APROL
A Hybrid Language

Dennis Holmes
IBM Corporation
5600 Cottle Road, D25/513
San Jose, California 95193 USA
(408) 256-4516
dholmes@netcom.com

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200 USA
(210) 736-7480
FAX: (210) 736-7477
jhowland@ariel.cs.trinity.edu

Abstract

This paper describes the design of a hybrid language which combines the features of an array processing language and lisp dialect in a consistent and useful manner. This language, APROL (Array PROcessing Lisp) is derived from the J dialect of APL and the Scheme dialect of Lisp. The base syntactic structure is taken from Scheme, while the array processing features are based on the J programming language. A prototype implementation has been made and some experiences with this implementation are described. This implementation uses J as an imbedded array processing engine in a Scheme intrepreter/compiler.

The language as specified provides a set of data types and manipulation tools which is more diverse than found in either Scheme or J. APROL allows the programmer to apply array processing functions to lists of arrays in the Scheme style and list processing functions to arrays of lists in typical J style. The result is a language which not only brings array processing capabilities to Scheme, but also significantly extends the functionality of the Scheme language.

Keywords: APL, J, LISP, Functional Programming, Lists, Arrays

1 Introduction

1.1 Functional Languages

A functional programming language has as its basis the mathematical concept of a function. One basic characteristic of mathematical functions is abstraction, the practice of separating the details of a process from the conceptual representation of the process. We abstract a function by giving it a name such as $f(x)$; in some cases, we may further abstract the function by assigning a symbol, as with addition (+) or integration ($\int$). While a symbol is in one respect just another name, the association of a symbolic character rather than a word with a function can serve to remove any detrimental connotations or preconceptions which an English name may cause one to associate with the function. The same technique of abstraction applies in a programming language, where the details of a process are replaced at higher levels within a program by a function or procedure call denoted by something as simple as `calculate_result`. In the same way, data types in a programming language are given names to hide their internal representations.

Modern functional languages carry this concept of data abstraction further, allowing multiple objects of unspecified types to coexist as elements of a single data structure similar to an n-tuple in mathematics. Thus, in addition to the abstraction technique of naming procedures described above, functional languages exhibit a high degree of data abstraction, where the specific characteristics of data objects are replaced at high levels by simplified conceptual descriptions of the objects.

The basic algorithmic constructs of sequence, condition, and repetition are generally implied in or understood from mathematical notation of functions. Consider integration as an example. By definition, an integration function is the limit of the sum of a series as the number of terms approaches infinity. The sequence implied is to sum the series and then evaluate the limit, while taking the sum of the series is an iterative process. Yet we manage to convey all this information as

$$\lim_{n \to \infty} \sum_{i=0}^{n} f(\xi_i)\Delta_i x,$$

or more simply,

$$\int_a^b f(x)dx.$$

Conversely, most imperative programming languages (such as C or BASIC) have structures or commands of an explicit nature for implementing sequence, condition, and iteration. The notation of functional languages, however, is more closely tied to mathematical notation, and these constructs are similarly implied by the notation.

Functional languages are also known as applicative languages due to the nature of their operation. In mathematics, we say that a function is applied to its arguments. In a composite function such as $f(g(x))$, one function is applied to the result of another; in this case, the function f is applied to the result obtained by applying g to the argument x. Functional languages operate in exactly the same way; an entire program is built up from the composition of functions. This style helps to eliminate the "side effects" that occur with imperative languages in which preliminary results are often assigned to an auxiliary variable or procedure parameters are manipulated in place. Functional language notation is usually much more compact than imperative notation as well, a direct result of the similarity to mathematical notation.

1.2 Scheme

The Scheme programming language is a dialect of Lisp (from "LISt Processor") based on Alonzo Church's lambda calculus. The base data structure in Scheme is the list. The notation of a list is that of an ordered n-tuple, but in fact the object is treated as an ordered pair. Each member of the pair may be a fundamental type, such as a number or string, or another list. Strictly speaking, the second member of a list is also a list, but note that pairs which are not lists are perfectly permissible and may not have this property. If a list appears to have only one member, the second member is the empty list; considered independently, the empty list is denoted by `()`. The two parts of a list are accessed by the functions `car` and `cdr`, which return the first and second parts of the list argument, respectively. The `car` of a list is always the first element appearing in the list; the `cdr` is a list containing the remainder of the members of the list argument. Because of the nature of lists, a technique frequently used in Scheme for processing the elements of a list is to create a function to process the `car` of the list and then recurse on the `cdr` of the list.

A powerful and attractive property of the Scheme language is that program code looks, behaves, and can be used as data; thus, functions are handled as first class objects. This property allows one to write functions which might take functions as arguments, return functions as results, and even construct new functions and add them to the environment. To differentiate a literal list from a procedure application in program code, Scheme employs a method known as *quoting* in which a single quotation mark is placed immediately before a literal list or the Scheme function `quote` is applied to the list. These characteristics provide an ideal set of tools for implementing a system which exhibits a high degree of data and procedural abstraction.

Another data structure available in Scheme is the vector, essentially an array data structure of one dimension. The notation of a Scheme vector is identical to a list preceded by the number or pound sign (#). (Note that the current Scheme specification indicates that vector expressions must be quoted within program code [CLIN91,□26].) However, Scheme does not provide any extended array processing facilities, and the recursive techniques used when approaching problems in Scheme do not always lend themselves easily to performing the kinds of iterative operations typically used in array processing. While the list is a powerful structure, Scheme is simply impractical for many mathematical applications involving arrays.

1.3 J

The J programming language is Dr. Kenneth Iverson's recent derivative of APL, a language which he developed and which was first implemented on a computer in 1966. APL is based on standard mathematical notation; the syntax utilizes infix notation and even the same symbols used in mathematics. J retains this concept but uses the ASCII character set for greater portability and user ease. In addition, J offers significant improvements and capabilities over the original APL.

