BACKTRACKING IN MLISP2
An Efficient Backtracking Method tor LISP

David Canfield Smith, Horace J. Enes
Computer Science Department
Stanford University
Stanford, California

Abstract

An efficient backtracking method for LISP, used in the MLISP2 language, is described. The method is optimal in the following senses:

<1) Only necessary state information is saved. The backtracking system routines are sufficiently efficient to require less than ten percent of the execution time of typical jobs.

(2) Most common operations — fetching/storing the value of a variable or the property of an atom, function entry/exit — take no longer with backtracking than without it. This is achieved by not changing the way values are stored.

(3) If backtracking is not used, an insignificant overhead is involved in maintaining the backtracking capability.

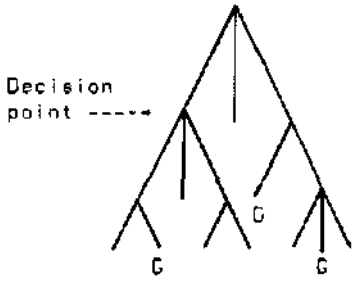
The MLISP2 algorithm and philosophy are briefly contrasted with those of several existing backtracking systems, with historical comments on the development of the theory of backtracking.

Backtracking

> Backtracking algorithms are nondeterministic, not in the sense of being random, but in the sense of having free will.
>
> Robert Floyd

Backtracking 4,5,6,7,8,9,10,11,12,13 has begun to be used in the past five years as a control structure for programming languages dealing with problems in artificial intelligence. This paper is an attempt to contribute to a better understanding of what backtracking is, what it is useful for, and how to implement it efficiently. Briefly, backtracking is an algorithmic device for solving problems expressible as a set of possible alternatives, called a goal tree, where not all of the alternatives will lead to the desired goal. At each branching point in the tree, a decision must be made as to which alternative to try next. Backtracking is designed to simplify the programming of this type of problem.

This work was supported by Grant PHS MH 06645-12 from the National Institute of Mental Health.



Idealized goal tree

A branching point is called a "DECISION POINT" in this paper. Frequently, insufficient information is available at decision points to determine which branches will lead to the goal. If a wrong branch is tried, it "FAILs"; that is, the program returns to the decision point, pretends that the incorrect branch had never been attempted, and selects another alternative, This process is called "BACKTRACKING." It is algorithmic because it covers the entire goal tree; every branch at every branching point will eventually be tried in the worst case.

Several languages have incorporated backtracking, but none of them have incorporated exactly the same features and none of them have implemented backtracking in exactly the same way. The differences are in terms of objectives and in terms of methods. We shell first discuss the nature and then the origin of these differences, beginning with the different implementation methods used

Before continuing, we should mention briefly that backtracking has been criticized in some quarters as being an inherently bad control structure.[4] Criticisms of any concept come in two flavors: (1) "theoretical criticisms" of the concept, and (2) "pragmatic criticisms" of machine implementations of the concept. The criticisms of backtracking have mostly fallen into this latter category; they criticize the early, pioneering systems like PLANNER. More recent systems like ECL and MLISP2 incorporate enough flexibility to answer many of the published objections to backtracking. Finally, drawing on our personal experience, we wish to point out that MLISP2 has been used for two years by the artificial intelligence community at Stanford. It has shown itself to be versatile, easy to use, and efficient enough to be eminently practical.

Two Views of Computations

All existing backtracking methods have been implemented from one of two viewpoints, which we shall call the "sequential" view and the "state" view of backtracking. These correspond to the way computations are ordinarily viewed The "sequential" view of computations holds that a computation is a sequence of discrete steps, the last of which yields the desired result. All algorithms consist of such sequences. For example, the Algol statements

```
X := 10;
Y := 20;
Z := (X*Y/2) + 3*Y;
W := (Z + 2*X) / (Y - 5);
```

may be viewed as a sequence of steps leading to the computation of a value for W. The third and fourth statements themselves consist of a sequence of more primitive steps.

