CLISP - Conversational LISP

Warren Teitelman
Xerox Palo Alto Research center
Palo Alto, California 94304

Abstract

CLISP is an attempt to make LISP programs easier to read and write by extending the syntax of LISP to include infix operators, IF-THEN statements, FOR-DO-WHILE statements, and similar ALGOL-like constructs, without changing the structure or representation of the language. CLISP is implemented through LISP's error handling machinery, rather than by modifying the interpreter: when an expression is *encountered whose* evaluation causes an error, the expression is scanned for possible CLISP constructs, which are then converted to the equivalent LISP expressions. Thus, users can freely intermix LISP and CLISP without having to distinguish which is which. Emphasis in the design and development of CLISP has been on the system aspects of such a facility, with the goal in mind of producing a useful tool, not just another language. To this end, CLISP includes interactive error correction and many 'Do-what-I-Mean* features.

* * *

The syntax of the programming language LISP»»»»»» is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesesization. For the benefit of readers unfamiliar with LISP syntax, the basic element of LISP programs is called a fgr.m. A form is either (1) atomic, or (2) a list, the latter being denoted by enclosing the elements of the list in matching parentheses. In the first case, the form is either a variable or a constant, and its value is computed accordingly. In the second case, the first element of the list is the name of a function, and the remaining elements are again forms whose values will be the arguments to that function. In Backus notation:

Figure 1
Syntax of a LISP Form

```
<form> ::« <variable>|<constant>|
     (<function-name> <form> ... <form>)
```

For example, assign x the value of the sum of A and the product of B and C is written in LISP as (SETQ X (PLUS A (TIMES B C))) .

The syntax for a conditional expression is correspondingly simple:*

Figure 2
Syntax of a conditional Expression

```
<conditional expression> ::x
     (COND <clause> ... <clause>)
<clause> :;= (<form> <form>)
```

Note that there are no IF's, THEN's, BEGIN's, END'S, or semi-colons. LISP avoids the problem of parsing the conditional expression, i.e., delimiting the individual clauses, and delimiting the predicates and consequents within the clauses, by requiring that each clause be a separate element in the conditional expression, namely a sublist, of which the predicate is always the first element, and the consequent the second element- As an example, let us consider the following conditional expression which embodies a recursive definition for FACTORIALS**

Figure 3
Recursive Definition of Factorial

```
(COND
  ((EQ N 0)

  (T (TIMES N (FACTORIAL (SOB1 N)))))
```

* Actually, a clause can have more than two consequents, in which case each form is evaluated and the value of the last form returned as the value of the conditional. However, for the purposes of this discussion, we can confine ourselves to the case where a clause has only one consequent.

** The expression in Figure 3 is shown as it would be printed by a special formatting program called PRETTYPRINT. PRETTYPRINT attempts to make the structure of LISP expressions more manageable by judicious use of identation and breaking the output into separate lines. Its existence is a tacit acknowledgment of the fact that LISP programs require more information than that contained solely in the parenthesization in order to make them easily readable by people.

The first clause in this conditional is the list of two elements ((EQ NO) 1), which says if (EQ N 0) is true, i.e., if N is equal to 0, then return 1 as the value of the conditional expression, otherwise go on to the second clause. The second clause is (T (TIMES N (FACTORIAL (SUB1 N)))), which says if T is true (a tautology-the ELSE of ALGOL conditionals), return the product of N and (FACTORIAL (SUB1 N)). The latter is evaluated by first evaluating (SUB1 N), and then calling FACTORIAL (recursively) on this value.

As a result of the structuring of conditional expressions, LISP does not have to search for words such as IF, THEN, ELSE, ELSEIF, etc., when interpreting or compiling conditional expressions in order to delimit clauses and their constituents: this grouping is specified by the parentheses, and is performed at input time by the READ program which creates the list structure used to represent the expression. Similarly, LISP does not have to worry about how to parse expressions such as A+B+C, since (A+B)*C must be written unambiguously as (TIMES (PLUS A B) c), and A+(B*C) as (PLUS A (TIMES B C)). In fact, there are no reserved words in LISP such as IF, THEN, AND, OR, FOR, DO, BEGIN, END, etc., nor reserved characters like +, -, *, /, =, «-, etc.* This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that "looks at" other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code.' It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners), 'pushing down' is something programs do very well, and people do poorly. As an example, consider the following two "equivalent" sentences:

> "The rat that the cat that the dog that I owned chased caught ate the cheese."
>
> versus
>
> "I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices Buch as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even when the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e., lists) for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the words, not in the structure. For example, the expression in Figure 4

Figure 4
careless Definition of Factorial

```
(COND (EQ N 0) 1)
      (T TIMES N FACTORIAL ((SUB1 N)))
```

is recognizable as FACTORIAL even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP's benefit, than for the programmer's.

