR FN with documentation DOC)
  (DOC is one list; args and type are automatic)
  (BODY as for DE))
DOC)

(DEFPROP DDE
 (LAMBDA(L)
  (PROG (FN ARGS DOC BODY R)
        (SETQ FN (CAR L))
        (SETQ ARGS (CADR L))
        (SETQ DOC (CADDR L))
        (SETQ BODY (CDDDR L))
        (SETQ R
              (APPLY# (FUNCTION DE)
                      (CONS FN (CONS ARGS BODY))))
        (PUTPROP
         FN
         (CONS (QUOTE LAMBDA)
               (CONS ARGS (CONS (QUOTE EXPR) DOC)))
         (QUOTE DOC))
        (RETURN R)))
FEXPR)


(DEFPROP DDF
 (LAMBDA(L)
  FEXPR
  (DDF "FN" "ARGS" ("DOC") "BODY")
  (Define FEXPR FN with documentation DOC)
  (DOC is one list; args and type are automatic)
  (BODY as for DF))
DOC)
```

```
(DEFPROP DDF
 (LAMBDA(L)
  (PROG (FN ARGS DOC BODY R)
        (SETQ FN (CAR L))
        (SETQ ARGS (CADR L))
        (SETQ DOC (CADDR L))
        (SETQ BODY (CDDDR L))
        (SETQ R
              (APPLY# (FUNCTION DF)
                      (CONS FN (CONS ARGS BODY))))
        (PUTPROP
         FN
         (CONS (QUOTE LAMBDA)
               (CONS ARGS (CONS (QUOTE FEXPR) DOC)))
         (QUOTE DOC))
        (RETURN R)))
FEXPR)


(DEFPROP DDM
 (LAMBDA(L)
  FEXPR
  (DDM "FN" "ARGS" ("DOC") "BODY")
  (Define MACRO FN with documentation DOC)
  (DOC is one list; args and type are automatic)
  (BODY as for DM))
DOC)

(DEFPROP DDM
 (LAMBDA(L)
  (PROG (FN ARGS DOC BODY R)
        (SETQ FN (CAR L))
        (SETQ ARGS (CADR L))
        (SETQ DOC (CADDR L))
        (SETQ BODY (CDDDR L))
        (SETQ R
              (APPLY# (FUNCTION DM)
                      (CONS FN (CONS ARGS BODY))))
        (PUTPROP
         FN
         (CONS (QUOTE LAMBDA)
               (CONS ARGS (CONS (QUOTE MACRO) DOC)))
         (QUOTE DOC))
        (RETURN R)))
FEXPR)
```

```
(DEFPROP DOC
 (LAMBDA(L)
  FEXPR
  (DOC "I")
  (Prints documentation of I in DEFPROP format if defined))
DOC)

(DEFPROP DOC
 (LAMBDA(L)
  (COND
   ((GET (CAR L) (QUOTE DOC))
    (PRINC (TERPRI (QUOTE /()))
    (PRIN1 (QUOTE DEFPROP))
    (PRINC (QUOTE / ))
    (PRIN1 (CAR L))
    (SPRINT (GET (CAR L) (QUOTE DOC)) 2.)
    (PRIN1 (TERPRI (QUOTE DOC)))
    (TERPRI (PRINC (QUOTE /))))))
  NIL)
FEXPR)


(DEFPROP GRINDOC
 (LAMBDA(X)
  EXPR
  (X : Add DOC to GRINPROPS)
  ((NOT X) : Delete DOC from GRINPROPS)
  (Returns new value of GRINPROPS))
DOC)

(DEFPROP GRINDOC
 (LAMBDA(X)
  (COND (X (COND
            ((NOT (MEMQ (QUOTE DOC) GRINPROPS))
             (SETQ GRINPROPS (CONS (QUOTE DOC) GRINPROPS))))
           GRINPROPS)
        ((SETQ GRINPROPS (REMOVE (QUOTE DOC) GRINPROPS)))))
EXPR)


(DEFPROP EDITD
 (LAMBDA(L)
  FEXPR
  (EDITD "I" "COMS")
  (Edits documentation of I)
  (No constraints))
DOC)
```

```
(DEFPROP EDITD
 (LAMBDA(L)
  (EDITE (GET (CAR L) (QUOTE DOC)) (CDR L) (CAR L))
  (CAR L))
FEXPR)


(DEFPROP EDITFD
 (LAMBDA(L)
  FEXPR
  (EDITFD "FN" "COMS")
  (Edits combination of function and documentation as)
  (DOC documentation type body)
  (Insists that documentation agree with function))
DOC)