Another way to view computations is in terms of state transformations, the "state" approach. The "STATE OF A COMPUTATION" is defined to be the set of current values of all variables. The program counter, stacks, etc. are considered to be system variables. A computation is a sequence of state transformations, the result of which is a state representing the desired computation. Continuing the previous example but representing the state of the machine by a partially-specified tuple;

STATE = <X, Y, Z, W, ...>,

the computation becomes

```
<X, Y, Z, W, ...> → <?,  ?,  ?,  ?, ...>
                  → <10, ?,  ?,  ?, ...>
                  → <10, 20, ?,  ?, ...>
                  → <10, 20, 160, ?, ...>
                  → <10, 20, 160, 12, ...>
```

Though we have not mentioned the intermediate states (e.g. 3*Y), a complete description must include them.

Sequential Backtracking

Theoretical Systems [5,6] These two views of computations have affected the theory and development of backtracking systems. Golomb and Baumert were among the first to explore the computational applications of backtracking. They took the state view of computations and demonstrated that backtracking could be used to reduce the search of the space of solution states. Floyd's early investigations took the sequential approach. He proposed having an "inverse" for each statement in the computation, including assignments, conditionals, subroutine entries and exits, and I/O. Backtracking then consists of stopping the forward execution of statements end executing Inverse statements until the decision point is again reached. At this point, Golomb and Floyd agree, everything has been reset to its original condition, and processing may again proceed in a forward direction.

> Each command [computation step] is expanded into one or more commands, some of which carry out the effect of the original command in the nondeterministic algorithm, and which also stack information required to reverse the effect of the command when backtracking is needed, while others carry out the backtracking by undoing all the effects of the first set. [4, p.638]

This has several advantages. Expanding each computation step is a mechanical process that can be added to existing compilers and interpreters. The same code is generated as before to carry out the forward steps, plus some additional code to save the backtracking information. Furthermore, the modular design facilitates adding backtracking to a system. The inverse of each statement type can be added and debugged separately.

PLANNER ' But the sequential ("inverse statement") method of implementing backtracking soon changed its objectives. It was observed that not ell statements need to be undone. Sometimes variables will be set on one branch in the goal tree that cannot affect the execution of other branches. Therefore, the argument goes, why bother to backtrack these variables? This Is the approach taken by PLANNER, a LISP-based pattern-matching system incorporating several powerful non-procedural features such as goal-directed computation. PLANNER uses backtracking to implement these features. (Actually, PLANNER has not been fully implemented yet; this discussion really deals with a subset called MICRO-PLANNER implemented by Sussman, Winc-grad and Charniak. Nevertheless, we shall continue to use the name PLANNER.) Not all statement-types are undoable. Functions that may be backtracked are designated by the letters "TH" on the front of their names. For example,

```
(SETQ X 10)
```

cannot be backtracked, but

```
(THSETQ X 10)
```

can be. Similarly, THPUTPROP, THREMPROP, THASSERT end THERASE are functions for changing the data base that are backtrackable whereas PUTPROP and REMPROP are not. But function *entries and* exits *are* always undoable; i.e. PLANNER will always restore the control environment when a failure happens, but It will not restore the values of variables or properties of atoms unless explicitly instructed to. This gives the programmer more control over backtracking, and enables him to eliminate the saving of superfluous information On the other hand it requires the programmer to know what information mutt be saved, shifting the burden of backtracking

to the programmer's shoulders, and in the end making such systems harder to use. A further disadvantage is that the programmer may overspeolfy the amount of information to be saved; it is frequently difficult and non-intuitive to decide what is the optimum amount of information to save.

This implement ion of backtracking has had a profound and confusing effect on its objectives. A distinction has come to be made between automatic control and data backtracking. [1] In the authors' view, this distinction is a negative aspect of the theory of backtracking that has obscured rather than clarified the issues involved. The main purpose of backtracking is to enable a program to try later alternatives in a goal tree as if earlier unsuccessful ones had never been attempted If the state of the computation is not reset at the beginning of each alternative, then the alternatives will behave differently depending on the order in which they are tried. This is a red herring that merely obscures the understanding of nondeterministic programs. If the programmer wants some information preserved when an alternative fails, he should explicitly say so, and any good nondeterministic programming language should provide him with facilities for doing so.