CLISP is designed to make LISP programs easier to read and write by permitting the user to employ various infix operators, IF-THEN- ELSE statements, FOR-DO-WHILE-UNLESS-FROM-TO-etc. expressions, which are automatically converted to equivalent LISP expressions when they are first interpreted. For example, FACTORIAL could be written in CLISP as shown in Figure 5.

Figure 5
CLISP Definition Of FACTORIAL

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))
```

Note that this expression would be represented internally (after it had been interpreted once) as shown in Figure 3, so that programs that might have to analyse or otherwise process this expression could take advantage of the simple syntax.

CLISP also contains facilities for making sense out of expressions such as the careless conditional shown in Figure 4. Furthermore, CLISP will detect those cases which would not generate LISP errors, but are nevertheless obviously not what the programmer intended. For example, the expression (QUOTE <expression> <form>) will not cause a LISP error, but <form> would never be seen by the interpreter. This is clearly a parentheses error. CLISP uses both local and global information to detect, and where possible, repair such errors. However, this paper will concentrate primarily on the syntax extension aspects of CLISP, and leave a discussion of the semantic issues for a later time.

* except for parentheses (and period), which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers.

There have been similar efforts in other LISP systems, most notably the MLISP language at Stanford.* CLISP differs from these in that it does not attempt to replace the LISP language so much as to augment it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in INTERLISP*. Thus, any expression that is well-formed from LISP'S standpoint will never be seen by CLISP (e.g., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the interpreter does not •know* about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-what-I-Mean (DWIM) analyzers'»■• which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent LISP form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it becomes the correctly translated LISP form. In this way, the analysis and translation are done only once.

Integrating CLISP into the LISP system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions.* For example, if the user has defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1) (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

2) (LIST X*FACT N), where FACT is not the name of a variable but instead is the name of a function, means (LIST X*(FACT N)>, i.e., N is FACT'S argument.

3) (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X* (FACT N)) .

4) cases (1), (2) and (3) with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (LIST X*FCCT N) were encountered, where there was both a function named FACT and a variable named FCT. The user would first be asked if FCCT were a misspelling of FCT. If he said YES, the expression would be interpreted as (LIST (X*FCT) N).* If he said NO, the user would be asked if FCCT were a misspelling of FACT, i.e., if he intended X*FCCT N to mean X*(FACT N). If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if X*FCCT was to be treated as CLISP, since FCCT is not the name of a (bound) variable.** If he

* INTERLISP (formerly BBK-LISP[6]) is implemented under the BBN TENEX timesharing system[2] and is jointly maintained and developed by Xerox Palo Alto Research Center and Bolt, Beranek, and Newman, Inc., Cambridge, Mass. It is currently being used at various sites on the ARPA Network, including PARC, BBN, ISI, SRI-AI, etc.

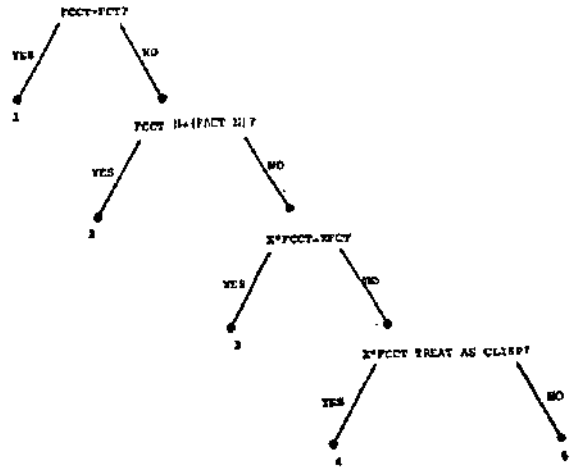
* Through this discussion, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary.' For transformations on expressions in his programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.

** This question is important because many of our LISP users already have programs that employ variables whose names contain CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask the user if A/B is intended to be CLISP, in case the user really does have a free variable named A/B, but has mistakenly used A/B here in a context where it was not bound.

said YES, the expression would be transformed, if NO, it would be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, as with GET-PRAENT. This sort of transformation would be considered after the user said NO to x*FCCT N -> X*(FACT W). The complete graph of the possible interpretations for (LIST X-FCCT N) where FCT and XFCT are the names of variables, and FACT is the name of a function, is shown in Figure 6.

Figure 6

Possible interpretations of (LIST X*FCCT N)



The final states for the various terminal nodes shown in Figure 6 are:

```
1:   (LIST (TIMES X FCT) N)
2:   (LIST (TIMES X (FACT N)))
3:   (LIST XFCT N)
4:   (LIST (TIMES X FCCT) N)
5:   (LIST X*FCCT N)
```

CLISP can also handle parentheses errors caused by typing 8 or 9 for '(• or ')'- (On most terminals, 8 and 9 are the lower case characters for '(• and «)', i.e., •(< and '8' appear on the same key, as do •)' and •9•.) For example, if the user writes N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the LISP function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled!* For example, CLISP can successfully translate the definition of FACTORIAL:

(IFFN*0 THENN 1 ESLE N*8FACTTORIALNN-1)

to the form shown in Figure 3, while making 5 spelling corrections and fixing the parenthesis error.*

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits the user to say things like:

(FOR OLD X-M TO N
DO (PRINT X) WHILE (PRIMEP X))**

However, the user can also write OLD (X–M), (OLD X-M), (OLD (X-M>), permute the order of the operators, e.g., DO (PRINT X) TO N FOR OLD X+M WHILE (PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, TO, DO, or WHILE, or leave out the word DO entirelyl And, of course, he can also misspell PRINT, PRIMEP, M, or N!***

CLISP is well integrated into the INTERLISP system. For example, the above iterative statement translates into an equivalent LISP form using PROG, COND, GO, etc.**** When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if the user PRETTYPRINTS his program, at the corresponding point in hie function, PRETTYPRINT "knows" to print the original CLISP. Similarly, when the user edits his program, the editor makes the translation invisible to the user. If the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

* Where misspelling includes running adjacent words together, as shown in example.

* CLISP also contains a facility for converting from LISP hack to CLISP, so that after running the above definition of FACTORIAL, the user could 'CLISPIFY» to obtain:
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))-

** This expression should be self explanatory, except possibly for the operator OLD, which says X is to be the variable of iteration, i.e., the one to be stepped from M to N, but X is not to be rebound. Thus when this loop finishes execution, X will be equal to N.

***** In this example, the only thing the user could not misspell is the first X, since it specifies the name of the variable of iteration. The other two instances of X could also be misspelled.**