As in APL, the base data structure in J is the array, representing the mathematical concept of a vector or matrix. Two important concepts in J are the shape and rank of an array. The shape of an array is a vector containing the number of elements in each axis of the array. The rank of an array is the number of elements in the array's shape vector, or the number of axes the array has.

The J environment offers an extremely powerful set of array processing tools. Matrix arithmetic operations are fundamental; several common matrix functions, such as inverse and transpose, are provided and even extended to apply over arrays of rank greater than two.

One disadvantage of J is that it can be a very difficult language to learn. Although it uses the ASCII character set, the notation is very symbolic and requires much time to master. The programming techniques used and procedural abstraction facilities employed in J are powerful and innovative, but somewhat different from other languages and can seem rather formidable to the beginner.

1.4 Array PROcessing Lisp

It seems that a new language is desirable which would combine the characteristics of both of these languages, utilizing Scheme's simplicity and intuitive procedural abstraction techniques and J's powerful array processing capabilities. Such a language would allow one to make full use of the list processing features and powerful recursive nature of Scheme while retaining the ability to easily and efficiently manipulate complex mathematical and tabular data. This language currently has a working name of Array Processing Lisp, or APROL.

The Scheme syntax offers one advantage with respect to this goal that J does not. With the vector data structure, Scheme has an existing representation for a basic array type. J, however, has no representation suitable for a list; one could implement lists or an equivalent using arrays, but significant syntactic extensions must be made to the language in order to avoid losing much of the intuitive benefits of the list data structure. In addition, the simplicity and clarity of Scheme seems to make it the better basis for the new language.

The importance of this decision concerning the basic syntactic structure of the language should not be underemphasized. One might at first be inclined to suggest that a hybrid syntax be developed which could use Scheme's prefix notation for list processing and J's infix notation for array processing. While this idea has some appealing aspects, it does not lend itself well to a full integration of Scheme and J characteristics. We wish to do much more than simply give Scheme the capability to manipulate arrays; we wish to extend the operations of Scheme to apply to arrays and the operations of J to apply to lists in a consistent manner. A hybrid syntax would simply confuse the user, complicate the language implementation, and increase the potential for ambiguities in the language.

The APROL language has several potential applications. Existing Scheme and J programs could be ported quickly and easily to this environment; ideally, existing Scheme programs would run on this system with no modification. There may conceivably exist applications which seem to be suited to functional language solutions but require slightly more capability for a practical implementation than either Scheme or J offers. In addition, other applications may be discovered or invented which favor implementation in APROL.

Furthermore, the APROL environment would make a powerful educational tool. Since Scheme programming is relatively easy to learn, APROL can provide novice programmers a powerful array processing package without the complexity of the J syntax. Additionally, APROL can be used to introduce J to students who are familiar with Scheme. Working in APROL would teach them the concepts, functions, and techniques of J programming quickly; all that remains then is to associate the J symbols with the function names and change from prefix to infix notation. Such a technique could greatly reduce the J learning curve.

2 Project Objectives

The purpose of the APROL language is to bring together the list data structure and processing functions from Scheme and the array structure and operations from J in a single, unified programming environment. The Scheme syntax will provide the basic structure of APROL, and adherence to the Scheme specification [CLIN91] will help to maintain consistency in the language, provide a basis for making decisions concerning the language design, and aid in gaining acceptance for the language in the programming community. As such, Array Processing Lisp can be classified as an *extension* to the Scheme language, and also as a dialect of Scheme or Lisp.

3 Underlying Basis of APROL

It is important to examine the underlying model of any new programming language. The Scheme language is firmly based on the lambda calculus developed by the mathematician Alonzo Church and published in 1941 [CHUR59]. The notation used in Scheme to build the lambda constructs (functions) and function applications which make up the language is taken directly from Church's work.

J is built on an extended model of matrix arithmetic. This language has a strong relationship to the lambda calculus as well, however. Although the notation is somewhat different from that used by Church, careful examination shows that the same techniques of function abstraction are used. In Scheme (and the lambda calculus), one has the ability to define a new function by applying a function to primitive object parameters which define an instance of the function. One may also apply a function to a set of other functions, producing a new function which performs some composition of the parameter functions. Such constructor functions may be said to define a class of operations; the functions resulting from their application are the instances of the class.

J accomplishes the same functionality with symbols known as conjunctions. A conjunction is a function which operates on two verbs (functions) or on a verb and a primitive or array object. For example, consider the conjunction `&` ("with" or "compose"). If we execute the phrase `^&2`, we obtain the square function [IVER91B, 5]. Likewise, `(+/).*` provides the inner product of addition over multiplication, or matrix multiplication, by composing the operations of multiplication and inserted addition using the conjunction `.` ("dot product") to form an inner product function.

One can easily see how the Scheme syntax provides the tools to construct equivalent mechanisms to the J conjunctions. The J function `^` ("power") is a function of two arguments, requiring a base and an exponent. With the construct `^&2`, we apply this function to a single argument, the exponent, and obtain a function of one argument, the base, in return. Similarly, the **.** ("dot product") operator is a function over a domain of functions and defines a new operation based on its arguments. Both of these cases correspond precisely to the previously described facilities of the Scheme language; one can simply create a function which operates on a domain of other functions.

It therefore seems safe to say that the underlying models of Scheme and J are not incompatible. What remains to be examined is the issue of smoothly integrating the array data structure and operations into the Scheme environment. Through abstraction, Scheme has the capability to manipulate several primitive data types within the list and vector structures. Some basic array handling can then be obtained simply by defining an array as yet another primitive type.

However, considering the vector, Scheme already possesses a structure which is functionally equivalent to the array of rank one. In J, all higher order arrays are constructed from rank one arrays by the application of a *shape* function, an instance of a class of functions defined as the application of a shape vector. A shape function applies its associated shape vector to an array argument to obtain an array consisting of the same elements but rearranged to fit the applied shape vector. Arrays of any rank and shape can therefore be represented in Scheme simply by associating a shape vector and an element vector together in a data structure such as a list. Once the array data structure has been represented, one should then be able to implement functions to perform recursive or iterative manipulations on the array and its elements.