State Backtracking

Corresponding to the "state" view of computations, there is a "state" view of backtracking. This approach may be summarized as follows: When a decision point is encountered during a computation, the complete state of the machine (the current values of all variables including system variables) is "saved." When a failure occurs, the state of the machine is "restored" to the saved state in one operation, and computation proceeds along another alternative at the decision point. If there are no more alternatives, the failure is propagated to the preceding decision point. The state method has led to a more faithful adherence to Floyd's and Golomb's view of backtracking, namely that everything is restored to the way it was before the incorrect alternative was attempted. In addition to the control environment being always restored, as in PLANNER, the data environment is also always restored.

The state-saving and -restoring approach (state method) has one main advantage over the inverse-statement approach (sequential method): the process of failing and trying another alternative is usually more efficient. In the inverse-statement approach, statements may be backtracked unnecessarily. For example, suppose a decision point occurs, and then the value of a variable is changed 100 times before a failure happens. Each of these 100 stores to the variable will have to be undone by an "inverse store," a restoration of the previous value of the variable. But each inverse store will just undo the effect of the previous one; the last inverse store executed effectively undoes them all. 99 of the 100 inverse stores will be wasted. In the state-saving approach, failure transfers control directly back to the decision point. Once there, the entire state is restored in one operation. In MLISP2, this operation is a very rapid one.

ECL [10] ECL is a blend of the sequential and state methods. It has an NASSIGN operator corresponding to PLANNER'S THSETQ, but it also uses a "backup stack" much like MLISP2's state stack (see below). In an interesting variation on MLISP2's stack saving algorithm, the working stack is saved on the backup stack when functions are about to exit, rather than when decision points are executed. In some cases this will result in less information being saved than in MLISP2. However, it has the peculiar property that when a failure occurs, variables not explicitly saved will be restored to the last values they had in the fuction, not to the values theny had when the decision point was set. As in PLANNER only the control environment is automatAically restored. The programmer has the responsibility for explicitly saving (via NASSIGN) variable values he wants restored correctly. In most other respects the ECL algorithm is very similar to MLISP2's.

SAIL [3] SAIL has taken a different approach. Rather than add a context mechanism to the language, the SAIL implementers added a coroutine structure. Coroutining is logically equivalent to backtracking. The set of alternatives at a decision point is represented by a set of coroutines. Failure is achieved by deactivating the coroutine representing the current path and activating another one. In SAIL, the programmer is required to do all state saving himself. Two new statements have been added to do this:

REMEMBER <list of variables> IN <context>

RESTORE <list of variables> FROM <context>

The special word ALL may be used instead of the list of variables and means "do the operation on all the variables previously remembered in that context," Thus SAIL is similar to PLANNER and ECL in that it automatically restores the control environment but not the data environment; the data must be explicitly restored. If a programmer really wants SAIL to behave like a state-saving system, he must at every decision point do a REMEMBER of every variable (including every array element) in the system, and at failure do a RESTORE ALL. It then becomes equivalent to P0P-2 (see below). One seldom uses SAIL in this manner, however; the REMEMBER/RESTORE primitives *are* intended to give the programmer a convenient way to keep certain data from being destroyed during processing in hypothetical situations, not to implement full backtracking.

While coroutining is logically equivalent to backtracking, it is neither physically nor conceptually the same. The internal structures are quite different; for example, there is no "backup stack" in coroutine systems. Conceptually the main difference seems to center on the issue of whether information should be saved automatically or explicitly. The emphasis on simplifying programming has led most backtracking systems to do a large number of things

automatically — saving and restoring data, and control management — whereas coroutines have been regarded as a language tool, like iteration, that should be invoked explicitly.

MLISP2 [13] In MLISP2, the system automatically saves the necessary information. The advantages of automatic state saving are that ft is simple to use, and it eliminates human error. The programmer never has to figure out what values to save; indeed, he may often be unaware that state saving is going on. In addition to eliminating a possible source of bugs (not saving enough information), possible inefficiencies (saving too much information) are also prevented. Programming in this type of a system is not much more difficult than programming in a system without backtracking.

