# E. PATTERN MATCHING

## 1. Traditional

Traditional pattern matching languages, like SNOBOL4, represent patterns as an abstract data type. A special and specific pattern evaluation routine traverses both the pattern structure and the subject text, applying evaluation rules as indicated by the pattern, and advancing or regressing depending on their outcome.

### a) Representation of a pattern

The pattern's data structure must include concatenation and alternation, the specification of the test to be applied at any point in the pattern, and the test data such as text "literals." One such representation is described in [WAI73] and is illustrated in figure E-1.

The pattern is represented as a complexly linked list of pattern elements of the form shown in E-1(a). Each element has two links; one is the "successor" link, indicating a concatenation, and the other is the "alternative" link, indicating an alternation. The pattern element also includes the address of an "element processor," which is a routine which specifies the test to be performed. There may also be a variable amount of information for the use of element processor.

Figure E-1(b) illustrates a simple pattern involving both alternation and concatenation. Figure E-1(c) shows how the linkages in the pattern elements represent this pattern.

### b) Pattern matching routine

There is a "procedural" element to this representation. The test to be applied at any point in the pattern is indicated by the address of a processing routine. This has two advantages: first, flexibility, since any number of processing routines may be defined; second, speed, since in most cases it is faster to invoke a routine directly by its address than by indexing into a table. (Programming languages which do not support pointers to functions would be limited to a fixed set of element processors, selected by a multi-way "case" construct.)

The pattern itself is not an executable structure. A controlling routine, or "pattern engine," must both traverse the pattern and invoke the elemental tests. Such a global control routine has been described in [WAI73,SOL85]. Waite's algorithm is shown in Figure E-2.

These are the key steps in the processing of one element of a pattern:

1. Invoke an element matching routine.

2. If it is successful, examine the pattern for a successor.

    2a. If there are no successors, the pattern is complete and the match was successful.

    2b. If there is a successor, get its element processor and return to step 1.

3. If the element routine was not successful, examine the pattern (via the history stack) for either an immediate alternative, or, if none such exist, for an untried alternative farther back in the pattern.

    3a. If there are no alternatives, the options are exhausted, and the match is unsuccessful.

    3b. If there is an alternative, restore the scan position to the point at which the alternative applies, get the alternative's element processor, and return to step 1.

The scan position (the position in the subject text string) is not advanced by the control routine. Since different element processors may match varying amounts of text, the element processors themselves should advance the scan pointer by the appropriate amount.

It would be possible to include this pattern traversal logic in each element processor. But this would be a needless duplication of this logic, which is common to all elements in the pattern.

### c) The history stack

A pattern matcher must be able to "backtrack" to untried alternatives. Consider, for example, a choice between two alternatives encountered very early in the pattern. The first alternative is chosen, and a long series of successors execute successfully. Then, a successor element fails, ending the successful match. This may not invalidate the pattern, since it is conceivable that, had the second alternative been chosen, all of its successors would have been successful.

The verdict of "failure" cannot be pronounced as long as any alternative remains untried. So, the pattern matcher must keep track of all untried alternatives.

This is actually two problems: "remembering" all of the untried alternatives which remain to be evaluated, and deciding in which order to return to them.

A convenient solution to both of these problems is to keep the untried alternatives on a stack. The stack is a dynamic data structure, which can expand as new alternatives are added, and contract as old alternatives are attempted. The "last-in-first-out" (LIFO) nature of the stack means that the most recently encountered alternative will be the first attempted on a failure. This agrees with our "intuitive" processing order for alternatives, attempting "short" backtracks before "long" backtracks.

The "backtracking" logic of the global control routine in Figure E-2 can be summed up:

1. If ever an element with an alternative is encountered, "push" the information about that alternative onto the stack.

2. If ever the pattern match fails, "pop" an untried alternative from the stack and resume the pattern match there.

This approach will exhaustively attempt all possible combinations of alternatives in the pattern. Only when the stack is empty are there no untried combinations, and only then can the pattern match be deemed to have failed.