Although many of the algorithms for array operations one might implement in Scheme would not be very efficient, it is nonetheless possible to bring J arrays and operations to Scheme simply by implementing them in the Scheme language itself. With this knowledge of the basic compatibility of the concepts, it becomes desirable to provide within the system a more efficient set of array manipulation tools and extend the operations of the combined languages to allow efficient new methods of processing combined structures of lists within arrays and arrays within lists.

3.2 Data Abstraction

Data abstraction is an important feature of modern programming techniques; it allows maximum flexibility of program design and permits modifications and enhancements to be added to a program relatively easily when necessary. In J, data abstraction is provided by the functions `<` ("box") and `>` ("open"). When the `box` function is applied to an array object, the object is metaphorically sealed in a box, transforming the object into a rank zero (primitive) object. This box can be used as an element of an array of boxed items, even if the other boxed objects are of varying types, as illustrated below:

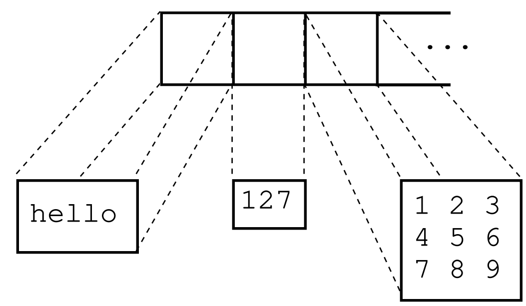

Figure 3.2.1: An array of boxes

To access the contents of a box, the open function is applied to the box. Thus the expression `><A` obtains the object denoted by `A`, and the function denoted by the sequence `><` represents an identity function for all objects in the J environment. If `A` represents an array of boxed numbers, then `2+>A` evaluates to an array in which each element is equal to two plus the value of the unboxed corresponding element of `A`.

The Scheme language has no functions corresponding to the box and open operations of J, as the abstraction and referencing operations are handled automatically by the Scheme environment. Each member of a list exists within the list as simply an abstracted "list member" object. Thus the user may freely insert objects of any primitive or structured type into the list with a `cons` or `append` function. The user never needs to apply any explicit abstraction function. If deeper levels of abstraction are needed, however, the item or items can easily be enclosed in another list within the list.

Boxing is necessary in J because of the parallel nature of the way the language operates. When a simple arithmetic operator is applied to an array in J, the result is the effect of applying the function across the entire array. `2+A` results in an array which is the result of adding two to every element of `A`; `A+B` performs the matrix addition of `A` and `B` (an element of the result is equal to the corresponding element of `A` added to the corresponding element of `B`).

If elements of an array were permitted to have rank greater than zero, ambiguities would run rampant in the language. For example, consider a rank two array `C` with elements of rank two (a matrix of matrices). Also consider, for this example, the function `{.` ("take"), which in the form `N{.A` evaluates to the first `N` items (elements, rows, matrices, etc.) of `A`; the rank of an item of `A` is one less than the rank of `A`. If we execute the sentence `1{.C` ("Take one from `C`."), two reasonable interpretations are possible. Should we obtain the first row of `C` (a vector of matrices) or a rank two array whose elements consist of the first row of each corresponding element of `C` (a matrix of vectors)? Note that the normal semantics of a function applied to an array indicate to apply the function to each element of the array; if this approach is taken, however, it becomes impossible to ever obtain the first row of `C` by any function!

In addition to requiring array elements to have zero rank, J requires all elements of an array to be of the same general type. If numeric data is to be placed in the same array alongside character data, then each element of the array must be boxed. This requirement insures uniform applicability of any function over the elements of the array; very few operations over the elements of an array would make sense if the elements were not of similar type.

Although the explicit nature of J's abstraction techniques can be beneficial in introducing the user to the specifics of abstraction, it seems out of place in APROL, where the Scheme-based environment already hides the abstraction from the user in most cases. For consistency in the language and clarity and conciseness of programs written in APROL, it would seem desirable to continue to hide the abstraction technique from the user; essentially, we abstract the abstraction. However, this is only possible in part, due to the potential ambiguity just described. Boxing will occur automatically within APROL, but we must retain the `open` function in order to be able to apply functions to the components of an array rather than to the array itself. In addition, it may prove beneficial to make explicit abstraction techniques available to the user for special applications without actually requiring or encouraging the use of these functions. Furthermore, it will almost certainly prove beneficial to display boxed array elements within a box (in J style) when output to the terminal to insure that the output is clear and unambiguous.

3.3 Arrays in APROL

We have existent in Scheme a vector data structure, which corresponds conceptually and functionally to an array of rank one. Thus it is appropriate to use this structure as a convenient means of representing such an array. If we follow the method used in J for constructing arrays of higher rank, we can obtain such an array by applying a shape function to a rank one array. For example, the APROL expression `(shape '#(2 2) '#(1 2 3 4))` results in a two by two matrix whose elements are the integers from one to four, in row major order.

This correlation between the vector and the rank one array helps to retain simplicity in the language; having two separate but functionally equivalent data types would not provide any particular benefit. For compatibility with Scheme, of course, it is desirable to have the Scheme vector functions be applicable to the array type, but defined only for arrays of rank one. The correlation also prevents the need to modify or add to the syntax of the language in order to accommodate array construction. Thus the language should be extremely easy for Scheme programmers to use, appearing at one level as a powerful Scheme array processing package.

The semantics of complex vector constructions is also retained. For example, the expression `'#(#(1 2 3) 4 5)` remains valid, indicating an array of rank one whose first element is the rank one array `#(1 2 3)` and whose other elements are the integers four and five. This example also illustrates the kind of abstraction that is desirable in APROL. In J, this object is created by the expression `(<1 2 3),(<4),<5` (or `1 2 3;4;5`) and corresponds to an array with a boxed array as the first element and boxed integers as the second and third. In APROL, although the elements are of unlike type, no explicit abstraction operation is applied to any of the elements at the user interface level; likewise, no special operation need be applied when referencing the elements of the array.