```
****(PROG NIL
        (SETQ X M)
    $$LP(COND
          ((OR (IGREATERP X N)
               (NOT (PRIMEP X)))
           (RETURN)))
        (PRINT X)
        (GO $$LP)).
```

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant.* Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the LISP executive, CLISP sits between the user's programs and the LISP interpreter.

Only a small effort has been devoted to defining a core syntax for CLISP. Instead, most of the effort has been concentrated on providing a facility which' 'makes sense' out of the input expressions using context information as well as built-in and acquired information about user and system programs. Just as communication is based on the intention of the speaker to produce an effect in the recipient, CLISP operates under the assumption that what the user said was intended to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the syntactic aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

CLISP has just become operational and the expected reactions and suggestions from users will do much towards polishing and refining it. However, the following anecdote suggests a favorable prognosis: after being cursorily introduced to some of the features of CLISP, two users wanted to try out the iterative statement facility, but neither of them were sure of the exact syntax. The first user thought that if they just typed in something "reasonable", the system would figure out what they meant. And it did!

References

1. Berkeley, E.C. "LISP, A Simple Introduction," in The Programming Language LISP, its Operation and Applications. Berkeley, E.C. and Bobrow, D.G. (editors), MIT Press, 1966.

* Bobrow, O.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S. "TENEX, a Paged Time Sharing System for the PDP-10." communications of the. ACM, March 1972, vol. 15, Ho. 3.

' McCarthy, J. et al. LISP 1.5 Programmer's Manual. MIT Press, 1966.

* Smith, D.c MLISP User's Manual. Stanford Artificial Intelligence Project Memo AI-84, January 1969.

5 Teitelman, w. "Automated Programming - The Programmer's Assistant." Proceedings of Fall Joint Computer Conference. December 1972.

* Teitelman, W., Bobrow, D.G., Hartley, A.K., Murphy, D.L. BBN-LISP Tenex Reference Manual, Bolt, Beranek, and Newman, Inc., August 1972.

7. Teitelman, W. "Do What I Mean," Computers and Automation. April 1972.

* Teitelman, W. "Toward a Programming Laboratory," Proceedings of First International Joint Conference on Artificial Intelligence. Walker, D. (editor), May 1969.

* weissman, C. LISP 1.5 Primer. Dickenson Press, 1967.

2.PAK: A SNOBOL-BASED PROGRAMMING LANGUAGE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

JOHN MYLOPOULOS, NORMAN BADLER, LUCIO MELLI AND NICHOLAS ROUSSOPOULOS

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO

Descriptive Terms: Programming language, SNOBOL, backtracking, relational data base, artificial intelligence.

Abstract

This paper describes a programming language for Artificial Intelligence applications which offers

(a) a data base, in the form of a collection of labeled directed graphs where knowledge can be stored

(b) pattern directed information retrieval and pattern invoked function calls

(c) primitive statements which enable the user to construct flexible searching algorithms.

The language is an extension of SNOBOL in its design and implementation and uses SNOBOL's string pattern matching facilities for its own (graph) pattern matching.

1. Introduction

l.pak was designed and implemented in order to facilitate AI research at the University of Toronto. Its design was influenced by other languages designed for similar reasons such as C0NNIVF.R[1,2] , PLANNFR[3], QA4[4], SAIL[5,6], and our decision to implement it as an extension of SNOBOL and keep it relatively inexpensive (in terms of the time and space required for the execution of programs).

This paper only discusses the main features of *l.pak,* how they can be used, their relation to features offered by other languages for AI, and the success of the current implc-mentation. More details on the language are available elsewhere [7] It is assumed that the reader is familiar with the basic features of SNOBOL [8].

2. The Data Base

The data base for each *l.pak* program consists of a collection of directed, labeled graphs (hereafter graphs) such as that shown in FIG. 1. A list of transitive and/or intransitive edges is associated with each node, which define either properties of (the object represented by) the node or relations which hold between the node and other elements of the same graph.

A linear order exists for the nodes of each graph defined by special edges labeled NEXT.

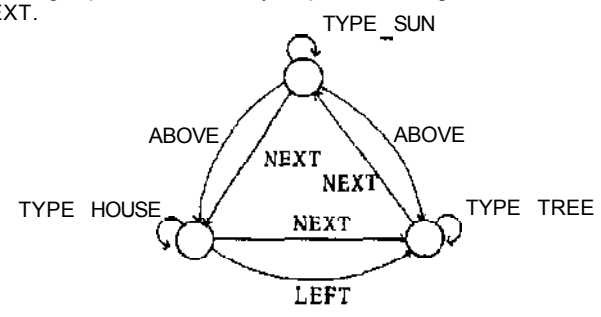


FIG. 1.

Edge labels (properties) consist of one or more atoms separated by underscores. The first atom is the attribute of the property, while subsequent atoms are its modifiers.

Information can be retrieved from the data base by matching (graph) patterns against it.

Patterns arc specified in terms of sequences of path descriptions. For example, the pattern

<$X,(LEFT_AARB?YJ, (FAR).SW-

contains one path description which will match paths that

(a) Begin at the node which is the value of atom X ,

(b) Move along an edge whose label has LEFT as attribute and is followed by at least one modifier,

(c) Move along an edge whose label has FAR as attribute,

(d) End at node $W.

If such a path connects nodes $X and $W for a particular data base, the pattern match succeeds and the modifier of LEFT is assigned to atom Y , while the graph pattern match returns $W , the last node vi?ited during the match. Patterns are evaluated by the function SEARCH.

AARB is a special property pattern which will match any atom. *l.pak* offers several other built-in property patterns and operators similar to those offered by SNOBOL to help the user specify classes of edge labels in his graph patterns Thus (graph) pattern matching is essentially driven by property pattern matching and can be easily implemented using the string pattern matching facilities in SNOBOL.

Note that there may be several paths which will match the same pattern. For example, the pattern match

SEARCH(<$X,(LEFT),(ABOVE^FAR)>)

could return node n1 or nZ In FIG. 2. The pattern match however will return just one of the two nodes.

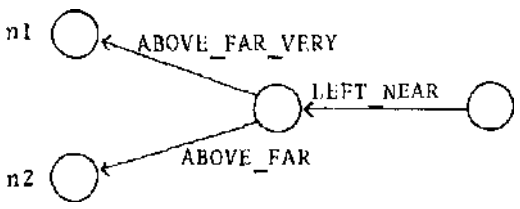


FIG. 2.

We will show later how to generate all the matching alternatives.

Information can be added to or deleted from the data base through the special functions ADD and DEL.

ADD's main function is to create new information with a graph as context. For example, the function call

ADD(<$X,(LEFT),(FAR),?W>)