Since this stack contains information about "past" pattern matching decisions, it is called the "history stack."

## 2. Executable patterns

This traditional pattern control logic is, in many respects, an interpreter for a pattern "language." And, in common with other language interpreters, it adds an interpreter overhead. Is it possible, implementing this pattern "machine", to improve efficiency by using a compiled language rather than an interpreted language?

In other words: is it possible to cause the execution mechanism of the CPU to traverse the pattern, and implement the pattern control logic? Or, can the pattern itself be made an executable structure?

Such a scheme has been previously employed in parsers [ROD89a], and is now employed in patternForth.

One reason for adding patterns to Forth in this way is the ease of creating a compiler. To produce the data structures of Figure E-1, from, say, SNOBOL4 source code, requires quite a complex compiler. By following the "natural" control structures of compiled Forth, and exploiting Forth's support for them, the program to create the executable pattern from the source code is made quite simple.

This compiler is embodied in the patternForth word set. It outputs high-level Forth code, which can be executed on any Forth machine. "High-level" Forth code is a "thread," or list, of executable routines, typically represented by their addresses. This is executed by the Forth "address interpreter," so that Forth itself is considered an interpreted language. However, the interpreter for a threaded language is vastly more efficient than any other language interpreter, adding on some CPUs an overhead as little as 20% [JAM78]. And, the newer "subroutine-threaded" Forths [KOG82] and Forth processors [JEN85,HAR88] have no interpreter overhead at all, and produce code which is equivalent to compiled code (somewhat blurring the distinction).

PatternForth has been written on an IBM PC using a "conventional" indirect-threaded-code (ITC) Forth. The illustrations of compiled code, later in this section, are for an ITC system.

## 3. Element representation

Four elements are necessary in a pattern element: the element processor (identifying the test to be performed), optional information to be supplied to the element processor, an alternative, and a successor.

**a) element processors**

In the threaded Forth representation, the address of the appears directly in the object code. The Forth "address interpreter" executes it. The Forth instruction stream for a single element is shown in Figure E-3(a).

The element processing routine is defined as a normal Forth "word" (subroutine). Forth words may be written in "high-level" Forth, or in the processor's native machine code.

There are two conventions which apply to all element processors. First, each element processor is responsible for advancing the scan pointer. Second, each element processor returns, on the Forth "parameter stack," a success flag; true if the match was successful, false if not.

**b) data for element processors**

Some element processors, such as the literal text matching routine, require additional information to be stored with the pattern element, i.e., "parameters." This is done in same manner that parameters are passed to "ordinary" Forth routines. Two techniques are common:

First, parameters may be passed on the stack. A previous Forth word must have left this data on the stack, as indicated in Figure E-3(b). This is most commonly used in Forth to pass numeric parameters. PatternForth also passes text data in this way, in the form of string descriptors on the stack.

Second, parameters may be included "in-line" in the object code, as illustrated in Figure E-3(c). The element processor must advance the Forth instruction pointer past such in-line parameters. This method is commonly used in Forth to pass string data; it is not used (yet) in patternForth.

## 4. Concatenation of successors

The obvious representation of a successor element is as a successive subroutine call, and a chain of successors as a series of subroutine calls. To this end, we cast the global control logic previously described into the form of a "concatenation operator" which links two successive elements. This operator must perform the following:

If the element just processed was successful, allow execution to continue sequentially to the following (successor) routine. If the element failed, invoke the backtracking logic (to be described).

This is implemented as the Forth word (>>). A sequence of three successors is illustrated in Figure E-4.

(This is not an exact description of the word (>>). This word is a byproduct of the alternation logic, to be described next; it must perform additional functions as well, and will be balanced with the (<<) word.)

## 5. Alternation

The logic for simple alternation (without backtracking) is inspired by Forth's "natural" implementation of a choice between alternatives, the IF...ELSE...THEN construct.