POP2 [2] The efficiency of the state approach depends on the efficiency of saving and restoring the state. The straightforward but inefficient method is to copy the value of every currently-active variable into some area of memory or secondary storage *every* time a decision point is encountered. Thereafter, no further attention is paid to changes in variable values. When a failure occurs, the saved values are reloaded from storage, and every variable is restored to its old value regardless of whether or not its value had changed. This is the method used by POP-2 when backtracking was introduced into that language.

QA4 [12] QA4 is much more efficient in its state saving than POP-2. At decision points not much happens except that a context number is incremented. Thereafter, when a value is stored in a variable, the context number is associated with the new value. In fact nothing much happens at failures either, except that the context number is decremented and a jump is executed! This is faster than any other system's failure. Furthermore, In QA4 it *is* possible to modify or restore a backtracking context anywhere in the goal tree and then resume from there. But in QA4 the penalty is paid in referencing a variable's value (or any atom's property). All properties including the VALUE property are stored in a structured AL1ST under each atom. This ALIST must be searched every time a variable or property is referenced The net effect is that fetches and stores slow down by one to two orders of magnitude. MLISP2 stores variables and properties in such a way that fetches take virtually no longer with backtracking than without, and stores are slower only in some cases.

The MLISP2 Algorithm

MLISP2 is an extensible language using backtracking and based on the Stanford LISP 1.6 system " on the PDP-10 computer. The language is intended to be a practical tool for implementing product/on *compilers and translators, as* well as being a research tool. ML1SP2 has been operational for two years, which has given us a good deal of experience with the practical problems of production backtracking systems. A logic compiler (LCF [9]), a deduction system (FOL), an English parser, an English to French translator, an ALGOL compiler, and the MLISP2 translator itself are major systems that have been written in MLISP2. This experience has led us to the conclusion that at this point not just one more feature, but efficiency and simplicity are the fundamental needs to make backtracking practical.

The philosophy of MLISP2 is that backtracking should simplify programming. The programmer should never be concerned with explicitly saving information just so that statements can be backtracked, and he should seldom have to rewrite routines so that they can be included in a backtracking program. (In PLANNER, existing routines may have to be rewritten by changing SETQs to THSETQs, PUTPROPs to THPUTPROPs, etc, before they can be included in a nondeterministic program. The inverse changes have to be made when a backtracking routine is included in a deterministic system.) Such considerations have nothing to do with problem solving.

MLISP2 implements backtracking by modifying the LISP interpreter and the LAP assembly program. The principle change is to the BIND code, by which LAMBDAs and SETQs bind their variables In interpreted functions. In addition, a "STATE STACK" is maintained on which information from the normal pushdown stack (P) and the special pushdown stack (SP) is saved. In the Stanford LISP 1.6 system, all control information and the values of local variables in compiled functions are put on the P stack. Therefore, saving the contents of the P stack will save this information, The values of all variables in interpreted functions and of variables used free in compiled functions are put on the property lists of the variables. These will be called "property list variables." Saving the property lists of atoms will save the current values of property list variables, as well as any other property list changes. Recursive calls on functions using property list variables stack the old variable values on the SP stack before rebinding, so that saving the SP stack will save their old values. These three structures — the P stack, the SP stack, and the property lists of atoms, — together with a control point to transfer to when a failure occurs, completely specify the state of any LISP l.C imputation.

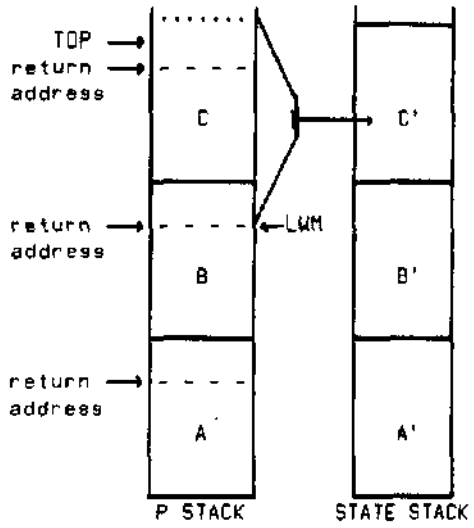
This is a lot of information to handle at once, causing many implementers to shy away from the state-saving approach. But the volume may be reduced by saving just the incremental state, just the changes to the state that have occurred since the last decision point. The work load at decision points may be further reduced by distributing the state saving throughout the computation.