will create enough new edges and nodes to make the pattern match of its argument against the data base successful. In the process, a (node) value will be assigned to W . ADD will also perforin a simple consistency-redundancy test for each new edge attached to a node, with respect to the edge list of that node. Thus an intransitive edge labeled *TYPE HOUSE* will not be added to the edge list of a node which already contains an edge labeled TYPE_H0USE TALL because it is redundant.

An example of a DEL function call is

DEL(<?X,(LEFT),-$Y,(RTGHT)>)

where the special operator '-' specifies that the node or edge it operates on must be deleted. For this function call, a pattern match takes place first, and if unsuccessful the call fails; otherwise the node $Y is deleted along with all the edges that point, to or away from it.

l.pak also offers SNOBOL tables and arrays as means for representing information. Moreover, as in SNOBOL, the user can define his own data types. Explicit list facilities using CONS, CAR and CDR and list notation are provided. Expressions such as

$L = ($A, $B, $C, $D)

may be used to construct lists; here the value of L is a four element list.

3- Program Structure and Control

l.pak statements are similar to their SNOBOL cousins. For example, the statement

A: DEL(<$X, (LEFT) , - (R1GIITJVB0YE) >*)* :S(A)F(B)

is labeled A , and will attempt to delete the edge whose label matched R1GHT_AB0VE, if the pattern match of <$X,(LEFT),(R]CHT_ABOVE)> against the data base succeeds. If the DEL function call succeeds, control is passed back to A , otherwise control is passed to the statement labeled B .

Each *l.pak* statement will succeed or fail and this can be used for program control. Unless otherwise specified, control will pass to the next statement of a *l.pak* program. Thus backtracking as encountered in PLANNER and QA4 is only offered during graph pattern matching in *l.pak.* A different kind of backtracking will be discussed later.

4. Function Requesjts_

4.1. Explicit function requests

l.pak functions can be defined through PDEFINE. For example, the statement

PDEFINE(NEW(X:NODE]NULL,Y:PROPERTY|PATTERN,? Z) W_W1,NEW)

defines a function named NEW with three formal parameters; the first must be of type NODE or have as value the null string, the second of type PROPERTY or PATTERN, while the third is called by result. Thus the statement

NEW($X1,$Y1,$Z1)

will fail even before NEW is called because the third argument is not called by result, while

NEW($X1,$Y1,?Z1)

will proceed with the execution of the function call if $X1 is a node or the null string, and $Y1 is a property or a (graph) pattern.

The specification of allowable types for each formal parameter does not mean that the type of function arguments is fixed inside a function call; it only helps to check whether the function request makes sense.

The same function name may be used inside several PDEFINE calls, thus giving the same name to several different functions. This means that a statement such as

NEW($X1,$Y1,?Z)

may cause several function calls unti] one is found that succeeds. In fact, the function name specified by a function request in *l.pak* can be a string pattern, as in

(T1I|AX) ($X,$Y,?Z)

which will cause the call of any function whose name matches the string pattern (THJAX) (i.e., begins with TH or AX) and whose parameter description is matched by the arguments of the function request.

This feature offers the user flexibility in specifying a function request similar to that offered by theorem provers where each axiom can be considered as a function without a name to be called whenever there is a suspicion that it may be of use.

Unlike SNOBOL, *l.pak* treats field designators of programmer-defined data types, array references and table lookups as special cases of a function request. Thus, if the user writes

CAR($X)

1.pak will try to treat this as a field designator, then as an array reference, then as a table lookup, and if all these attempts fail, it will start calling functions named CAR until one of them succeeds.

4.2 Implicit function requests

It is often the case that information not stored in the data base is actually true in the universe of discourse. For example, a node may represent a house which has been recognized as one of the objects represented in a line drawing, and the question "Does there exist an object to the left of the house?" may be asked. The user may attempt to answer it by writing

SEARCH (<-$X, (LEFT_* FCN) ,?Y>)

where X points to the node representing the house, and *,FCN arc special modifiers to be discussed later. If the pattern match fails, either there is no such object, or it exists but it is not represented by a node in the graph where $X is imbedded. We would like to keep the value of the pattern match independent of these two possibilities and have therefore introduced the notion of imp!icit function requests ■ Informally, such function requests will be invoked by the system in an attempt to construct information needed for the successful evaluation of a pattern match whenever the special modifier FCN appears at the end of a property pattern. * is also a special modifier which specifies that modifiers that follow it should be treated differently by *l.pak* than atoms that precede it. In the previous example, if the system has found no edges which leave $X and match LEFT (not LEFT *FCN), it will therefore call functions named" LEFT with implicit argument $X , looking for one that succeeds.

Thus the pattern match

SEARCH(<$X,(DIMENSIONS AARB?Y AARB?Z AARB?W * FCN)>)

will either match an edge label to the property pattern

DJMENS10NS_AARB?Y AARB?Z AARB?W

and will return its modifiers through Y, 2 and W , or it will call a function named DIMENSIONS which has three formal parameters, all called by result, which will attempt to find the dimensions of $X . In this case the modifiers of the property for which we are trying to establish an edge were used as arguments of the implicit function call, while the attribute was used as a function name. In the example

SEARCH(<$X, (TYPE_HOUSE_TALL_VKRY_*__FCN)>)

on the other hand, we may want to establish the existence of an (intransitive) edge labeled TYPEJIOUSE_TALL_VERY by first calling the function TYPE with only argument the node $X , then the function HOUSE with arguments $X and the output of TYPE, then TALL and finally VERY. In other words, an attempt to establish an edge in the data base may cause several function calls.

5. Generator Calls

Generators were introduced by PLANNER and used extensively by CONNIVER where they provide one of the most important language constructs. *l.pak* offers them too, as a method of generating alternatives.

A generator is defined by a piece of code and is assigned a name. For example,

