Fornax: A General-Purpose Programming Language

J. Storrs Hall
Laboratory for Computer Science Research
Rutgers University
`josh@cs.rutgers.edu`

Abstract

Fornax[1] is a very high-level programming based on pattern-matching concepts from Snobol (and Icon) and array or "data-parallel" concepts from *APL* (and J).

Introduction

During the 1980's it was the practice of the Student Chapter of the ACM at Rutgers to hold annual programming contests. This author had the honor of being one of the judges at such a contest. It worked as follows: The entrants were teams, of from 3 to 5 students. Each team was given an assignment of four programs to write. The team which got all four programs running correctly the earliest, won.

Normally most of the students used Pascal, which was the language taught in computer science courses, or Basic, which many had started in. A few engineering students did their programming in Fortran. This year there was entered a motley team of mavericks. Instead of working together, each member would do his work alone, and in a different language. One, who had interned at Bell Labs, would work in C. Another, God help him, was going to use assembly language. A third would try Lisp. And the other member of the "screwball" team, whose name was Damian Osisek, worked in SNOBOL.

The contest was unexciting that night, because Mr. Osisek, working alone, completed all four programs before anyone else, individual or team, completed even one.[2]

Philosophy

The general reaction to this phenomenon, and similar feats occasionally seen in *APL*, is to dismiss them as not applicable to larger, multi-person projects. We believe the truth is not so simplistic. If there is a tool, such as SNOBOL, which can rapidly solve problems of size x, it is the job of the software project manager

[1] Fornax, "the Furnace," is an obscure Southern Hemisphere constellation between Cetus and Eridanus.

[2] The following year, the use of SNOBOL (and *APL*) was banned.

to reduce the project to problems of size x and combine the results. In fact, the manager must do this anyway, and tends naturally to prefer software systems that alleviate the integration task; he is less concerned about the actual programming of the subparts because other people do that.

The other major objection to SNOBOL (and *APL*) is that they are interpreted, and thus unsuited for large systems that are heavily used or where performance is critical. This objection carries forward to the more powerful of the modern languages such as Prolog and the functional languages. It is perhaps even more cogent in the latter case, since Prolog is notoriously difficult to control; runtimes are not only slow, but unpredictable.

The design of Fornax is based on the notion that the state of the art in compilers has advanced sufficiently since the mid-1960's when SNOBOL and *APL* were invented, that reasonably efficient implementations of their primitives can now be compiled. The design goals of Fornax are, in order of importance:

- A set of datatypes and data-manipulation primitives that are powerful, well-matched to each other, but which can be implemented efficiently.
- A concise, uncluttered syntax. The austere beauty of Miranda is not to be achieved, but we can avoid the garbage-dump appearance of Common Lisp.

The primitive operations, of course, are part of an abstract world that includes the ontology of data-structures and the framework in which the operations are done. Logic and functional languages attempt to abstract away from the notion of a small sequence of changes to an overall state; lower-level conventional languages overindulge in it as being an efficiently implementable reflection of the actual process in hardware. Fornax includes state changes to be used where necessary, and indeed encourages this as quite a powerful primitive concept, but also provides primitives for operations that are conceptually parallel or unitary to avoid gratuitous decomposition.

Elements

The overall structure of a Fornax program is a set of function definitions, as in Lisp, *APL*, or the various logic and functional languages. Like Prolog, there may be several entries with the same function name, which are selected between on the basis of pattern matching with the arguments. Unlike Prolog, there is no backtracking. There are vector operators like *APL* and string pattern matching like SNOBOL.

Within a given definition the syntax is a simple operator expression. Undeclared variables are lexically scoped and local to the function definition. Side-effects are allowed but restricted to the action of explicit assignment operators; parameter passing is by value.

There are two kinds of expressions, pattern and value. A pattern can be thought of as a program that takes a string as input and either accepts (matches) it or not. Pattern matching has many of the features of programs, e.g. variable assignment can be done as part of a match. The left-hand sides of assignment statements, and the formal parameters of functions, can be patterns, with interesting and sometimes useful results.

Arrays

The datastructure in Fornax is a recursive, multidimensional array. It is similar in semantics to the "array theory" array of Trenchard More. Arrays may be "closed" to create synthetic scalar objects; objects of arbitrary complexity may be created in this fashion and treated as atomic by other code. This capability is intended for for such "object oriented" uses; multidimensionality is sufficient for more pedestrian aggregations.