IF the condition is true, allow execution to proceed sequentially to the first clause; otherwise, perform a forward branch to the second clause.

Neglecting backtracking for the moment, consider a similar "alternation operator" could be written:

If the element just processed failed, allow execution to proceed sequentially to the following (alternative) routine. If the element succeeded, perform a forward branch over all remaining alternatives.

This is implemented as the Forth word (|), and is illustrated in Figure E-5 in an alternation of three elements. Such an operator was described in [ROD89a]. The problem of resolving multiple branches is addressed in [EAK80].

Since all of the alternatives apply to the same subject text, the scan pointer must be reset to that point when each new alternative is tried. One way to do this is to require each alternative to advance the scan pointer only on success, and to leave it undisturbed on failure. A more advantageous approach is to have the alternation logic save and restore the scan pointer.

The group of alternatives must behave like a pattern element, and return a flag on the stack. The (|) operator must preserve the success flag on the stack when branching to the end of the sequence. The final alternative, if executed, may leave either a success flag, or a failure flag which would indicate that no alternative was successful.

To compile a branch in Forth, we need an operator at the end of the last alternative, to provide the address for the forward branches. Syntactically, we need an "end" statement to delimit the scope of the control structure. The compiler directive which performs this function is >> . This word compiles the same (>>) used for concatenation, as it turns out that the same functions are required.

## 6. Grouping: Mixing concatenation and alternation

All of the element information illustrated in Figure E-1 can be represented "procedurally." We now turn to the problems posed by their inclusion in complex patterns.

Two constructions commonly encountered are a succession within an alternation, in which an alternative is itself composed of several elements in succession, and an alternation within a succession, in which one element in a succession is itself an alternative choice of several elements.

These are two manifestations of "grouping" within a pattern, the pattern equivalent of parentheses in arithmetic expressions. The two "connective" operators, concatenation and alternation, can appear in four possible groupings. The groupings of an alternation within an alternation, or a succession within a succession, are trivial. So, only these "mixed" constructions are of concern.

Figure E-6 illustrates how our definition of the concatenation and alternation structures allows them to be "nested" in either order.

Figure E-6(a) shows a succession of three elements, as one choice in an alternation. Observe that:

a) if the first alternative succeeds, the succession (as second alternative) is completely skipped;

b) if the succession is entered, and all of its elements succeed, the second alternation operator (|) is entered with "success" on the stack, and the third alternative is skipped;

c) if any element of the succession fails, the sucession operator will backtrack.

Figure E-6(b) shows a three-way alternative choice, as part of a succession. Observe that:

a) if any alternative succeeds, control is transferred to the succession operator (>>) with "success" on the stack;

b) if all the alternatives fail, control is transferred to the succession operator with "failure" on the stack, and the succession fails.

Note that the forward branches from the alternation operators in Figure E-6(b) terminate on a succession operator (>>). We arbitrarily proclaim that all alternations must exist only within successions, since an alternation standing alone may be converted to the trivial case of an alternation within a succession of one element. So, all alternations terminate on a succession operator. This operator is both the terminator for a sequence of alternatives, and the concatenation operator for a series of successors.

An extension of these examples shows that this "nesting" of succession and alternation can occur to any depth.

## 7. Backtracking

We have carefully avoided describing the backtracking logic used by the concatenation operator, other than to assume the existence of some mechanism by which this operator can "jump back" to a previous alternative. It is now necessary to address this issue in greater detail. A further examination of Figure E-6(a) and Figure E-6(b) will show why.

Consider again possibility (c) described for Figure E-6(a). If the second alternative (the succession) fails, backtracking is invoked. The third alternative is never tried! But, since the succession is simply one of three alternatives, its failure should not disqualify the other alternatives.

And, consider failure in the third element of the succession of Figure E-6(b). If the concatenation operator backtracks to some previous (not illustrated) point in the pattern, and one of or more of the alternatives in Figure E-6(b) had been skipped, these alternatives will never be attempted.