For representing arrays of rank greater than one, something more than the Scheme vector is needed; obviously the data structure must contain, at the very least, the shape of the array and a list of its elements. In addition, accessor functions are needed which can determine the rank, shape, and elements of the array. In Scheme, however, we can make use of procedural abstraction to bundle these accessor functions along with the data structure, resulting in an array object which knows its own identifying characteristics. Thus we simply "ask" the array to return its rank, its shape, or its elements, eliminating the need for external accessor functions.

In order to gain the behavior specified by these requirements, the array object must take the form of a Scheme function. This function takes as its single argument a symbol identifying the desired characteristic. If the name `a` is bound to an array object, then the expression `(a 'rank)` evaluates to the rank of `a` and `(a 'shape)` to the shape vector of `a`

Whatever the specification for the nature of array objects, it should be realized that for consistency this specification must apply to all arrays, including those of rank one. In order to accomplish this, the specification proposed here, in particular, requires a redefinition of the nature of the Scheme vector object. In its present form, a vector cannot be applied to any object as a procedure can. We have already established the equivalence relationship between the vector and the rank one array. The proposed specification is capable of representing any array object; vectors can therefore be represented as rank one arrays using this system. The underlying representation of the vector must be changed to form it into an array object as defined in the above specification. Additionally, the Scheme vector processing functions which access the underlying structure of the vector object must be rewritten to accommodate the new internal representation.

This change in the nature of the vector object will be mostly transparent to users writing and executing regular Scheme code; that is, this structure will retain all the functionality and input notation of the Scheme vector type. The new representation does, however, extend the capabilities of the vector type to include all those of the array data type. The vector has become an instance of the array class, and its *internal* representation is similar to that of other array objects. The only difference that the Scheme programmer will notice between the APROL vector and the Scheme vector is the external, or output, representation. The Scheme vector notation is inadequate for displaying arrays of rank greater than one, nor is it suitable for displaying vectors containing elements of rank greater than one. In order to maintain consistency, then, vectors in APROL will be printed using the same J-like technique that will be used to print arrays in general:

```
    1 2 3
    4 5 6        #(1 2 3 4 5)     1 2 3 4 5
    7 8 9
   Matrix         Vector           Vector
(J notation) (Scheme notation) (J notation)
```

Figure 3.3.1: Array output formats

It should be noted that although the array object, and therefore the vector, is now defined as a particular class of function, the vector remains defined as a disjoint type per section 3.4 of the *Revised⁴ Report on the Scheme Programming Language* (R4RS); that is, a vector object always satisfies the predicate `vector?` but not the predicate `procedure?` [CLIN91,7]. Likewise, a procedure (other than an array) always satisfies `procedure?` but not `array?` or `vector?`. An array will always satisfy `array?`, but satisfies `vector?` if and only if its rank is one. The array type is exclusive to all types to which the vector type is exclusive, but naturally not to the vector type itself; therefore, since `array?` represents the more general case, `array?` is substituted for `vector?` in the list of predicates appearing in R4RS[3.4] for determining disjointness of types in APROL. It should be noted at this point that since an APROL vector has been fully defined to be an array of rank one, the terms "vector" and "array of rank one" are interchangeable and will consequently be used as such in this discussion; additionally, the terms "array" and "array object," referring to the class of arrays, shall include vectors.

These differences in the representation of vectors in Scheme and APROL require no modification to the Scheme syntax for APROL, nor is Scheme functionality impeded in APROL as a result of these differences. With arrays being represented as procedures, one might be concerned about including arrays as members of list structures; ordinarily this is how a Scheme programmer indicates a function application. However, literal lists are already required to be quoted in the Scheme language to prevent evaluation. A list whose `car` is an array object can be evaluated; however, the user must insure that the `cdr` of the list contains an argument appropriate to the application of an array object. Failure to do so could result in an error message or unpredictable results, as the application of the procedure which is the array object would be undefined.

3.4 Strings

One may raise questions concerning the handling of the string data type, which may be viewed as a special type of vector containing an ordered sequence of character data. In Scheme, the string exists as a primitive type, one of the eight disjoint types defined by the predicates listed in R4RS[3.4]. In J, the concept of a string exists only as an array of character data. The representation used in J is very flexible, allowing the full range of J array-handling operations to manipulate the string. For consistency, it is best to select a single representation for all strings in APROL.

While the array representation is more powerful, we can easily obtain this representation from a Scheme string using the conversion procedures available in the Scheme environment; the expression `(list->vector (string->list s))` yields a vector containing the characters from the string `s`. If this is too unwieldy, the user may easily define a function `string->vector` which combines the two conversions (see Appendix B). Alternatively, the functions `string->vector` and `vector->string` could be specified as essential procedures and included in every APROL implementation. In order to maintain compliance with the Scheme standard and thus compatibility with portable Scheme code, it is reasonable to use the Scheme representation and convert strings to vectors when complex manipulation is necessary. This method also allows the Scheme string comparators and manipulation functions to be retained and provides the capability to convert computed vectors to strings.

3.5 Designations of Array Functions

In order to maintain familiarity and compatibility in some form with the existing J notation, it is desirable to use names for the J functions of APROL which resemble the names used in J. Unfortunately, some of the symbols used in J have meanings in Scheme with which interference is not acceptable; in addition, many of the symbols could be ambiguous, or at least confusing, when removed from the context of an infix notation. A good solution, then, is to use the English names for the functions as given in the J dictionary [IVER91B]. Additionally, some J functions already have corresponding functions in Scheme; the implementations of these functions will need to be altered to extend their functionality to include arrays to prevent having redundant operations in the language.