```
GEN.INT, INTEGER;
     $INT = $INT+1;
     $DOMAIN = C0NS($]NT,$D0MA1N)  :(EXIT);
END.INT;
```

defines an integer generator named INT. This generator simply considers the first element stored in its DOMAIN list (which is similar to CONNTVER's possibilities list), increments it by 1, stores the result in DOMAIN, and returns it as value. EXIT specifies that execution of this generator call is complete and that its DOMAIN list should be kept active for future calls.

A generator can be used once it has been bound to an atom. Thus

```
$X <= INT(5)
```

assigns the generator INT to X and initializes its DOMAIN list to (5). Now whenever we write X< > , the INT generator will be evaluated returning another integer. Note that there can be several active copies of the same generator, each bound to a different atom. For example,

```
     $X <- INT(51;
     $Y <« INT(7);
A:   $OUTPUT = X< > + Y< >;
     LT($OUTPUT,100)  :S(A);
```

will assign to X the generator INT with its DOMAIN list initialized to (5). It will also assign to Y the generator INT with its DOMAIN list initialized to (7). 5+7 will then be evaluated and assigned to OUTPUT. As in SNOBOL, any string assigned to OUTPUT is automatically printed. It is then checked whether the value of OUTPUT is less than 100 and if so 6+8 is added and printed, then 7+9, 8+10 etc.

In general, the user can pass within the angle brackets a list of expressions to be appended to the DOMAIN list of a generator each time he calls it. He can also pass an argument to be used for that particular generator call.

There are several built-in generators. For example, NODES will generate all the nodes that lie at the end of a path which matches a given graph pattern. Thus if we write

```
$Z <= NODES(<$X,(LEFT),(ABOVE_FAR)>)
```

the first time Z< > is encountered with background the graph shown in FIG. 2, it will return, say, node n] , the second time node n2 , and if it is called again it will fail.

6. More on Backtracking

In some cases backtracking (i.e., reset of the program state in case of failure) will not be necessary, but in others it will save the programmer considerable effort. *1.pak* allows a form of backtracking by extending the feature of declaring variables local to a function or a generator in several directions:

(a) All variables which appear in a function or generator body can be declared local for a particular function or generator call by using the keyword VLOCAL during the function definition or generator definition or binding. LOCALness may be specified as dependent on the success of failure of a function or generator call by using SVLOCAL or FVLOCAE instead of VLOCAL. Thus FVLOCAL defines a backtracking situation (i.e., reset in case of failure) for program variables only for "the function or generator it is associated with.

(b) Changes made to the data base can also be declared local by using SDLOCAL, FDLOCAL or DLOCAL.

If the user wants a reset of variables and the data base state, he can use SLOCAL, FLOCAL or LOCAL. This way he has some flexibility in specifying exactly which changes in his program he considers reversible and under what, conditions .

7. Examples

This section describe? three simple *l.pak* programs which demonstrate the features already discussed and point out a few others that arc not as important.

Suppose that we want to define a collection of functions named LEFT which somehow express adequately our own notion of what LEFT means (geometrically). These functions will be defined with respect to a list, $LIST, of objects and it will be assumed that there exists a function REL.LEFT which succeeds or fails depending on whether the object represented by its first argument is to the left of the object represented by its second argument.

First we give a definition of LEFT which postulates its transitivity

```
     PDEFINE(LEFT(TA1L:NODE,HEAD:NODE)Z_AUX,
                          LEFT,DLOCAL);
BEGIN.LEFT;
     $Z <- NODES(<$TAIL,(LEFT)>);
     $AUX = Z< >   :F(FRETURN);
A:   ADIH<$AUX,(TRIED)>)    :F(B);
     I DENT($AUX,$HEAD)      :S(RETURN);
B;   $AUX - Z<(<$AUX,(LEFT)>)>  :S(A)F(FRETURN);
END.LEFT;
```

This function will be evaluated as follows for given $TAIL and $HEAD:

(a) Z is bound to the NODES generator described in section S with its DOMAIN list

initialized *to* the pattern <$TAIL, (LF.FT)> . Thus the first call of Z will return a node to the LEFT of $TA1L which is assigned to AUX.

(b) Each node assigned to AUX is labeled TRIED by ADD. If $AUX already has an intransitive edge labeled TRIED, ADD fails (because it cannot change the data base) and another node to the LEFT of $TAIL is assigned to *AUX.*

(c) It is checked whether $AUX is IDENTical to $HEAD, and if so the function call RETURNS;

(d) Otherwise, another node to the LEFT of $TAIL is assigned to AUX and the new graph pattern <$AUX, (LF.FT)> is appended to the DOMAIN list of Z .

When all the nodes to the LEFT of $TAIL have been considered, Z will Start generating nodes to the LEFT of nodes to the LEFT of $TAIL, and this will be repeated until all the nodes to the LEFT of nodes... to the LEFT of $TAIL have been tried. Because of the third argument of PDEFINE, all changes made to the data base will be erased when the call to LEFT is complete, also AUX, Z will be reset to the values they had before the function call.

Thus this definition of LEFT defines a breadth-first search of the graph where $TAIL is imbedded in an attempt to find $HEAD, and it has been formulated by using the generator feature. it is interesting to compare it with the following *l.pak* function which also postulates the transitivity properly of LEFT, by relying on graph pattern matching and implicit function requests:

```
        PDEFINE (LEFT (TAIL: NODE ,1IEAI): NODE),
                                        SLEFT);
BEGIN.SLEFT;
      SEARCH(<$TAIL,(LEFT),$HEAD>) :S(RETURN);
      SEARCH(<$TAIL,(LEFT),(LEFT * FCN),
             $HEAD>) :S(RETURN)F(FRETURN);
END.SLEFT;
```

Here it is first checked whether there is an edge with attribute LEFT connecting $TAIL to $HEAD, and if this is not the case, it is checked whether there is a node, say nl , such that there is an edge with attribute LEFT connecting $TA1L to nl , and nl and $HEAD can be connected with an edge labeled LEFT. In checking for the latter condition, the user has specified that implicit function requests involving LEFT-named functions are allowed. This is a recursive definition of transitivity for LEFT in other words. The searching algorithm it defines is depth-first and may enter an infinite loop for graphs representing geometrically strange worlds.

The second definition of LEFT accepts two nodes as arguments and succeeds or fails depending on whether the object represented by the first node is to the LEFT of the object represented by the second

```
      PDEFINE[LEFT(TAIL:NODE,HEAD:NODE)A  B,
                                        TLEFT.)7
BEGIN.TLEFT;
     SEARCH(<$TAIL,(REPR_AARB?A) :
            $HEAD,(REPR_AARB?B)>)
                                     :F(FRETURN);
     REL.LEFT($$A,$$B) :S(RETURN)F(FRETURN);
END.TLEFT;
```

The graph pattern match executed by SEARCH finds the atoms which modify REPR on intransitive edges associated with $TAJL and $HEAD and assigns them to A and B respectively. ':' specifies the end of one path description and the beginning of another. The values *of* the two *atoms* assigned to A *and* B are the elements of $L1ST represented by $TAIL and $HEAD (FIG. 3). Once these atoms are found, REL.LEFT can be

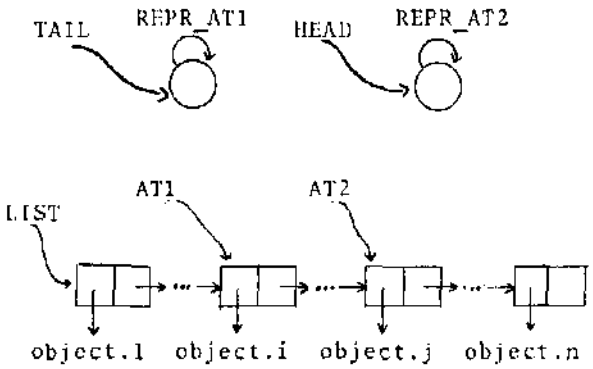


FIG. 3.

called with arguments the objects object.i, object,j, represented in some way.

A third possible definition of LEFT as a generator returns nodes representing objects to the LEFT of a given node $TAIL which is passed as message to the generator, and is also a global variable:

$Lt-ir <= LIiFTDEF(#$rAILj

This statement assigns to LEFT the generator LEFTDEF with its DOMAIN list initialized to the unevaluated expression $TAI1. ('W' keeps its operand unevaluated until it is encountered at execution time, and is therefore similar to the SNOB0L, unary operator '*') - Below we define LEFTDEF.