We solve both of these problems by keeping a record of the most recent alternative branching point, and causing the "failure" backtracking logic to return to that point and take the "other" branch. Since there may be several such alternatives which could

be taken on failure, and we always need the most recent, a Last-In-First-Out stack is used for storage. This is the "History Stack" in patternForth.

For a multiway alternative (3 or more alternatives), it is only necessary to record the fact that at least one alternative remains untried. If alternative #1 succeeds, the "alternative branching point" would be alternative #2. If a later backtrack causes a return to alternative #2, and it succeeds, its "alternative branching point" is alternative #3. See Figure E-7.

Note that alternative #3 is stacked immediately after alternative #2 was "unstacked." All of the alternatives in this multiway branch will be recorded at the same stack position, and, thus, all of these alternatives will be attempted at the same "depth" -- after any "later" alternatives have been tried, but before any alternatives from earlier in the pattern are tried.

Logically, this decomposes the multiway alternative

```
( ... | ... | ... | ... )
```

to a series of binary alternatives

```
((( ... | ... ) | ... ) | ... )
```

## 8. The History context

The most important item of information to be stored on this "History" stack is this "return" branching point. It is not necessary to re-evaluate the branch decision which was made by the alternation operator; as Figure E-7 illustrates, the backtrack should "fall through" the operator to execute the element code which immediately follows. We store the address of this following code on the History stack.

When backtracking to an alternative, we must also restore the scan pointer to its position in the subject text where the alternative was encountered. This position may not be known -- any number of other elements may have advanced the scan pointer. So, this too must be kept on the History stack. Since we need to know the value of the scan pointer before the first alternative was attempted, we prefix a new operator, (<<), to save the scan position.

(<<) and (>>) also bracket individual elements in a succession. This removes the task of saving the scan pointer, and restoring it upon failure, from the element routines.

Some "private" context information may need to be shared among the alternatives. For example, the logic to match an arbitrary number of characters can be implemented as a single element in an alternative "loop;" requiring a loop counter to be preserved across the "alternatives". This information can be stored "deeper" in the History stack, but it is the responsiblity of each alternative to remove any private context data stacked by the previous alternative.

This works because the alternative which removes this data cannot be executed until the stack is "popped" down to where its context data was stacked; and no further History "pops" will take place until after this alternative executes.

The History stack is kept in Forth's "data" (or "parameter") stack. This makes the storage of private context data logical to the programmer, since this stack holds the "execution context" of, and passes parameters to and from, Forth routines. (Storing History on Forth's "other" stack -- the Return stack -- would require great care to avoid interference with Forth's execution mechanism. Also, we will soon see that the Return stack is required for a different function.) Figure E-8 illustrates how the backtracking context is stored on the Forth data stack.

## 9. Subpatterns

Patterns are implemented as executable Forth "words." One of the greatest strengths of Forth is that newly-defined words can be used, in turn, within other words. Can a "pattern word" be used as an element within another such pattern word? Or, can "subpatterns" be defined and used?

This requires an intimate knowledge of the Forth call and return mechanism. The execution of a high-level Forth word involves four steps:

a) the current IP (Instruction or Interpretation Pointer) is pushed onto the Return stack;

b) the IP is loaded with the new execution address;

c) the word executes normally;

d) the "return" IP is popped from the Return stack.

Machine-language Forth words use a slightly different procedure, but the net effect is equivalent.

The state of the return stack throughout this process is shown in Figure E-9. During the execution of the subroutine, the top element of the Return stack holds the address where execution will resume at the conclusion of the subroutine.

The Forth execution mechanism can execute a subpattern word within a "main" pattern. The "subpattern" must obey the rules for a pattern element, and return a success code on the stack.

During the execution of the subpattern, the top of the Return stack holds the address in the "main" pattern where processing is to resume. This adds one item to the execution context of a pattern: the return address.

