Backtracking with Generators

John T. Korb

TR 78-5

April 10, 1978

Department of Computer Science

The University of Arizona

*This work was supported by the National Science Foundation under Grant MCS75-01307.

Backtracking with Generators

1. Background

This paper describes work in progress on new features for high-level programming languages to support string and structure processing. These facilities are being designed as part of Icon, a new programming language [1]. This language has its roots in SNOBOL4 [2] and SL5 [3,4], and may be considered an extension of these languages. Design goals include the development of a practical tool for string and structure processing that includes many of the novel features existing in SNOBOL4 and SL5.

Most of the features described in this paper grew out of the need for language facilities to support backtracking, especially backtracking used in string scanning [5]. Their utility, however, is not limited to backtracking contexts.

2. Backtracking

Backtrack programming is a problem solving technique that allows tentative decisions to be made and later reversed if it is found that a different decision would be better. A mathematical formalization of this technique can be found in [6]. Such a trial and error approach provides a useful tool for solving certain types of problems. For example, it can be used to exhaustively enumerate solutions to certain combinatorial problems. A tree is used to represent the decisions to be made in solving the problem. A node in the tree represents a decision point, branches emanating from the node correspond to potential decisions at that node. When solving this type of combinatorial problem, a decision is made at each node, descending deeper into the goal tree. Eventually, a leaf node is reached, which may be either a dead end or represent a correct solution. A dead end is a node of the goal tree at which there is either no available decision, or it is known that no decision will lead to a correct solution. If a dead end is encountered, backtrack to a preceding node, make a different decision, and continue descending the tree. This process is continued until a set of decisions are found that lead to a leaf node that is a solution.

When performing long searches in a deep decision tree, early detection of a "blind alley" is important. This eliminates testing a large number of incorrect solutions. For smaller goal trees, the ability to avoid blind alleys is not as important. Exhaustive enumeration with a final test after each iteration may be fast enough.

A problem that is easily solved using a backtracking strategy is the eight queens problem [7]. The problem is to place eight queens on a chess board such that no two of them can capture the other (i.e. are not on the same row, column, or diagonal). To solve this problem, first observe that exactly one queen must appear in each column. Work left to right across the board, placing one queen in each column on a row that doesn't conflict with any of the previous queens. Begin by placing a queen on column one, row one. Next, place a queen on column two, noting that the lowest row that it can be placed on is row three. Continue placing queens on the board until there is no safe row on which to place a queen. Backtracking must now be done; a decision made earlier was incorrect. Retreat to the latest decision made - placement of the last queen - and move that queen to a different row. If no safe row remains for that queen, backtracking must be continued. Once a safe move has been found, continue placing queens on columns in the forward direction again. This back and forth placement and removal of queens will eventually lead to a solution (in fact, to all solutions).

Solutions to the eight queens problem can be generated very quickly, despite the fact that there are a huge number of ways to place eight queens on an eight by eight board. The reason for this speed is that many incorrect solutions can be easily eliminated without descending too deeply in the decision tree. A technique for estimating the running time of backtrack programs by random exploration of the goal tree has been developed [8].

One of the principal advantages of the backtracking approach to a solution is that it does not require a rigorous analysis of the problem. No elaborate study of the eight queens problem, or of chess for that matter, was necessary to find a solution to the problem. One observation is made immediately (and almost intuitively) that exactly one queen appears in each column. This provides a point of reference for the placement and removal of each queen. The rest is done through backtracking and following the problem statement.

3. Backtracking Facilities

Backtracking facilities are provided in several different languages, for example, SNOBOL4, SL5 [9], PLANNER [10], and MLISP2 [11]. This section describes some of the differences in the facilities provided by these languages. Other than syntactic issues, languages differ in three fundamental ways in their handling of backtracking, (1) initiation of backtracking, (2) scope of backtracking, and (3) reversal of effects.

An important consideration in the design of backtracking facilities is how to initiate backtracking. When it has been determined that an incorrect decision was made, how does the programmer communicate to the system that it is time to backtrack? One technique is to provide a function, such as FAIL(), that may be called to initiate backtracking [12]. This causes control to be transferred to the last (most recent) decision point.