When an MLISP2 decision point is encountered, there are three major effects on the system:

1. A unique positive integer is associated with each dynamic decision point. This number is called the "CONTEXT NUMBER," and the context number in effect during any given part of the computation *is* called the "CURRENT CONTEXT NUMBER" When a decision point is encountered, the current context

number is incremented by one. It monotonically increases unless a decision point is deleted. (Normally a decision point is deleted only when all the alternatives at it have been tried and have been unsuccessful, although it may also be deleted explicitly.) The context number provides a means of referencing specific backtracking contexts and of communicating between contexts. In MLISP2, any context which knows the number of an earlier context may pass information back to that context, called "setting variables in context." Thus when a branch fails but gains valuable information in the process of trying, it can pass the information up the tree to be used in selecting another branch or by the next branch selected.

2. The second thing that happens at decision points is the "incremental" P and SP stacks are saved on the state stack. The definition of the incremental stacks is somewhat complicated. Consider the situation represented in the figure below.



Two decision points have already been set and the incremental P stack at each has been copied into the state stack: A -* A', B -* B'. (Since basically the same operations are performed on the SP stack, we will only discuss the P stack here.) Now a third decision point is about to be set. The question is; how much should be copied this time? A system variable called the low water mark (LWM) always points to the level of the P stack below which the state stack contains a complete copy of everything. Therefore, just the information from LWM to the top of the P stack must be copied to the state stack, This is accomplished by a memory-to-memory block transfer (one multi-cycle instruction on the PDP-10 after some initial set-up). Finally the new LWM is set to the stack location of the return address of the function containing the decision point, which is usually the first return address from the top of the P stack. This requires searching the P stack. We cannot simply use the top of the P stack since the function's local variables and temporary results, which are stored above its return address, may be modified before the function exits. (If this happened, the state stack would no longer contain a complete copy of everything below LWM.) Also the return address to which LWM points is changed to jump to a system routine; this routine moves LWM down to the previous one whenever the etack is about to become smaller than the current LWM. This Is similar to ECL's method which moves LWM down to the next return address, rather than to the previous LWM,

3. No property list variables are explicitly saved at decision points, but incrementing the context number affects property list variables later. Associated with every property list variable is the context number in effect when the variable was last set. Whenever a property list variable is about to be set, its context is compared with the current context number. If they are the same, then the value is simply changed and processing continues. No information is saved If they are different, then

(a) the variable's old value and context, together with the current context number, are saved on a context list (see below);

(b) the variable's context is changed to the current context number, to reflect the fact that the variable has now been set in this context;

(c) finally, the variable's value is changed to the new value.

The same process occurs whenever any property under an atom is changed, not just the VALUE property. The context associated with an atom is actually in the form

```
((indicator1 . context1)
 (indicator2 . context2)
 ... )
```

where in the case of variables one of the indicators is VALUE. This list must be searched whenever a property Is about to be changed. The old values are saved on a "CONTEXT LIST," in the form

```
((context-changed-in
    (atom1 indicator1 old-value1 old-context1)
    (atom2 indicator2 old-value2 old-context2)
    ... )
 ... ).
```

When a feilure occurs, the system runs down this list restoring all the properties that had been changed in the current context. The context list is generally short since only one atom/indicator pair will occur in each context, namely the first change to that pair. This is a large improvement over the inverse statement method ot restoring values.

Saving the values when variables are about to be set distributes the state saving throughout the program. Most

existing backtracking methods have this property. The advantage is that the amount of information saved becomes roughly proportional to the amount of work done on any one branch of the goal tree. If the branch fails early, then little state-saving work will have been done; if processing continues longer on the branch, then more state information comes to be saved.

MLISP2 Improvements In State Saving

(a) Not *every* change to a nondeterministic variable is saved (as in PLANNER and ECL), just the *first* change in each backtracking context.