This added context must be properly managed when backtracking. There are four possiblities:

**a) backtracking over a subpattern**

When backtracking from one point in the main pattern to another point in the main pattern, the Return stack context remains unchanged. The previously described logic is satisfactory.

**b) backtracking within a subpattern**

When backtracking from one point in the subpattern to another point in that subpattern, the Return stack context remains unchanged. This, too, presents no problem.

**c) backtracking into a subpattern**

A problem arises when a failure in the main pattern requires backtracking to an alternative which occurred in the subpattern. This is shown in Figure E-10 as example #1.

Suppose that a pattern failure occurs at the point marked "failure #1" in the main pattern, and the most recently stacked "alternative #1" is in the subpattern. The backtracking logic will restore the scan pointer and force execution to continue within the subpattern. When the subpattern exits, it will not find the return address for the main pattern on the Return stack!

We cannot push an "extra" return address on the Return stack, since this would disrupt the normal "forward" execution of the pattern. Also, the first backtrack after the subpattern may not be to the subpattern itself (it could be to "alternative #3").

What we need is some means of detecting when the backtracking logic is about to "back into" the subpattern, and, when this happens, pushing the return address on the Return stack. We do this by adding a special "backtracking record" on the History stack, immediately after the last alternative from the subpattern. When the pattern logic "backs up" through the History stack, this record is processed just before the subpattern alternatives. We refer to this as "falling (backwards) into" the subpattern. The special "fall-in record" contains the return address for the subpattern, to be pushed onto the Return stack when the record is processed.

When a backtrack takes place, an execution address is fetched from the History stack record. The "fall-in" record contains the execution address of the code to push the subpattern's return address onto the Return stack.

Figure E-11 shows the fall-in record. Its format is identical to a "normal" History record; the return address for the subpattern is its "private context." Note that, for consistent format, a superfluous copy of the scan pointer must be included. This value will be overridden, because the next backtracking record -- which specifies the backtrack into the subpattern -- is immediately popped from the stack.

The sequence of "falling into" a subpattern is:

(backtracking logic invoked by a failure)
* the fall-in IP is popped from the History stack;
* fall-in scan pointer is popped from the History stack;
* execution continues at IP

("fall-in" logic executes)
* the return address is moved from the History stack to the Return stack;
* a backtrack is forced.

(backtracking logic invoked)
* the backtrack IP is popped from the History stack;
* the backtrack scan pointer is popped from the History stack;
* execution continues at IP.

...which accomplishes the desired result.

A final refinement: it is only necessary to make provision for "falling into" a subpattern if there were alternatives put on the History stack by the subpattern. So, at the conclusion of the subpattern, the depth of the History stack is tested. If it is unchanged, no fall-in record need be pushed. (Actually, it is simpler to test whether or not the top History record is a "fall-out" record...to be described next.)

**d) backtracking out from a subpattern**

The converse problem is backtracking to a point in the main pattern, when a failure occurs in the subpattern. This is shown in Figure E-10 as example #2.

Suppose that a pattern failure occurs at the point marked "failure #2" in the subpattern, and the most recently stacked "alternative #2" is in the main pattern. It is necessary, before resuming execution at "alternative #2", to "pop" the subpattern's return address from the Return stack, restoring the Return stack to the state it had while executing the main pattern.

Again, this must only take place at the transition between the subpattern and the main pattern -- we might need to backtrack within the subpattern. And, again, this can be expeditiously accomplished by inserting a special record at the right place in the History stack -- after the History records from the main pattern, and immediately before any History records from the subpattern. As the pattern logic "backs up" through the History stack, this "fall-out" record is processed just before the return to a main pattern alternative.

There is a subtle trap in the logic for "falling out" to the main pattern. At first glance, one might think that the process parallels that of "falling in:"

("fall-out" logic executes)
* the top item on the Return stack is discarded;
* a backtrack is forced.