The subject of conjunctions and adverbs as used in J also needs to be addressed. The conjunction is a type of operator which takes two parameters and derives a new function based on those parameters. Thus the conjunction behaves as a function which operates on either a function and some object or two functions. Like any other function in J, conjunctions have symbols and associated English names. To use a conjunction, then, we apply the desired conjunction function to two appropriate entities and apply the result to the desired arguments. For example, the expression `((with power 2) 5)` evaluates to five squared or 25, and `((compose times negate) 2 5)` to -2 times -5, or ten. These expressions correspond to the J expressions `^&2 5` and `2*&-5`, respectively. Note the use of different names for the two forms of this conjunction; such a distinction may contribute to the clarity of intent of programs.

Adverbs in J operate in a similar manner; an adverb is a symbol which modifies the behavior of a single verb. For example, the J expression `+/1 2 3` uses the adverb `/` ("insert") to modify the behavior of the verb `+` ("plus"). The result is the evaluation of the expression formed by inserting the verb `+` between each of the elements of the argument, which gives `1+2+3`, resulting in the value six. Adverbs, then, can be viewed as the monadic version of the conjunction; whereas a conjunction produces a function based on two arguments, an adverb produces a function from a single function argument. The above example would be written in APROL as `((insert +) '#(1 2 3))`.

There exist in J the implied operations of fork and hook which are represented by an isolated sequence of verbs. The effects of these operations are like those of conjunctions; a new function is formed which is defined by some composition of the parameter functions. Interpretation occurs in typical J right to left style. A fork contains three verbs, with the central verb operating on the results of independently applying the parameter verbs to the arguments. A hook contains two verbs, with the leftmost verb operating on the first argument and the application of the parameter verb to the second argument. The following diagrams illustrate the monadic (unary) and dyadic (binary) cases of these constructions for functions `f`, `g`, and `h` and arguments `x` and `y` [IVER91B,□6]:

```
         Fork                    Hook
    g           g         g         g
  /   \       /   \     /   \     /   \
 f     h     f     h   y     h   x     h
 |     |    / \   / \        |         |
 y     y   x   y x   y       y         y

 (fgh)y     x(fgh)y    (gh)y     x(gh)y
```

Figure 3.5.1: Fork and Hook

Generally, a sequence of odd length forms a sequence of forks, and a sequence of even length produces a hook followed by a sequence of forks. So the sequence `f g h i j` (all functions) is equivalent to `f g (h i j)`, and `f g h i` is equivalent to `f (g h i)` [IVER91B,6]. Programming examples and equivalences are shown in Appendix A. In order to provide this same functionality in APROL, these implied operations must have a name. While "compose" would perhaps be an applicable name, the conjunction `&` already uses this as an alternate name; therefore we may find the name "train," the term used in the J dictionary for such a sequence of verbs, to be appropriate. Note that this function should accept any number of function arguments greater than or equal to two.

3.6 Extension of Scheme and J Functions

In the context of the APROL language, several of the functions inherited from J and Scheme can be extended to provide greater functionality and integration of the concepts behind those languages. The dyadic J functions can be extended to accept more than two arguments in the style of many Scheme procedures; for example, the Scheme expression `(+ a b c)` is equivalent to the mathematical expression $a+b+c$ and could be extended to apply to array addition in the same manner. The monadic J functions need no such modification, as the Scheme function `map` allows the parallel application of a procedure to all the elements of a list. For example, if `a` and `b` are arrays, `(map transpose '(a b))` is equivalent to `(list (transpose a) (transpose b))`. Thus we already have the capability to apply a monadic array operator across a list of arrays.

Likewise, functions inherited from Scheme can be extended to apply across arrays in typical J style. For example, if `c` is an array of lists, then `(car c)` evaluates to an array `r` in which each element is the `car` of the corresponding list, and the shape of `r` is identical to the shape of `c`. The `map` procedure provides this type of operation on a list of lists, but we wish to extend this capability to apply to arrays of lists where we can gain the benefits of greater organizational flexibility and array operability. This enhancement is not intended to fully replace the use of the list structure and `map` as a means of storing and manipulating lists, but rather to provide a tool for efficiently manipulating lists contained within array structures.

These extensions immediately raise issues concerning the associativity of procedures in APROL. The Scheme language implements left to right associativity; the expression `(- a b c)` is equivalent to the mathematical expression $(a-b)-c$, not $a-(b-c)$. J, however, uses right to left associativity. The J expression `a-b-c` translates to the mathematical expression $a-(b-c)$. Thus in APROL an interpretation must be selected for expressions of the form (<procedure> <arg> ...) as well as any newly defined forms of multiple-argument expressions (such as those created by `insert`). If we are to keep to the goal of maintaining compatibility with Scheme, then obviously (<procedure> <arg> ...) must associate from left to right; however, it is possible to define an adverb `rtol` which will force right to left associativity of a verb. Since the other forms in APROL which require associativity to be defined are the result of inheriting functions or styles from J, it is convenient to the J programmer to have these expressions associate from right to left. One advantage of this method is that, since right to left associativity is used in the definitions of the fork and hook, these constructs retain the semantics previously described; otherwise they would have to be redefined, which would be inconsistent with the J style elements incorporated into APROL.

In another extension to the notation, it would be possible to have APROL use the notation for negative numbers employed by J. In order to eliminate ambiguity, the negative sign in J is represented by a symbol different from the negation operator. However, the Scheme notation has no such ambiguity. Although they use the same symbol, the negation operator always exists as an independent function object, and the negative sign is always bound to a numeric constant and is found immediately preceding a numeral with no white space in between. Utilizing a different symbol for the negative sign would have the result that some Scheme code would require slight modification to run in an APROL environment. Since there is no ambiguity or unclarity in the existing syntax, it is best to continue to use the existing Scheme symbolism.

4 Implementation Issues

Some questions and issues concerning the implementation of an APROL interpreter have arisen during the design process. While decisions concerning the design of a programming language should not be made on the basis of any particular implementation technique, considering methods for and potential problems in implementation can provide and have provided valuable insight into design concerns.