(b) MLISP2 uses the standard LISP 1.6 value cell for variable values, so that fetching a variable's value is just as fast with as without backtracking, in fact, MLISP2 uses the standard LISP representation for all properties on property lists, so that all GETs are the same speed nondeterministicaily as deter ministically. This is the main source of the efficiency that MLISP2 has over QA4.

(c) When LISP 1.6 functions are compiled, many things are done more efficiently. Local variables are stored on the P stack, and their values are fetched or stored in one machine instruction. Fetches on free variables require only one instruction. Function entry and exit require one instruction. All of these efficiencies are preserved by MLISP2. The only inefficiency introduced is in stores to free variables, which are represented as property list variables in LISP J.6 and thus must go through the process explained in (3) above. Since in LISP, functions are usually debugged in interpreted mode and only compiled to increase efficiency, it is crucial in a production backtracking system that the efficiency of compilation be preserved. In research systems like PLANNER and QA4, efficiency comparable to MLISP2 has not been attained.

Language Features for Backtraeking

> **She knows there's no success like failure and that failure is no success at all.**
>
> **Bob Dylan**

MLISP2 uses backtracking to implement a context sensitive pattern matcher. Pattern matching routines are written using a new expression, the LET expression. As in PLANNER, these routines may be invoked when their syntax pattern matches an input stream, though unlike PLANNER the MLISP2 input stream must be unstructured (e.g. tokens in a file or In a linear list). The MLISP2 pattern matcher is designed to assist *and* simplify writing translators for other languages. In addition to pattern matching, a second new expression has been added to MLISP, the SELECT expression, as the way to incorporate backtracking in ordinary programs. We will not explain these expressions in great detail here; they are explained fully in the MLISP2 report.[13] However we will discuss their use of backtracking.

LET expression

The pattern language includes three "meta syntax" constructions which directly create decision points; REP (repeat), OPT (optional) and ALT (alternative). Their primary purpose is to simplify, clarify, and reduce the number of rules necessary to specify a complex syntax. Examples:

1. { REP <MIN> <MAX> { <PATTERN> } <SEPARATORS> }

```
LET IDENTIFIER_LIST (L) =
    { (REP 0 M (<IDENTIFIER>) ',) }
    MEAN
    MAPCAR('CAR, L);
```

Meaning: Let the production "identifier_list" be zero or more identifiers separated by commas, and have as its value a list of the identifiers scanned. This corresponds to the BNF rules:

```
<IDENTIFIER_LIST> ::= <EMPTY>
<IDENTIFIER_LIST> ::= <IDENTIFIERS>
<IDENTIFIERS>     ::= <IDENTIFIER>
<IDENTIFIERS>     ::= <IDENTIFIERS>,<IDENTIFIER>
```

2. { OPT <PATTERN> }

```
LET IF (*,E1,*,E2,E3) =
    ( IF <EXPRESSION> THEN <EXPRESSION>
            (OPT ELSE <EXPRESSION>) )
    MEAN
    <'COND, <E1, E2>,
            <'T, IF E3 THEN E3[2] ELSE 'NIL>>;
```

Meaning: Let the production "if" be the word IF followed by an expression, followed by the word THEN and another expression, optionally followed by the word ELSE and a third expression, and have as its value the corresponding LISP "COND" expression. This corresponds to the BNF rules:

```
<IF> ::= IF <EXPRESSION> THEN <EXPRESSION>
<IF> ::= IF <EXPRESSION> THEN <EXPRESSION> ELSE
                <EXPRESSION>
```

3. { ALT <PATTERN> | ... | <PATTERN> }

```
LET EXPRESSION (E) =
    ( (ALT <BEGIN_END> | <IF> | <FOR>) )
    MEAN E[2];
```

Meaning: Let the production "expression" be either the "begin_end," the "if," or the "for" production, and have the value of the respective production as its value. This corresponds to the BNF rules:

```
<EXPRESSION> ::= <BEGIN_END>
<EXPRESSION> ::= <IF>
<EXPRESSION> ::= <FOR>
```

We will give a brief example of how nondeterminism is used in these expressions. The execution of an OPT expression proceeds as follows:

(a) Set a decision point

(b) Match the OPT pattern against the current input stream.