(backtracking logic invoked)
* the backtrack IP is popped from the History stack;
* the backtrack scan pointer is popped from the History stack;
* execution continues at IP.

But, as can be seen in Figure E-12, this may give incorrect results. This example shows a subpattern as the second of three alternatives. If the above logic is performed, a failure in the subpattern will cause a backtrack to a previous point in the main pattern (path A) -- instead of attempting the third alternative, which is the desired action (path B).

To cause the third alternative to be attempted, we cause the subpattern, when its "internal" backtracking options are exhausted, to report failure to the main pattern, i.e., to return "normally" with a failure flag. This allows the alternation operator (|) to act.

If there are no alternatives after the subpattern, the main pattern will see this as an element failure, and force a backtrack...to the next previous alternative in the main pattern, since the subpattern has removed all of its entries from the History stack! Thus, having the subpattern report "backtracking failure" will trigger an alternative or cause further backtracking, as appropriate, and exactly as required.

The revised "fall-out" procedure is:

(backtracking logic invoked by a failure)
* the fall-out IP is popped from the History stack;

* the fall-out scan pointer is popped from the History stack;
* execution continues at IP.

("fall-out" logic executes)
* "failure" is pushed on the data stack (as a return value);
* a normal Forth "return" is executed (IP is "popped" from the return stack);

(the main pattern may execute an alternative, or cause further backtracking)

Again, for consistency of record formats, a superfluous copy of the scan pointer is included in the "fall-out" record on the History stack, shown in Figure E-13.

We must push a "fall-out" record whenever a subpattern is entered. However, at the end of the subpattern, if there is no chance that the pattern will be re-entered -- if no alternatives were stacked by the subpattern -- this "fall-out" record can be discarded.

This technique, for "falling into" and "falling out of" subpatterns, extends automatically to any number of levels of nesting.

## 10. Deferred evaluation

"Deferred evaluation" allows part of a pattern to be specified at run time rather than compile time. This pattern fragment may be compiled during the execution of the program, or may be one of several pre-compiled patterns selected by run-time data.

We need to have a pointer to a subpattern, which can be changed during the execution of the program. The implementation of patterns as executable Forth words provides an immediate solution, in the form of the Forth word EXECUTE. This word takes the address of a function from the data stack (which, in turn, could have been supplied by a program variable), and executes it as a subroutine.

So, all that is necessary to "defer" a pattern element is to include a phrase such as

```
variable-name @ EXECUTE
```

where desired in the pattern, and to store the address of the subpattern in the given variable at run time. All of the logic for entering and exiting subpatterns operates normally.

Compiling a new pattern, at run-time, from a supplied input text, is slightly more complicated. Forth provides the programmer complete access to its "text interpreter" (the Forth compiler), and this can be used to perform Forth compilation -- and, hence, pattern compilation -- as part of an application program. An excellent example of this is given in [WIN82]. This function is used so often that the ANS Forth Committee has adopted a new word specifically for it [ANS89].

## 11. Recursion in patterns

High level Forth words which use only the stack for context are "naturally" re-entrant. The patternForth implementation of patterns and subpatterns preserves this property, and allows recursion within patterns.

The normal Forth mechanisms for recursion are used. Most Forth implementations provide a compiler directive, usually called RECURSE (sometimes MYSELF), which compiles the address of the word currently being defined into its own definition -- in other words, compiles a recursive reference to the word being defined [ANS89]. RECURSE can be used within pattern definitions to invoke the pattern as a subpattern within itself.

More complex recursion, involving more than one word, must be done with deferred evaluation. Forth in general provides no mechanism for forward referencing, although some Forths [LAX83] do provide a word called DEFER to support forward references.

## 12. The Top Level pattern

The implementations of subpatterns and the "main" pattern are identical: both are high-level Forth words with a "procedural" representation of the pattern. There is only one operational difference between a subpattern and a "main" pattern. A subpattern may succeed and leave untried alternatives on the History stack. (There may be more than one successful path through any pattern.) When a main pattern succeeds, the pattern match is complete, and any leftover History information can be discarded. If there is more than one successful path through the main pattern, by default, the first one encountered is used.