While no concrete datastructure implements Fornax arrays efficiently for all operations, linked lists, indexed memory arrays, trees, hash tables, and so forth are appropriate for various subsets. It is intended that the programmer ignore the details and use the abstraction as needed, and the compiler will determine how best to implement the resulting operations (even changing data representations if necessary).

An array in Fornax code is written as a sequence of values, e.g. `6 7 8`. A "sequence" of one value, is an array of one value, is a scalar, e.g. `4`. There is no requirement that each subarray of an array have the same number of elements, or indeed have the same dimensionality. The "nested" and "multidimensional" interpretations of array structures in Fornax are unified; any array can be treated either way at any time.

Scalar Extension and Parallel Application

In *APL*, `2 + 3 4` automatically extends the `2` to the length of the vector `3 4` and automatically does + in parallel between the resulting vectors, producing `5 6`. Fornax does not do this; most of the functions operate on more complex structures than numbers, and it would be quite ad hoc to "destructure" them to some arbitrary level to do the parallelization automatically.

Instead parallelization is specified explicitly, using the "`.`" operator. `2 +. 3 4` produces the vectorization we want. In general, `a f. b` produces a result that is the same length as `b`, of which each element is `a f [the corresponding element of b]`. Likewise `a .f b` produces a result corresponding to `a`.

Extension is often used in conjunction with literal functions; any expression in `{}` is taken as a function; `&` and `&&` represent its right, and if supplied, left arguments. To take the absolute value of each number in `list` (without using `abs`), do `{&>0|-&}. list`.

Success and Failure

As in SNOBOL and Prolog, expressions in Fornax can succeed, returning a value, or fail. Normally any expression having a subexpression which fails, will itself fail; some constructs, used for control flow, "catch" failure. Unlike Prolog, backtracking is not done.

Arrays can contain failures as elements. Any "dotted" function application catches single-element failures.

L-values and R-values

In conventional programming languages, only variables, subscript expressions, and record field expressions have L-values, that is, only those expressions that refer to an explicit storage cell. SNOBOL allows the result of a pattern match to be an L-value, doing replacement of the substring that matched. As special cases, insertion and deletion could be done, replacing a null string with a non-null one and vice versa. Fornax adopts this formulation of L-values, allowing a fairly general range of expressions.

Patterns can be used as L-values, generally as the formal parameters to functions. Variable names in this context are interpreted as equivalent to `__`, the pattern that matches any object. Thus we can define list reverse as:

```
rev (a,,b) := (,b), rev a
rev () := ()
```

(`b` is rotated so it must match a single element; `a` can match any length of list, including 0, preceding the last element.)

Patterns

Pattern matching happens in two modes. The first is exact matching, denoted by `^`, which is essentially an equality predicate. The other mode is a substring match, denoted by `?`, where the pattern must match some substring of the subject array. That is to say, `s ? p` if there are arrays `a`, `b`, and `c` such that `s = a,b,c` and `b^p`.

The simplest patterns are values. A value, as a pattern, matches itself and nothing else: `5 6 ^ 5 6` and `4 5 6 7 ? 5 6` succeed, but `6 5^ 5 6` and `3 4 5 ? 5 6` fail. The L-value, as well as the R-value, of a successful `^` is its left argument. A `?` returns an array of the substrings matched; its R-value is the sequence of places in the string where the matches occurred. If

```
a = "peter piper picked a peck of pickled peppers"
```

then `a ? "pe"` returns `"pe" "pe" "pe" "pe" "pe"`, and `a ? "pe" .= ""` leaves `a` equal to `"ter pir picked a ck of pickled prs"`.

Fornax has a set of pattern primitives allowing the construction of patterns roughly equivalent to the ones in SNOBOL. Pattern functions and value functions are overloaded onto the same symbol set, sometimes subtly: `v1, v2` is the concatenation of `v1` and `v2`, whereas `p1, p2` is a pattern that matches any concatenation of something matching `p1` and something matching `p2`. On the other hand, `p1 | p2` is the pattern alternation of `p1` and `p2` (just as in SNOBOL) but `v1 | v2` is a flow-of-control expression like a Lisp OR.

All value functions and their pattern cognates have the same precedence; an expression parses the same whether it is in value or pattern context.