4.1 Scheme Implementation

The APROL language as specified here cannot be fully implemented in a Scheme environment. While many functions could be written to provide capabilities equivalent to those discussed here, several modifications to the Scheme interpreter would be required in order to permit the notation desired. If Scheme functions (both those with equivalents in J and others we wish to extend) are to be extended, they must be implemented as completely separate functions, as many Scheme systems do not allow the redefinition of integrable procedures. Thus, instead of using the function + to perform all addition we must define a new function `aprol-plus`; similarly, `car` cannot be applied across arrays, so we must define `aprol-car`. However, it should be noted that `aprol-plus` and `aprol-car` could could be designed to include the functionality of + and `car`.

Changing the nature of vectors to conform to array objects also requires modifications to the interpreter, as vector notation entered by the user must be converted. Using an implementation in Scheme, the user would be required to apply a conversion function to all Scheme vectors. The Scheme vector processing functions also need to be modified, which requires redefining integrable procedures or separation of the APROL versions of these functions.

Another important consideration is the need to properly display the external representation of arrays upon output to the terminal. The notation used by J is clear, concise, and true to mathematical representation. However, printing array results requires that they be passed to a printing function or that the printing method be written into the output formatting routines used by the interpreter.

4.2 Using LinkJ as an Imbedded Language

The J source code contains an option activated at compile time which allows the system to be compiled, essentially, as a library; in this form, the J system is known as LinkJ. This feature can be used to facilitate a rapid implementation of APROL without having to re-implement all the functionality of J in Scheme. The source for LinkJ is simply compiled along with the source code for the Scheme system, producing a single executable file possessing the workings of both environments.

Careful selection of a Scheme system for such an implementation is important. Of course, the source code for the Scheme system must be available. In addition, one must be able to insert hooks into the system whereby calls to Scheme functions within the Scheme environment can reference C functions (J is currently implemented only in C). In addition, we must have some method of operating on Scheme objects and executing Scheme functions from J; ideally the Scheme system would also have hooks to allow the referencing of Scheme functions from C. As another consideration, a Scheme system which is very complete and adherent to at least the R4RS specification will provide APROL with the greatest functionality and reliability.

Another possibility for hooking into the Scheme system is to obtain string representations for Scheme objects and pass them into the Scheme reader; perhaps a function of the Scheme interpreter could be used to print the external representation of an object to a string in preparation for this operation. The problem which remains is obtaining results back from the Scheme environment. If a special Scheme interface function were used, however, it could reference a C function to store the result in a static memory area. The J system could then use this same C function to obtain the stored result.

The abstraction techniques of both languages can be used to an advantage in such an implementation. If no operation is to be performed on a foreign object, it can be abstracted and manipulated as a unit; we simply encode the block of data used to represent the object in its parent language and assign it an appropriate type. In J, this type is the box, which is capable of holding data of any type; in Scheme, we must make a new type, perhaps named "j-object." This type of encoding can be used to avoid much unnecessary conversion and translation between the data structures of the two languages.

There is another method which can eliminate this complex encoding and passing of object representations and the difficulty of interacting with the Scheme reader. If, within the context of the array object as specified, there exists a Scheme data structure containing the array information, then the array elements can be accessed and processed using procedures implemented in Scheme from within the Scheme environment. When J processing of the array is desired, a Scheme interface function can be used to construct the J representation by means of C function calls.

As this interface function will be the last step before J execution occurs, it will be responsible for invoking the C interface to J. When the external J execution terminates, control is returned to the interface function along with the encoded J object result. The interface function then works with C decoding functions (which utilize the J system) to convert the object into its Scheme representation. This Scheme object is then returned to the invoking Scheme procedure; it should be noted that this object is not necessarily an array; J functions can be constructed to produce a primitive type from an array, such as accessing a particular array element.

Each function inherited from the J language will require a Scheme interface function the name of which carries the English name of the J function. In order for the conjunction and adverb functions to provide applicability to Scheme and user-defined functions, however, these classes of functions are best fully implemented in Scheme; this technique requires perhaps a little more work in the implementation but prevents the need to transfer control back and forth between the Scheme and LinkJ systems.

4.3 Independent Implementation

Designing an implementation independent of existing environments can result in a much more efficient interpreter. Furthermore, this allows the implementor to optimize and unify the underlying data structures of the system to eliminate translation problems.

To save having to re-implement the operations of J to work within an independent system, one might consider developing a new Scheme system specifically designed to interface with the LinkJ environment. Such a system would need to be carefully designed, and the implementor would probably benefit from experimenting with LinkJ and existing Scheme systems first. Although this type of implementation would perhaps not work as smoothly as a completely independent product, it does allow much optimization of data structures and interface techniques between the systems.

5 Conclusions

5.1 Evaluation of Project

The nature of this project is rather experimental, and its goal has been to determine a means to provide the data handling capabilities and programming techniques used in two differing functional languages in a single unified programming environment. In this specification of Array Processing Lisp, the means is provided for an integration of the array type and operations used in J into the Scheme environment. List and array members are interchangeable, and the intermixability of the combined set of operations provides a very powerful and diverse set of tools for manipulating these types among each other.

The APROL language allows for the application of functions across lists of arrays and arrays of lists; this type of parallel record processing may be useful in many applications. Both list and array processing techniques are available in the APROL environment. The reasoning behind this is that most real-world applications do not present sets of data that are of a uniform type and structure; often some particular data type, structure, and set of operations is best for representing one group of data, while other types, structures, and operations are desirable for manipulating other sets of data within the same application. The types of structures and algorithms required by even a single application may differ wildly and evoke different responses among various programmers. The APROL language provides greater flexibility in its tools than either Scheme or J.