It would be possible to define two kinds of patterns, "sub" and "main," which embody this difference. Instead, to preserve the common format of all patterns, we use a special invocation sequence to identify the "main" or top-level pattern. This is the word EVALUATE, whose function is as follows:

> save the position of the data stack pointer;
> invoke the top level pattern;
> restore the position of the data stack pointer;
> push the success/failure result.

Only three possibilities may occur:

**a) the main pattern may fail**

> This occurs when all of the alternatives on the History stack are exhausted. In such a case, the data (History) stack is at its original depth, and the restoration of the stack pointer is superfluous.

**b) the main pattern may succeed, with no alternatives**

> In such a case, the History stack would be "empty," and the stack pointer would be at its original position. Again, the restoration is superfluous.

**c) the main pattern may succeed, with untried alternatives**

> The most likely scenario for success is with several untried alternatives on the History stack. (It is of no consequence whether or not these alternatives would succeed or fail if attempted.) From the viewpoint of the calling program, this history information is unneeded "garbage," once a successful match has been obtained. To discard this data we, restore the position of the data stack pointer to its position at the beginning of the pattern match.
>
> It is valid to act directly on the stack pointer to discard data, reducing the size of a stack. All of the stack information "underneath" the top is valid data. It would not be valid to increase the size of the stack in this manner. Fortunately, the only stack "cleanup" necessary is either to discard data, or to leave the stack unchanged.

## 13. Implications for Coroutines

It is instructive to compare this implementation of pattern matching to that of the Icon language, particularly regarding Icon's use of coroutines.

**a) use of coroutines in pattern matching**

> A coroutine is essentially a subroutine whose activation frame is not destroyed upon exit, and which can be re-entered with the contents of this activation frame (i.e., the local variables) preserved. Such a routine has two "entry points" or modes of entry: the "initial" entry, which sets up and initializes the activation frame, and the "resumption" entry, which uses the already-existing activation frame. There are also two possible returns from a coroutine: to preserve, and to destroy, the activation frame.
>
> Icon uses coroutines to implement generators, which in turn may be used to implement patterns. A "generator" is a routine which can return more than one result. The "result sequence" which can be returned by a generator is, in actuality, obtained by repeated calls to the generator coroutine. Successive "resume" calls return successive results, until no additional results are available.
>
> This, of course, is an obvious way to represent an alternation -- a matching routine with more than one result. In fact, the alternation operator is one of the fundamental operators used to write generators in Icon. Each successive resumption of a generator coroutine will return a successive alternative.
>
> Icon's "goal-directed evaluation" implements "control backtracking" [GRI83], which invokes generators using exactly the method described previously for pattern matching:
>
> > ...if the call of a function or operation fails, goal-directed evaluation resumes the last (rightmost) argument. If this argument produces no result, the previous argument is resumed, and so on. (p.119)

Observe the use of coroutines: when passing "forward" through the expression, each generator is invoked by its "initial" call. During control backtracking, a generator is invoked by its "resume" call.

Contrast this with the processing of "alternative sets" in patternForth. When an alternative set is first entered (by passing through the (<<) operator), the first successful alternative is "returned." When the alternative set is re-entered (by backtracking), successive alternatives are returned. The interim exits are performed by the (|) operator, and the final exit from the alternative set occurs when the (>>) operator is encountered.

### b) patternForth parallels to coroutines

The patternForth implementation of alternatives and backtracking, viewed in the Icon context, parallels the function of coroutines in several ways.

Realize first that, unlike most procedural languages (such as Pascal or C), Forth routines do not explicitly allocate an activation record for a routine. Instead, the space used on the data stack by the workings of the routine represents an "implicit" activation record. (The obvious corollary is that Forth uses a stack-like allocation of activation frames for its routines.)