<c) If the match succeeds, the OPT returns with a list of the values of the pattern elements matched. If the match fails, FAILURE is called.

(d) If a FAILURE happens, either in the matching of the OPT pattern or later, the state of the computation is restored to its state at the beginning of the OPT, the decision point is deleted, and computation proceeds with the empty list NIL as the value of the OPT.

SELECT expression

```
SELECT <value_expression>
    FROM <formal_variable>:<domain>
    SUCCESSOR <successor_expression>
    UNLESS <termination_condition>
    FINALLY <termination_expression>
```

The other way to incorporate backtracking into a program is by the SELECT expression. The SELECT expression is like a nondetemninistic FOR-loop. It sets a decision point and allows each of a set of alternatives to be tried. It has a very general form, and is a generalization of Floyd's CHOICE function and Fikes* CHOICE-CONDITION combination.[4] The various expressions are converted to functions by the following expansion:

(LAMBDA <<formal_variable>) <expression>)

These LAMBDA functions are defined as:

```
<value function>          : <domain> → <value>
<successor function>      : <domain> → <domain>
<termination condition>   : <domain> → TRUE, FALSE
<termination function>    : <domain> → <value>
```

The execution of a SELECT expression proceeds as follows:

(a) Evaluate the domain, which may be any expression, to get an initial domain.

(b) Set a decision point.

(c) Apply the termination condition to the domain. If the value is TRUE, delete the decision point and apply the termination function to the domain. Exit with this value as the value of the SELECT. (The termination function may call FAILURE). If the value of the termination condition is FALSE, proceed to the next step.

(d) Apply the value function to the domain, and exit with this value as the value of the SELECT.

<e) If a failure returns to the SELECT, apply the successor function to the domain to yield a new domain.

(f) Go to step (c).

We will give a few examples of how the SELECT may be used,

(1) Floyd's CHOICE function may be written:

```
EXPR CHOICE (N);
    SELECT I FROM 1;I SUCCESSOR I+1
        UNLESS I GREATERP N
        FINALLY FAILURE();
```

Calling CHOICEUO) will give ten choices. The initial domain is just the integer 1. The value function is the identity function

(LAMBDA (3) 1).

The successor function is addition by one

(LAMBDA (I) (PLUS I D).

The termination condition is a check if the maximum has been exceeded

(LAMBDA (I) (GREATERP I N)).

The termination function

(LAMBDA (I) (FAILURE))

propagates the failure if the termination condition becomes true. This illustrates the use of the intrinsic function FAILURE, 3 function of no arguments that fails to the last decision point set and restores the state of the computation at that point.

(2) The most common use of SELECT is to select items one at a time from a list, and try out each item selected. For this reason, several of the clauses in the SELECT expression syntax are optional. The following two forms are equivalent.

```
X ← SELECT FROM '(A B C D E);

X ← SELECT CAR(L) FROM L:'(A B C D E)
        SUCCESSOR CDR(L)
        UNLESS NULL(L) FINALLY FAILURE();
```

The FINALLY expression need not propagate failure; it may pass information back to earlier contexts, or simply compute a final value.

(3) One of the most interesting uses of the SUCCESSOR expression, which computes a new domain, is to reorder the old domain based on the information gained by trying the last alternative. QA4 and other systems have this ability.

```
GOAL ← SELECT TOP_PRIORITY(GOALS)
        FROM GOALS: INITIAL_GOALS()
        SUCCESSOR REORDER(GOALS)
        UNLESS TIME_TO_STOP(GOALS)
        FINALLY TRY_SOMETHING_ELSE();
```

Here the functions all operate on a goal list, represented by the variable GOALS. The successor function modifies this variable, in a way that may be influenced by the last alternative tried (for example, by passing some information in context), thus affecting subsequent alternatives.

Other features

Two final backtracking features available in MLISP2 are a function called FLUSH, which prunes a part of the goal tree by flushing elements off of the state stack, and a notation for setting variables in other backtracking contexts. Normally assignments such as "X ← Y" take effect in the current context, but "X{10} ← Y" sets X to the value of Y in the context 10. X will now not be backtracked until a failure to context 10 occurs or until it is again set in the current context. This enables the programmer to specify that information is to be saved until a certain context is destroyed. Any expression to compute a context number may occur inside the braces.