Examples

Suppose we want to count the characters in a file, say standard input. Although normally the line-at-a-time association is used for standard input, direct association is usually used for other files, and can be used for standard i/o if desired. The associated variable is stdio. Directly associated variables have the file's contents as a value; changing the variable changes the file. When interactivity is not necessary, this is usually a more straightforward approach to most algorithms:

```
flen = # stdio
```

Now suppose we want to count words instead of characters. The easiest way to do that is pattern matching. If we had a pattern that matched a word, we'd be well on our way to counting them. Fornax has built-in patterns alf and dig that match alphabetic characters or digits. Unary `?` creates a pattern:

```
word = ? (+ digit) | (+ alf), ["'", + alf]
```

Now for the pattern matching function, binary ?. It takes a string (actually, any array) on the left and a pattern on the right, and returns a list of all the places the pattern matched in the list; but only non-overlapping ones. When possible matches overlap, the one that starts earliest wins, and if two start at the same place, the longer one wins. That's just what we need; the length of the list of matches is the number of words:

```
nwds = # stdio ? word
```

Since Fornax actually has the pattern word built in, as well as one called line, the Unix utility `wc` can be written:

```
tty = (# stdio ? line) (# stdio ? word) (# stdio)
```

Alternatively, one could write:

```
tty =  #. stdio ?. line word _
```

where `_` is the primitive pattern that matches atomic objects, characters in this case.

This example shows how the results of a pattern match are used as data in further operations. More complex results can be produced by explicitly specifying the "value of the match" in the pattern. `p^x` is a pattern that matches whatever pattern `p` would match, but instead of returning the string that matched as its value, the match returns the value of `x`. (Any occurence of `&` in `x` has the value of the string that `p` matched.)

The following pattern definitions, assuming `var` and `num` are defined, match an arithmetic expression and produce as a result a Lisp-like nested prefix parse tree for it. A similar pattern could evaluate the expression in the process of parsing it:

```
factor := ? var | num | "(", (?exp), ")"
term := ? factor | x:factor, "*", y:term ^ ‘times’ x y
            | x:factor, "/", y:term ^ ‘quo’ x y
exp := ? term | x:term, "+", y:exp ^ ‘plus’ x y
            | x:term, "-", y:exp ^ ‘sub’ x y
```

then `"4+x*3-y" ^ exp` would produce

```
‘plus 4 ‘sub ‘times x 3’ y’’
```

assuming `var` and `num` produced symbols, and noting that `‘...’` is a syntax for quoted nestable symbol lists.

Note also that the monadic question mark in the definition of `factor` prevents the parentheses being returned as part of its value. `"(", (?exp), ")"` is equivalent to `"(", x:exp, ")" ^ x`, see below.

Conclusion

Fornax was originally designed as a general-purpose implementation language for an experimental associative architecture with hardware parallel and pattern-matching primitives (the Rutgers CAM). However, we feel it has significant contributions to make on conventional architectures, given the state of the art in compiler technology.

Fornax's current implementation status is “under construction.” Our strategy is a dumb interpreter and a smart compiler, in Fornax, to be bootstrapped. Fornax, as would be expected from its antecedents in SNOBOL, is a marvelous language for writing compilers. We find that Fornax programs are typically 25 times shorter than their equivalents in raw C.

A Abridged Dictionary of Fornax

The following is an incomplete list of Fornax functions, operators, and constants. Note that many of the function symbols have both value and pattern cognates; the precedences of cognates is always the same so that parsing is consistent. Precedences are arranged so that a few stereotypical statements, e.g.

```
(vec ? ( (((v:p),(v:p))^exp)
       | (((v:p),(v:p))^exp) ))
= ((x^y) ! exp)
```

can be written without parentheses.

Value Expression Functions

`()` (*Nil*) A list with no elements. It is identical to `‘’` (no symbols) and `""` (no characters).

`: n` (*Index Vector*) `n` must be a non-negative integer; the result is a vector of length `n` consisting of the integers `0` through `n-1`. This is one form of an *idempotent* array, i.e. one in which translates each valid index value into itself.

Often the argument to `:` is `_`, the primitive atomic pattern. Generally this is when the result is involved in a paired operation (e.g. `.+.`) with another vector. Primitive patterns may be used this way when some lexically apparent value (i.e. the length of the other array) will alone allow the expression to succeed.