When the alternation operator (|) preserves the "private context information" of an alternative, stacks a few values underneath it, and then goes on to activate a following word, it is in effect preserving the activation record of the alternative for later resumption.

Unlike a true coroutine implementation, however, the order in which patternForth alternatives can be resumed is strictly limited: only the "topmost" context on the data stack can be activated. This is because virtually all Forth code must "push onto" the data stack in the course of operation, and a stack cannot be expanded in its middle.

So, in a sense, patternForth implements "Last- In-First-Out" coroutines. As we have seen, this restriction does not constrain the operation of a pattern matcher. Indeed, when an equivalent pattern expression is evaluated in Icon, according to the goal-directed rule given above, it can be seen that the coroutines are activated in this same LIFO order.

### c) additional Icon capabilities

The Icon use of generators offers considerably more flexibility than the very pattern-oriented approach described here.

First, coroutine activation in Icon can be explicitly programmed, and need not be limited to goal-directed evaluation as it is in patternForth.

Icon coroutines can be activated in any order. This allows their use in a variety of problems which are suited to coroutine solutions.

The Icon generator can return any result, unlike the patternForth element routine which returns a simple pass/fail result. PatternForth element routines could be written to produce results, but the heavy dependence of the pattern logic on the data stack would necessitate some other means of returning the result -- and such a means would probably not be re-entrant or recursive.

Icon provides explicit support for some forms of "data backtracking," in which the effect of a data assignment can be reversed when control backtracking takes place. This, to a certain extent, is what the "fall-in" and "fall-out" backtracking records accomplish. There is no reason why patternForth could not support data backtracking as part of its backtracking logic; but it will be necessary to devise a suitable programming syntax for this operation.

## 14. Future work

### a) simplified successor syntax

The need "bracket" each element in a succession with the operators (<<) and (>>) is distasteful. This is much less readable than the SNOBOL4 syntax, where succession is indicated by simple juxtaposition of the elements.

Work done in parsing applications [ROD89a] indicates that the operators (<<) and (>>) can be subsumed into the entry and exit "housekeeping" of a pattern element.

**b) alternate stack usage**

It has been pointed out [WAI73] that, although it is much simpler to implement a History stack which is separate from the processor's return stack, it is not strictly necessary.

The potential advantages of freeing the Forth data stack for exclusive use by the element routines may well justify an attempt to rewrite the backtracking logic so as to use the Forth Return stack for History information. This would be of particular interest in the development of generators (below).

**c) implementation of generators**

The advantages cited previously for generators -- particularly the ability to return any result -- make the generator concept more powerful, and more widely applicable, than the logic of pattern matching.

It may be possible to modify the patternForth operation in such a way as to allow the return of "result sequences" by pattern routines. This may or may not necessitate a revision of stack usage (above), or an implementation of the full Icon coroutine system (below).

**d) implementation of full coroutines**

The recent advent of Modula-2 has renewed interest in the programming potential of coroutines, particularly in the field of concurrent programming [WIR88]. All of the task control and communication operations of a multitasking operating system have been portably implemented using of coroutines; thus, adding coroutines would create a Forth multiprocessing operating system.

Little work has been done with coroutines under the Forth environment. Forth's increasing use in concurrent processing systems would indicate that -- completely aside from any pattern matching implications -- a "true" coroutine system in Forth would find wide application.

**e) applications of LIFO coroutines**

Finally, an area of more speculative research concerns the existing coroutine capability of patternForth. It was noted previously that restricting the coroutine activation order to "last-in, first-out" was acceptable for pattern-matching problems. Since this form of coroutine is so easily implemented in any language using a stack allocation of "local" storage (not just Forth), it would be interesting and useful to ascertain what other kinds of programming problems can be successfully attacked with "LIFO coroutines." It would not be surprising to find that many "goal-directed" problems (e.g. expert systems) are amenable to this approach.