One advantage of the J syntax which is not fully retained in APROL is its unique readability. Most J expressions, if one reads the symbols as their associated names, can be verbalized as English sentences; the infix notation and right to left associativity of the operations contribute to this property. However, this characteristic of J may still be of some benefit to APROL programmers. Where associativity must be implied in array operations, the right to left associativity of J-like expressions remains intact as specified in section 3.6. In addition, the use of English names in place of the J function symbols may help to make up for the rearrangement of phrases due to the Scheme syntax structure. In general, applying this J characteristic of verbalization to APROL may assist in determining the correctness of expressions and in translating one's thoughts into APROL programs.

5.2 Topics for Further Research

While the focus of this project has been in sketching out the general nature of the APROL language, this is but the first stage. There is naturally an implied goal to develop a complete and detailed specification of the language and implement a fully functional, efficient interpreter. In light of this goal, several topics for future research are presented.

Scheme and J each use a different representation for Boolean values. In Scheme, `#t` represents the value of *true* and `#f` the value of *false*. In conditional expressions, `#f` is the only value considered false; all other objects are considered to have a Boolean value of true. However, Boolean is one of the eight disjoint data types listed in R4RS[3.4]. Thus the values `#t` and `#f` are equivalent to no other objects outside of conditional expressions. In J, however, the binary values `1` and `0` are used to represent the Boolean concepts of true and false. Thus values which are the results of comparisons can be used in numerical calculations; in fact, this is how the condition construct is implemented in J. In order to preserve Scheme compatibility, it is desirable to avoid compromising the independence and integrity of the Boolean type; however, we also wish to be able to perform conditional computations in the J style. A solution is therefore required which provides functionality over both conventions and/or implements a conversion facility. A technique to consider might be the addition of two adverbs `boolean` and `binary` which will modify a comparator to give results in either convention.

Operators such as − ("negate") are applicable only to numeric elements. However, since abstraction of elements is implicit in APROL, we may wish to consider the possibility of "deep" application of such operators to an array. That is, if an array `a` contains another array `b` as an element, the operator is applied to the element array `b` as well as `a` . If array `b` had another array `c` as an element, then the operator would be applied to `c` as well, and so forth. One should also consider the deep application of more complex functions such as `car` or `cdr` to such an array of lists. It should be noted, however, that potential problems may exist regarding this type of functionality. The J language does not support deep application of this type, and this option has not been fully explored with respect to APROL.

J possesses an interesting iteration mechanism known as `$.` ("suite"). It is possible in J to create a vector of boxed, independent J expressions which resembles an imperative program; such a vector can then be made into a J function and executed. The *suite* is a special variable which, when assigned to a vector of integers, determines the order in which the "statements" of a function defined in this manner are executed. Upon execution of a function, the suite is set to evaluate the expressions in order of their appearance in the function definition. However, by including statements within the function itself to reassign the value of the suite "on the fly," the programmer can control the flow of the program, creating loop structures and branching. This unique approach to repetition may have some powerful applications in APROL. Future research should include consideration of this feature and its implications and how it might be adapted to and implemented in APROL.

The basis of the APROL language rests on two languages which are relatively new and still developing. While the Scheme standard is gradually settling down, the language is not yet set in stone. J, on the other hand, is a very recent development and may undergo significant changes in the near future. While such changes need not necessarily be incorporated into this language, APROL research should include examination and evaluation of changes in these languages should evolutions occur which might be desirable features in APROL.

Appendix A
Programming Examples in APROL

```
: (define a '#(1 2 3 4))
a

: (+ a a a)
3 6 9 12

: (- a a a)
1 2 3 4

: (- a (array-reverse a) a)
_2 1 4 7
```

(Association is left to right.)

```
: ((insert plus) a)
10

: ((insert minus) a)
_2
```

(Association is right to left.)

```
: ((insert cons) a)
(1 2 3 . 4)

: ((insert cons) (array-append a '()))
(1 2 3 4)

: ((insert list) a)
(1 (2 (3 4)))

: (define b (shape '#(2 2 2)
            '#(1 2 3 4 8 7 6 5)))
b

: b
1 2
3 4

8 7
6 5

: ((insert minus) b)
_7 _5
_3 _1

: ((insert cons) b)
  1 2     8 7
( 3 4  .  6 5 )
```

This is a notation issue which has not yet been addressed. Here, the extra spaces surrounding the punctuation symbols signify that the object occupies multiple display lines.

The following illustrates how to find the indices of all the spaces in a string:

```
: (let
    ((text
      (string->vector "This is a sentence.")))
   (copy ((binary eqv?) (string->vector text)
      #\space)
     (integers (tally (string->vector text)))))
4 7 9

: (define c (reverse a))
c
```

```
: ((train plus times minus) a c)
_15 _5 5 15
```
This fork is equivalent to `(a+c)*(a-c)`.

```
: ((train plus signum minus reciprocal) a c)
1.75 2.66667 3.5 4
```
With rationals implemented, the results would appear in ratio form. This sequence is equivalent to `a+(*c)-%y` *(J notation). Note the need to use the monadic functions* `signum` *and* `reciprocal` *rather than* `times` *and* `divide`.

```
: ((with power 2) 5)
25

: ((with power 2) a)
1 4 9 16

: ((compose plus decrement) 3 5)
6
```
This is equivalent to `(<:3)+(<:5)`, *or* `2+4`.

```
: ((compose plus decrement) a c)
3 3 3 3
```

Appendix B
Assisting Procedures in Scheme

String and vector conversion procedures, as described in section 3.4:

```
(define string->vector
   (lambda (s)
      (list->vector (string->list s))))

(define vector->string
   (lambda (v)
      (list->string (vector->list v))))

: (string->vector "hello world")
#(#\h #\e #\l #\l #\o #\space #\w #\o #\r #\l
#\d)
```
Note: In APROL, the output would be simply the following line:
```
hello world

: (vector->string (string->vector "hello
world"))
"hello world"
```

Appendix C
Implementation Status

The current working implementation of the APROL interpreter uses Gambit Scheme 2.0 [GAMB20] compiled with the LinkJ 6.2 system [J62] using THINK C 5.0.4 for the Apple Macintosh. The entire system exists as a single standalone application, and memory is divided in a two to one ratio between the two environments, based on the recommended memory size for each system considered independently. Several functions have been written in C to facilitate translation of array arguments and passing of control between the two systems.