```
 GEN.LEFTDEF, NODE;
 GA:  SLEFTDEF = SEARCH (– $LEFTDEF , (NEXT)> ) ;
       I DENTf$LEFTDEF,$TAIL)    :S(FRETURN) ;
       SEARL'H(<$TAIL, (LEFT_*_FCN) ,$LEFTDEF>)
                                      :F(GA>;
       $D0MA1N = ($LEFTDEF)       : (EXI'I'") ;
 END.LEFTDEF;
```

Whenever this generator is executed, LEFTDEF is first assigned the value of TAIL (this is done automatically by the system). Then the NEXT node is found and assigned to LEFTDEF; NEXT defines a circular order on the nodes of the graph where STAIL is imbedded, and we are using it here in order to traverse the graph. It is checked whether SLEFTDEF is IDENTical to $TAIL, which would mean that we are back at the starting node and there are no more nodes to the LEFT of $TAIL; if not, it is checked whether the current value of LEFTDEF is to the LEFT of $TA1L. Implicit *function* requests are allowed for this check. If the answer is negative, control returns to the statement labeled GA and another node is assigned to LEFTDEF', otherwise the DOMAIN list of LEFTDEF is set to ($LEFTDEE), a one-element list, and we EXIT. Next time this copy of LEFTDEF is called, search will resume with the NEXT node in the graph where $TAIL is imbedded until they have all been considered.

The user can now write

SEARCH(<$X,(LEFT * FCN),$Y>)

or SEARCH(<$X,(LEFT~*_GEN),?Y>)

and expect the system to either find the necessary information in the data base or to call the appropriate functions or generators (depending on whether the special modifier is

FCN or GEN) in an attempt to construct it.

8. Discussion

1.pak has been implemented in SPITBOL[9], an efficient version of SNOBOL. SPITBOL uses both a compiler and an interpreter, offers most SNOBOL features (in particular the function EVAL), plus a few more, has a very fast garbage collector and handles user-defined data types very efficiently. It requires approximately ~50K bytes and runs several times faster than the BTL implementation of SNOBOL.

Some of the reasons that led us to choose SNOBOL over other candidate languages are listed below:

(a) It offers pattern matching facilities. This has helped the design and implementation of graph pattern matching; moreover, SNOBOL users will have no difficulty adapting to the graph pattern matching formalism since it is an extension of string pattern matching.

(b) It offers tables and user-defined data types. These features were used extensively during the implementation of the *1.pak* system, and are offered by *1.pak* at very little extra cost.

(c) SNOBOL's control structure is unusual but flexible and well-suited to AI applications programming. *1.pak* offers most of that control structure, in addition to the various shades of the LOCAL feature, function requests, generator calls etc.

Labeled graphs have already been used for the representation of knowledge (e.g., Palme [10], Rumelhart et al [11]). *1.pak* graphs have the additional feature however that the user has a choice of representing a piece of information structurally or as a property. Which form he chooses should depend on how often the parts of this piece of information will be retrieved and manipulated independently of each other. For example, the statement "John gives a gift to Mary" could be represented (rather crudely and with various subtleties of the sentence's meaning ignored) by the graph shown in FIG. 4(a), 4(b) or 4(c), depending on whether we will be referring explicitly to the gift or Mary and their properties, or will simply refer to them as parts of a property John has.

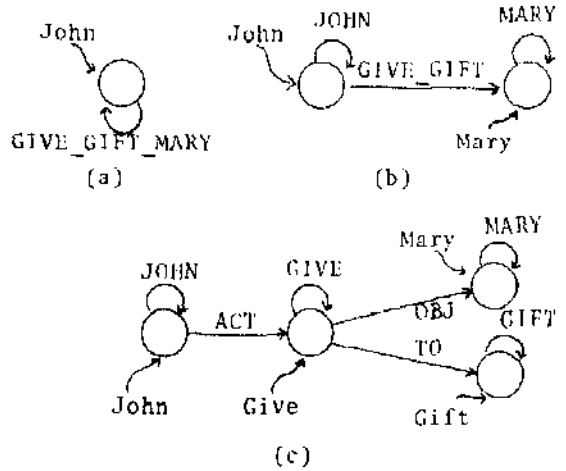


FIG. 4.

Unlike PLANNER etc., *1.pak*'s data base offers only partial associativity. This may be inconvenient for the user in certain cases, but it offers him more control over his data base's thirst for memory space. More convential data structures (arrays, tables, user-defined data types) are also available in *1.pak at very little expense for the 1.pak* system since they are mostly handled by SNOBOL.

Graph pattern matching offers many features found in PLANNER in that a similar backtracking mechanism is used and implicit function requests can be considered as consequent theorem calls. If the user agrees with Sussman and McDermott's criticism of PLANNER's backtracking [2], he can switch to a programming style favoring generators where he has more control over the backtracking mechanism he uses. We feel that both features will be found useful.

The *1.pak* implementation uses both a compiler and an interpreter and requires a minimum of ~140K (this includes ~50K for the SPITBOL system). There are plans to use *1.pak* for question-answering, scene analysis and natural language understanding to test it and find which features are useful and should be made more prominent and which ones should be modified.

Acknowledgements

We would like to thank Carl Hewitt for several useful suggestions, also Walter Berndl for helping us with the implementation and portions of the design of *1.pak*. This research was partially supported by DOC and DRB grants.

References

1. McDermott, D.V., Sussman, G.J. The CONNIVER reference manual. MIT AI Memo. 259.
2. Sussman, G.J., McDermott, D.V. From PLANNER to CONNIVER: a genetic approach. FJCC, 1972, 1171-1180.
3. Hewitt, C. Description and theoretical analysis of PLANNER. MIT AI-TR-258, 1972.
4. Derksen, J., Rulifson, J.F., Waldinger, R.J. The QA4 language applied to robot planning. FJCC, 1972, 1181-1192.
5. Swinehart, D., Sproull, R. SAIL. Stanford AI Project, Operating Note No. 57.2, Janaury 1972.
6. Feldman, J.A., Low, J.R., Swinehart, D.C. and Taylor, R.H. Recent developments in SAIL. FJCC, 1972, 1193-1202.
7. Mylopoulos, J., Badler, N., Melli, L., Roussopoulos, N. An introduction to *1.pak*, a programming language for AI applications. TR-52, Department of Computer Science, University of Toronto, May 1973.
8. Griswold, R.E., Poage, J.F., Polonsky, I.P. The SNOBOL4 programming language. Prentice-Hall, 1971 (second edition).
9. Dewar, R.B.K. SPITBOL, Version 2.0. Illinois Institute of Technology, 1971.

10. Palme, J. Making computers understand natural language. In Artificial Intelligence and Heuristic Programming, Findler N." and Meltzer, B. (Eds .). Edinburgh University Press, 1971.

11. Rumelhart, D,, Lindsay, P.H., Norman, D.A. A process model for long-term *memory.* In Organization of Memory, Tulving, E. and 'Donaldson, W. (Eds .), Academic Press, 1972.