In signal-driven languages, such as SNOBOL4 and SL5, a failure signal provides a convenient method to initiate backtracking. During expression evaluation, decisions are made until some subexpression fails. The system then returns to the most recent decision point and begins backtracking. In this way, any conditional expression or programmer-defined procedure may initiate backtracking simply by failing. In addition to being a more flexible way to initiate backtracking, in some cases this provides a de-emphasis from backtracking itself that is more natural to use. Using a FAIL function requires a two-step approach to backtracking: first determine that backtracking is necessary, then initiate it. With failure signals, the decision and initiation are combined into one step.

The scope of backtracking determines how far failure propagates. If a decision is made, what determines when that decision can no longer be reversed? In some languages, a decision may always be reversed. That is, after a decision is made, an arbitrary amount of execution may transpire before the decision is reversed. This has the unfortunate property that the flow of control from failure to decision point may be an unexpected jump. Languages supporting this unlimited scope of backtracking, for example, PLANNER, have been criticized for not providing the programmer the necessary facilities to prevent unwanted backtracking [13].

Another approach is to allow the decision to be reversed only if failure is detected in the same expression. That is, once the execution of an expression is completed, remaining alternatives within that expression are discarded. This restricted scope of backtracking is used in the language facilities described in this paper.

During backtracking, the state of execution is, in some sense, reset to the last point at which a decision was made. It is then up to the program to make a different (hopefully better) decision. The degree to which the state of program execution has been restored varies among languages. Some systems restore all variables to the values they had at the time the decision was made. In some cases, a user-selected subset of those variables is restored, while still other systems provide no restoration of variables.

For the types of problems intended for the programming language being described here, full restoration of the state of execution is not necessary. This eliminates the overhead associated with both retaining old values and resetting them during backtracking.

As the search progresses down a blind alley, valuable information may be gained that can be used to modify subsequent decisions. In programming languages that effect full restoration, these results are lost when the values of variables are reset.

4. Language Facilities for Backtracking

This section describes language facilities that are being incorporated into the Icon programming language. These facilities are designed to support backtracking for string scanning and similar applications.

4.1 Generators

The principal method of providing alternatives, or choices, is through a generator. A generator is an expression that computes a value during forward computation, but when reentered during backtracking produces subsequent (or alternate) values. For example, one of the simplest generators is the "or" operator ("|");

$$e_1 \mid e_2$$

The expression e_1 is evaluated first and if that evaluation succeeds, the result is the result of the expression $e_1 \mid e_2$. If e_1 fails, e_2 is evaluated and its result is the result of the entire expression. The utility of this expression as a generator occurs during backtracking. If the expression initially evaluated to e_1, subsequent values are generated from remaining alternatives in e_1. If no alternatives remain in e_1, the expression e_2 is evaluated, and alternatives from it become the value of the entire expression. The expression

```
x | y
```

evaluates to x during forward computation, but during backtracking evaluates to y. These generators may be nested, as in the following expression.

```
("hello" | "hi") | "howdy"
```

The string "hello" is the first value of the expression. During backtracking, the second alternative from e_1, "hi", is generated. Finally, a subsequent request for alternatives produces the string "howdy".

Generators can appear anywhere a value is expected. During normal computation, generators are transparent to the remainder of the system. The values produced are used just like any other computed value, for example, as arguments to functions or operators.

```
f(x|y) > 100

greeting == "hello" | "hi" | "howdy"
```

The latter expression compares the value of the identifier greeting to each of the given strings.

4.2 Activating a Generator

In order to use a generator, there must be some way of extracting values from it. This is termed activating a generator. A generator produces one value during forward computation. If another value is required during backtracking, the generator is activated, requesting a subsequent value.

The built-in expression evaluation mechanism automatically activates any expression that fails, thus attempting to convert failure into success. If an expression fails during evaluation, it is examined for possible alternatives. If alternatives remain untried, evaluation is backtracked to the most recent decision point and restarted. This activating is continued until either the expression succeeds, or until it is exhausted, i.e., until no alternatives remain.

For example, in the if statement

```
if greeting == "hello" | "hi" | "howdy"
   then write("How are you?")
```

the value of the identifier greeting is compare first to "hello". If that succeeds the message is written. If it fails, the built-in mechanism intercepts the failure, and backtracks to the next iteration of the generator, producing the value "hi". If failure again occurs, greeting is compared to "howdy". If none succeed, the then part is not executed.