REP, OPT, ALT, SELECT, FAILURE, FLUSH and setting in context constitute the complete set of backtracking facilities available in MLISP2. Note that unlike Floyd's theoretical system, there is no SUCCESS function; success is the absence of a failure, just as TRUE is anything but NIL in LISP. These primitives are slightly less powerful than those available in some other systems, particularly PLANNER and ECL which allow failing to a label. For example, in ECL the programmer may declare TAG points, which are like nondeterministic labels, and then fail explicitly to these TAG points. However, although they are very powerful (any control structure can be be buiit up out of GO TO's), the arguments against the use of GO TO's in deterministic languages apply as well to the use of these nondeterministic GO TO's and labels. Just as many languages are trying to replace GO TO's with WHILE- and FOR-loops, we have tried to replace nondeterministic GO TO's with a nondeterministic FOR-loop; the SELECT expression.

Summary

An efficient implementation of backtracking used In the MLISP2 language has been described. The efficiencies are due to a smooth integration of backtracking into an existing LISP system; in particular, the way variables are stored has not been changed, so that fetches and most stores are not degraded. The theory of existing backtracking systems has been related to two ways of viewing the structure of computations. MLISP2 has a "state-saving" structure, as opposed to a "sequential" or "inverse statement" structure. Other backtracking systems have been discussed and compared with the MLISP2 implementation. Finally, the language features available in MLISP2 for using backtracking have been presented. Our main conclusion is that it is possible to incorporate backtracking into a production system in such a way that the most frequent operations are not degraded in performance. Backtracking is a more useful and more used control structure when this ie done.

With some simplifications and omissions, the existing implementations of backtracking are summarized in the following table.

METHODS AUTOMATICALLY RESTORED	Sequential (inverse statements)	State (state saving and restoring)
Control environment	PLANNER	ECL (SAIL)
Control and data environment	Floyd	MLISP2 QA4 POP-2

Bibliography

1. Bobrow, D.G. and Wegbreit, B. A Model and Stack Implementation of Multiple Environments Report No.2334, Bolt, Beranek and Newman, 1972.

2. Bur stall, R.M., Collins, J.S. and Popple stone, R.J, Programming in Pop2, University Press, Edinburg, Scotland, 1971, 279-282.

3. Feldman, J.A., Low, J.R., Swinehart, D.C. and Taylor, R.H. Keoent Developments in SAIL, An ALGOL based Language for Artificial Intelligence Artificial Intelligence Project Memo AIM-176, Stanford University, 1972.

4. Fikes, R.E. A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures PH.D. Thesis, Carnegie-Mellon University, 1968.

5. Floyd, R.W. "Nondeterministic Algorithms" J.ACM 14, A (Oct. 1967), 636-644.

6. Golomb, S.W, and Baumert, L.D. "Backtrack Programming" J.ACM 12, 4 {Oct. 1965), 516-524.

7. Hewitt, C. "Procedural Embedding of Knowledge in PLANNER" Proc. IJCA1 2, 1971, 167-182.

8. Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot" AI Memo 168 (rev), MIT, 1970.

9. Milner, R. Logic for Computable Functions Description Artificial Intelligence Project Memo AIM-169, Stanford University, 1972.

10. Prenner, C.J., Spitzen, J.M., and Wegbreit, B. "An Implementation of Backtracking for Programming Languages" Sigplan Notices 7, 11 (Nov. 1972), 36-44.

11. Quam, L.H. and Diffie, W. Stanford LISP 16 Manual At Operating Note 28.7, Stanford University, 1972.

12. Rulifson, J.F. QA4 Programming Conoepts AI Technical Note 60, Stanford Research Institute, 1971.

13. Smith, D.C. and Enea, H.J. MLISP2 Artificial Intelligence Project Memo AIM-195, Stanford University, 1973.

14. Sussman, G.J. and McDermott D.V. "Why Conniving is Better Than Planning" Proc. FJCC 41 (Dec. 1972), 1171-1180.