`x = v` (*Assignment*) The L-value `x` is assigned the R-value `v`. This function has lower than standard precedence and returns the previous value of `x`; it is intended as the root operation of statement-level expressions.

`u f= v` (*Modification*) Has the effect of doing `u = u f v`, except that `u` is evaluated only once. `:=` is interpreted idiosyncratically as function or constant definition.

`x : v` (*Assignment*) The L-value `x` is assigned the R-value `v`. This function has higher than standard precedence and returns the value assigned; it is intended for use within other expressions.

`f / v` (*Reduction*) The elements of `v` are `f`'ed together from left to right; e.g. `+/1 2 3` is 6. Failure elements act as failures in expressions in a reduction. Thus `&/a` returns the final element of `a` if there are no fail elements, and `|/a` returns the first non-fail element.

The two-argument form `u f/ v` is equivalent to `f/ (,u),v`.

`n / v` (*Drop*) `v` with the first `n` elements removed. As in *APL*, negative values of `n` remove elements from the end.

`f\ v` (*Scan*) A parallel-prefix operator. Unlike *APL*, the definition of scan is based on a left-to-right accumulation, (rather than a sequence of partial reductions).

`u \ v` (*Take*) `u` is a non-negative integer. The result is a vector of the first `u` elements of `v`.

Take and drop can be used to specify substrings, either as L-values or R-values. If `a = "themselves"`, then `2\3/a` is `"ms"`; and after the evaluation of `2\3/a = " "`, `a` is `"the elves"`.

`~ v` (*Not*) Fails if `v` succeeds. Returns `()` if `v` fails.

`u ~ v` (*Not Equal*) Compares the values `u` and `v`, or if one or both are patterns, pattern matches between them. If the match succeeds, it fails. Otherwise returns `u`.

`? p` (*Pattern*) If found in a value context, evaluates `p` as a pattern.

`v ? p` (*Pattern Match*) Pattern matching: substrings (in the first dimension) of `v` that match `p` are sought. The value (L or R) is a vector of the matching substrings, left to right, maximal, and non-overlapping.

```
"aaabbc" ? x:_, *x
```

returns `"aaa" "bb" "c"`, even though the pattern actually matches 10 substrings of the original.

`&` (*Argument*) In `{}`, `&` is niladic (takes no arguments) represents the right argument to the function. E.g. `{&-2}. 3 8 4` is `1 6 2`.
In the value part, `v`, of a `p^v` expression in a pattern, `&` represents the portion of the string matched by the `p` part.

`u f|g v` (*Functional Or*) Equivalent to `u f v | u g v`, but only evaluates `u` and `v` once.

`u | v` (*Or*) If `u` succeeds, its value is returned and `v` is not executed. Otherwise, the value of `v`. (The so-called Cambridge OR, from its use as a control structure in LISP. The corresponding Cambridge AND is implicit in the actions of failure and sequencing. An if-then-else-like construct could be written `(if; then) | else`.

`^ v` (*Return*) Returns `v` as the value of the current function, defined or `{}`, or, if in a `[]`-protected expression, returns `v` as the value of the expression. An exact if-then-else construct could be written `[(if; ^then) | ^else]`.

`u ^ v` (*Equals*) Compares the values `u` and `v`, or if one or both are patterns, pattern matches between them. If the match succeeds, returns the match value, or fails if it fails. The L-value of a successful match is the left argument.

`@ v` (*Downrotate*) The opposite of uprotate (`,`). Has no effect on scalars. `@()` fails. To `process` each element of a list sequentially, do

```
@list ! process @(1\list = ())
```

(in practice, one would probably do `process. list`).

`u @ v` (*Indexing*) `u` is an array, and `v` is an index, e.g. `9 8 7@0` is `9`, `(2 3) (4 5)@1 0` is `4`, and so forth. Out-of-range indexes fail. Items selected are automatically downrotated.

`$ v` (*Indices Of*) The indices of `v` (along the first dimension), as a vector. Equivalent to `:#v` unless `v` has missing or non-numeric indices.

`u $ v` (*Associate*) A vector which maps the elements of `u` as indices, to corresponding elements of `v`. Can create symbolic indices.

`# v` (*Length*) The number of elements along the first dimension of `v`.

`n # v` (*Dup*) `n` copies of `v` concatenated together.

`* v` (*Iterate into List*) `v` is evaluated iteratively until it fails. The result is a list of the values produced. If the first attempt fails, the result is an empty list.