Various combinations of conditional expressions and generators allow novel comparisons:

```
if f(x|y) > 100 then g()

if (w|x) = (y|z) then f(w,x) else f(y,z)

if x < (y|z) < w then f(x,w)
```

Another example of a generator is the to expression.

e_1 to e_2

This generator produces a sequence of values from e_1 to e_2. For example, the expression

```
1 to 100
```

generates the integers from 1 through 100. The similarity between the to generator and the corresponding portion of the Algol for statement is not coincidental and will be described in the next section.

The to expression takes an optional by clause:

e_1 to e_2 by e_3

This allows an increment (or decrement) other than 1 to be used.

4.3 Exhaustion

A generator is exhausted when it has generated all of its values. One way to force the exhaustion of generators is with the every statement:

```
every e
```

This statement evaluates e, producing its first value. As long as there remain untried alternatives, i.e., generators that have not been exhausted, the every statement activates them to produce subsequent values. For example, if a is an array of 100 elements, the two statements below write the specified elements of the array.

```
every write(a[1 to 100])

every write(a[1 to 10 | 91 to 100])
```

The statement

```
every enter("begin" | "end" | "every")
```

calls the enter function three times, once with each string as argument.

The every statement has an optional do clause.

every e_1 do e_2

After each generation in e_1, the expression e_2 is evaluated. Using the to generator gives the for statement of Algol:

```
every i := 1 to 100 do a[i] := b[i]
```

The flexibility of the every statement is much greater than the Algol for statement. It does not simply iterate a control variable over an arithmetic sequence. There is not the restrictive and clumsy from, to, by, while, and until parts. Instead, a larger variety of testing and generation may be done.

The every statement is more concise in certain applications. The control variable in many cases is just an artifact and can be omitted. The statement

```
every 1 to 50 do f()
```

invokes the procedure f fifty times. In other cases, only a few values need to be iterated over, and they need not be integers.

```
every cond := "mildew" | "fungus" | "clean" do init(cond)
```

4.4 Mutual Success

The and operator

e_1 & e_2

is used to force two conditions (or expressions) to be mutually satisfied. It evaluates its left operand, if that succeeds it evaluates its right operand. If the right operand fails, and has no alternatives, the left operand is examined for alternatives, and if it has any, they are evaluated.

As an example, the expression

```
every i := 1 to 100 & p(i) do f(i)
```

performs the function f on each of the integers from 1 to 100 that satisfy the condition p(i). Of course, the & may be used in traditional testing situations where a simple boolean conjunction is required.

```
if x = y & y < 5 then ...
```

4.5 Scope of Generators

The scope of a generator is defined to be the expression in which it appears. This provides a uniform and simple mechanism that eliminates the uncontrolled backtracking and concomitant processing that exist in some other languages. The programmer has more precise control over the backtracking.

For example, in the if statement

if e_1 then e_2

if e_1 succeeds, e_2 is evaluated. If e_2 fails, any remaining alternatives in e_1 will not be tried. Compare this with the following "and" expression.

e_1 & e_2

If e_1 succeeds, e_2 is evaluated. Unlike the if statement, however, failure of e_2 results in examination of e_1 for alternatives. There may be considerable interaction between the two expressions, with control flow alternating between them.

On the contrary, in the expression sequence

e_1; e_2

failure in any portion of expression e_2 does not result in backtracking to e_1.

There is a similar correspondence between the expression

e_1 | e_2

and the statement

    _if_ e_1 _fails_ _then_ e_2.

In the former case, if e_1 initially succeeds, e_2 _may_ be evaluated during backtracking. On the other hand, the _if_ statement insures that if e_1 succeeds, e_2 will _not_ be evaluated.

4.6 Programmer-Defined Generators

Programmer-defined generators are allowed, using the procedure mechanism and the _suspend_ statement. The _suspend_ statement

```
suspend e
```

suspends the execution of the current procedure and returns the value computed by e. The state of execution is preserved so that during backtracking, the procedure may pick up where it left off. Control returns to just after the _suspend_ statement.

For example, the following procedure generates a sequence of Fibonacci numbers less than n

```
procedure fib(n)
   private ul, u2, fib
   suspend 1
   suspend 1
   ul := 1
   u2 := 1
   repeat {
      fib := ul + u2
      suspend fib
      ul := u2
      u2 := fib
      } while fib < n
   fail
end
```

The sequence is generated by the statement

```
every write(fib(n))
```

4.7 Generators in String Scanning

Built-in generators are provided for string scanning. Typically, a string scanning generator performs a simple lexical analysis of a string (or substring). During backtracking, the generator extends its domain, performing an expanded version of its initial analysis.