(DEFPROP EDITFD
 (LAMBDA(L)
  (PROG (FN DOC DEFN LST)
        (SETQ FN (CAR L))
   GO   (SETQ DOC (GET FN (QUOTE DOC)))
        (COND
         ((NULL DOC) (PRINT (QUOTE =EDITF))
                     (RETURN (APPLY# (FUNCTION EDITF) L))))
        (SETQ DEFN (GETL FN (QUOTE (EXPR FEXPR MACRO))))
        (COND
         ((NULL DEFN) (PRINT (QUOTE =EDITD))
                      (RETURN (APPLY# (FUNCTION EDITD) L))))
        (SETQ LST
              (LIST (QUOTE DOC) DOC (CAR DEFN) (CADR DEFN)))
        (EDITE LST (CDR L) FN)
        (COND
         ((NOT
           (AND (EQUAL (CADR DOC) (CADR (CADR DEFN)))
                (EQ (CADDR DOC) (CAR DEFN))))
          (PRINC
           (TERPRI
            (QUOTE
             "Documentation and function do not agree")))
          (GO GO)))
        (RETURN FN)))
FEXPR)


(DEFPROP WRITEPROGS
 (LAMBDA(F)
  FEXPR
  (WRITEPROGS "FILE" "LENGTH" "FLNM")
  (GRINDEFs all atoms which are MEMBERS of FLNM on DSK:FILE)
```

```
  (PRINTs non-atomic MEMBERS of FLNM)
  (Linelength given by LENGTH)
  (FLNM should be a filename defined for BUILD))
DOC)

(DEFPROP WRITEPROGS
 (LAMBDA(F)
  (PROG (OLDC LWB L)
        (EVAL
         (LIST (FUNCTION OUTPUT)
               (QUOTE WRITEPROG)
               (QUOTE DSK:)
               (CAR F)))
        (SETQ OLDC (OUTC (QUOTE WRITEPROG) NIL))
        (SETQ LWB (LINELENGTH NIL))
        (LINELENGTH (CADR F))
        (SETQ L (GET (CADDR F) (QUOTE MEMBERS)))
   LOOP (COND
         (L (COND ((ATOM (CAR L))
                   (EVAL (LIST (QUOTE GRINDEF) (CAR L))))
                  ((TERPRI (PRINT (TERPRI (CAR L))))))
            (SETQ L (CDR L))
            (GO LOOP)))
        (LINELENGTH LWB)
        (OUTC OLDC T)
        (RETURN (CAR F))))
FEXPR)


(DEFPROP WRITEDOC
 (LAMBDA(L)
  FEXPR
  (WRITEDOC "FILE" "LENGTH" "FLNM")
  (Prints documentation for all atoms which are MEMBERS)
  (of FLNM on DSK:FILE with linelength LENGTH)
  (FLNM should be a filename defined for BUILD))
DOC)

(DEFPROP WRITEDOC
 (LAMBDA(L)
  (PROG (OLDG)
        (SETQ OLDG GRINPROPS)
        (SETQ GRINPROPS (QUOTE (DOC)))
        (APPLY# (FUNCTION WRITEPROGS) L)
        (SETQ GRINPROPS OLDG)
        (RETURN (CAR L))))
FEXPR)
```

```
(DEFPROP WRITEANY
 (LAMBDA(#F#)
  FEXPR
  (WRITEANY "FILE" "LENGTH" "OPERATION")
  (Performs OPERATION with output directed to DSK:FILE)
  (Linelength LENGTH during operation)
  (Returns result of operation))
DOC)

(DEFPROP WRITEANY
 (LAMBDA(#F#)
  (PROG (#OLDC# #LWA# #LWB# #L# #FILE# #PF#)
        (SETQ #FILE# (CAR #F#))
        (SETQ #LWB# (CADR #F#))
        (SETQ #PF# (CADDR #F#))
        (EVAL
         (LIST (FUNCTION OUTPUT)
               (QUOTE WRITEPROG)
               (QUOTE DSK:)
               #FILE#))
        (SETQ #OLDC# (OUTC (QUOTE WRITEPROG) NIL))
        (SETQ #LWA# (LINELENGTH NIL))
        (LINELENGTH #LWB#)
        (SETQ #L# (EVAL #PF#))
        (LINELENGTH #LWA#)
        (OUTC #OLDC# T)
        (RETURN #L#)))
FEXPR)
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DA

FIL

4