`+ v` (*Iterate into non-null List*) `v` is evaluated iteratively until it fails. The result is a list of the values produced. If the first attempt fails, the expression fails.

`u + v` (*Addition*) Similarly, `-` is subtraction, `*` is multiplication, is division.

`- v` (*Negation*) Of numbers.

`< v` (*Close*) Returns an atomic scalar object that can be opened to produce `v`. Closing lists of characters produces symbols.

`u < v` (*Comparison*) Less than; `>` is greater than. Both comparison operators, as well as `^`, return the left argument if they succeed, making constructions such as `a<b<c` work properly. `<|^/list` tells you if a list is sorted. All Fornax objects fall into a global collating order; `1 < 1 1 < 1 2 < 2 < 2 1 < 2 2`.

`> v` (*Open*) Returns whatever was closed to produce `v`.

`, v` (*Uprotate*) In the sense of changing the direction of the dimensional axes of `v`. The primitive function `rot` does a cyclic shift on the elements along the first dimension. Note that rotating a scalar (in either sense) does nothing.

`u , v` (*Concatenate*) Along the first, i.e. outermost, i.e. top-level dimension.

`! v` (*While*) `v` is iterated until it fails. Returns an integer, the number of times the iteration succeeded.

`u ! v` (*While*) `v` is iterated until `u` fails. Returns an integer, the number of times the iteration succeeded.

`; v` (*Identity*) Returns `v`.

`u ; v` (*Sequencing*) Returns `v`. Side effects of `u` and `v` are done in that order. Has very low precedence; intended to separate statement-level expressions.

Pattern Functions

`_` and `__` (*Primitive patterns*) `_` matches any scalar, and `__` matches anything.

`x : q` (*Assign at Match*) In a pattern, the left argument of `:` a variable name which is assigned the substring (or substructure) which matches the pattern that is the right argument. The value is valid lexically throughout the pattern, even before the assignment. (The idea is to put the assignment in the most easily matched position, for efficiency, or equivalently, allow the compiler latitude to do the assignment from any reference. E.g.

```
(1 2 3 4) 2 ? (a:__, b, c:__) (b:_)
```

assigns `a = 1`, `b = 2`, and `c = 3 4`.)

`~ p` (*Pattern Negation*) As a pattern, matches any string `s` for which `s ? p` would fail, i.e. anything not containing a match for `p` as a substring.

```
ccomment := ? "/*", (~ "*/"), "*/"
```

`p ~ q` (*Pattern Difference*) Matches anything that matches `p` and not `q`. Not the same as `p & ~ q`.
Note that pattern negation and difference are somewhat more intuitive methods of obtaining the same pattern semantics provided by SNOBOL's "fence" and Prolog's "cut".

`? p` (*Selected Subpattern*) The value as a pattern is the same as the argument; but when the whole pattern matches, the value of the (overall) match is the part that matched the argument of the monadic `?`. Instead of writing

|p (break; q.v.) one could write `(?*_),p`. Note, however, that the scope of `?` is the lexical pattern it appears in.

`p ? q` (*Unification*) Matches anything that matches both `p` and `q`, i.e. the resulting pattern is the most general unifier of the two patterns.

`| p` (*Optional*) Matches `p` or a null string. May also be written `[p]`.

`p | q` (*Alternation*) Pattern disjunction, i.e. matches anything that matches `p` or `q`.

`^ n` (*Pos*) In a pattern, matches a null string at position `n` (0 is left end).

`p ^ v` (*Pattern Value*) In a pattern, matches whatever `p` matches but the "value of the match" is `v`. "&" in `v` means whatever substring `p` actually matched.

`* p` (*Arbno*) Matches any number of `p`'s, including none.

`+ p` (*Nonempty Arbno*) Matches at least one `p` and as many more as available.

`n \ p` (*Take*) `n` is a non-negative integer. Matches any list of length `n` which is an initial substring of a list which would match `p`. Drop (\) works similarly. There are subtle differences in the value and pattern forms; `1\()` as a value, fails, but as a pattern, will match a gap in the subject list.

`! p` (*Break*) A generalization of the function in SNOBOL. Matches everything up to, but not including, any substring that would match `p`.

`p ! q` (*Any Order*) Matches either `p,q` or `q,p`. However, if either argument is itself a `!` expression, any permutation of the constituent arguments is allowed.