For example, the upto generator is used to locate the first occurrence of a character in a string (analogous to BREAK in SNOBOL4). During backtracking, subsequent occurrences are located.

```
        upto(c,s,i,j)
```

locates occurrences of characters in c in the subject string s between character positions i and j. Character positions in Icon are numbered starting with 1. In addition, non-positive values specify character positions from the right end of the string. Thus, a zero (or omitted) value refers to the right end of the string, -1 refers to the last character, etc. Thus,

```
        upto("aeiou","phlegmatic",1)
```

looks for vowels in the entire string "phlegmatic". The first value generated is 4. If the generator is reactivated, the values 7 and 9 are generated.

Other examples of generators for string scanning include find(t,s,i,j) and thru(c,s,i,j). find locates the first occurrence of a string t in the subject string s between character positions i and j. If alternate values are required, subsequent occurrences of t are located. thru finds the position of the first character in s (starting at position i) that is not in c. During backtracking, this value is decremented, reducing the substring spanned. For example,

```
        x := thru("aeiou",s,1) & find("eek",s,x)
```

succeeds if s contains a string of vowels followed by an occurrence of the string "eek".

5. Conclusions

Backtracking is a useful technique for the solution of certain types of combinatorial problems. The particular method of backtracking provided by the language considerably affects the utility of backtracking, and the problems for which it is suitable.

Different types of generators are used to produce different sequences, or types of sequences, of values during backtracking. In addition, generators may also be used in situations that do not conform to what is usually considered to be backtracking.

The facilities described here have been designed for use in string scanning. In this application, the scope of backtracking is limited, and the nesting of generators is not very deep. This simplified approach appears to be the right mix of useful power and structured control.

6. Acknowledgments

I am indebted to Ralph E. Griswold and David R. Hanson for many of the ideas that appear in this paper. Their critical reading of drafts of this paper and helpful comments is also appreciated.

References

1. Griswold, Ralph E., David R. Hanson, and John T. Korb. The Icon Programming Language; a Preliminary Report. Technical Report TR 78-3, Department of Computer Science, The University of Arizona, Tucson, Arizona. April 10, 1978.

2. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

3. Griswold, Ralph E., David R. Hanson, and John T. Korb. An Overview of the SL5 Programming Language. SL5 Project Document S5LD1d, The University of Arizona, Tucson, Arizona. October 18, 1977.

4. Griswold, Ralph E. and David R. Hanson. "An Overview of SL5", SIGPLAN Notices, Vol. 12, No. 4 (April 1977), 40-50.

5. Griswold, Ralph E. An Alternative to the Concept of "Pattern" in String Processing. Technical Report TR 78-4, Department of Computer Science, The University of Arizona, Tucson, Arizona. April 10, 1978. Submitted for publication in Communications of the ACM.

6. Golomb, Solomon W. and Leonard D. Baumert. "Backtrack Programming", Journal of the ACM, Vol. 12, No. 4. (October, 1965). pp. 516-524.

7. Wirth, N. "Programming Development by Stepwise Refinement", Communications of the ACM, Vol. 4, No. 4 (1971). pp. 221-227.

8. Knuth, Donald E. "Estimating the Efficiency of Backtrack Programs", Mathematics of Computation, Vol. 24, No. 129 (January, 1975). pp. 121-136.

9. Griswold, Ralph E. and David R. Hanson, "Language Facilities for Programmable Backtracking", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8 (August, 1977). pp. 94-99.

10. Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot", *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 2 (1971). pp. 167-182.

11. Smith, David C., and Horace J. Enea. "Backtracking in MLISP2", *Proceedings of the Third International Joint Conference on Artificial Intelligence* (August, 1973). pp. 677-685.

12. Prenner, C. J., J. M. Spitzen, and B. Wegbreit. "An Implementation of Backtracking for Programming Languages", *SIGPLAN Notices*, Vol. 7, No. 11 (November, 1972). pp. 36-44.

13. Sussman, G. J. and D. V. McDermott. "From PLANNER to CONNIVER -- a Genetic Approach", *Proceedings of AFIPS 1972 Fall Joint Computer Conference*, Vol. 41, pp. 1171-1179.