For testing purposes, the only arrays currently supported are those consisting of integer elements smaller than bignums, although any shape and rank greater than zero are allowed. This capability will naturally need to be extended to include arrays of other primitive types, lists, and boxes. The implementation of boxes will require additional recursion in the translation routines in order to capture arrays within arrays (to any depth). Another consideration for future implementation will be the handling of Scheme's exact and inexact numeric representations. Also, little error handling and type checking of arguments is implemented at this time.

The Gambit Scheme system is somewhat unique in that it allows the use of names of integrable procedures for user-defined functions. The integrable functions themselves exist in the compiled interpreter and do not affect the internal operation of the interpreter. However, user-defined functions do take precedence over corresponding integrable functions with respect to user operations; to the user, therefore, it appears as though the integrable functions themselves have been redefined.

With two exceptions, all the standard Scheme vector processing functions have been redefined to work in terms of rank one arrays rather than Scheme vectors. `vector-set!` and `vector-fill!` require a syntax extension or macro facility to implement, as they modify the vector argument in place rather than generate a new vector.

Several functions are classified as APROL system functions; these procedures provide accessors to the array structure and the interface to the LinkJ system. `make-array` is analogous to the Scheme function `vector`; given a shape vector (an integer or an array) and an element or list of elements, it generates an appropriate array. In J style, the element list is replicated or truncated as needed to fill the array.

Since the interpreter has not been modified to read vectors as arrays, the conversion procedure `vector->array` is needed to produce a rank one array from a Scheme vector. This function is intended for use by the user until the appropriate interpreter modifications are implemented; beyond that time, it will not appear in future versions of this implementation of APROL.

The functions `array->j` and `j->array` handle the translation of arguments between the Scheme and LinkJ environments. Since arrays are normally stored in the Scheme environment and J functions take at most two arguments, only three arrays are ever bound in the J name space at any given time. The names used are `aprolx`, `aproly`, and `aprolr`, corresponding to the left argument, the right argument, and the result, respectively. That is, `aprolx` and `aproly` are inputs to LinkJ, and `aprolr` is the output. aprol-set-j-args accepts two arrays and uses array->j to bind them to their appropriate names in the J environment.

The procedures aprol-jx-monad and aprol-jx-dyad serve as the interfaces to LinkJ for executing functions; they accept a string containing the symbolic notation of the J function to be executed and one or two arrays. The value obtained by the application of either of these functions is the array result given by LinkJ (translated to APROL array representation via j->array).

The adverbs rtol, insert, boolean, and binary have been implemented. These functions are described in sections 3.6, 3.5, and 5.2, respectively. insert is implemented as a Scheme function rather than simply appending / to the J function symbol in anticipation of inserting non-J functions into arrays. Also note that boolean and binary currently operate only on scalar values and are given primarily for illustrative purposes.

The predicate array? is functionally the same as the Scheme predicate procedure?. Until interpreter modifications are made, no further distinction can be made without risking an error condition. (The object could be applied to an argument, but if the procedure is not an array, the argument may be invalid for the function in question.) The predicate vector? has been modified to recognize arrays of rank one rather than Scheme vectors.

The basic Scheme arithmetic functions +, -, *, and / have been extended to apply to array arguments. However, although / is defined, J uses floating point notation to express the results of the divide operation, so the currently implemented array representation technique does not support this function. The J verbs shape, negate, take, and drop have also been implemented, as have plus, minus, times, and divide (right-to-left-associating versions of the arithmetic functions already given).

Most supporting functions written in C are, for the most part, self explanatory. Many of these functions have names similar to their Scheme counterparts for clarity (the hyphen is replaced by an underscore). One function which has no Scheme counterpart is aprol_jpr; this function will print the J array specified to the Gambit window, although it is normally used to print aprolr. This function will become obsolete when interpreter modifications are implemented, as array results will automatically be written to the screen in the normal Scheme manner.

Because of the way in which aprol_fill_elements works, a problem exists in the translation from Scheme to J of array objects containing zero as an element. This can be corrected by modifying the function to accept an additional index argument; it is also anticipated that the problem will be fixed when the ability to have boxed elements is implemented. Additionally, caution should be exercised when working with scalar (rank zero) values in the prototype implementation. While some of the procedures incorporate hacks to temporarily handle scalars, objects of rank zero have not been fully considered in the implementation so far; it is anticipated that any conflicts or discrepancies will be resolved as the array type is more fully integrated into the system.

References

[CHUR59] Church, Alonzo. 1959. *The Calculi of Lambda-Conversion.* Annals of Mathematics Studies. Princeton: Princeton University Press, 1941; reprint, Ann Arbor: University Microfilms, Inc.

[CLIN91] Clinger, William and Jonathan Rees, eds. 1991. Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers* 4 (July-September): 1-55.

[GAMB20] Gambit Scheme 2.0. Marc Feeley, Montreal, Quebec.

[HOWL91] Howland, John E. 1991. Proposal for research project in Array Processing Lisp. Trinity University, San Antonio, Texas.

[HUI92] Hui, Roger K. W. 1992. An Implementation of J. Toronto: Iverson Software Inc.

[IVER91A] Iverson, Kenneth E. 1991. *Programming in J.* Toronto: Iverson Software Inc.

[IVER91B] Iverson, Kenneth E. 1991. The ISI Dictionary of J. Appendix to *Programming in J*. Toronto: Iverson Software Inc.

[J3] J 3.4. Iverson Software Inc., Toronto, Ontario.

[J62] J-Source Version 6.2. Iverson Software Inc., Toronto, Ontario.

[SPRI89] Springer, George and Daniel P. Friedman. 1989. *Scheme and the Art of Programming.* New York: McGraw-Hill Book Company.