GENERIC FUNCTIONS BY NONSTANDARD NAME SCOPING IN APL

James T. Kajiya
Assistant Professor
Computer Science Department
California Institute of Technology
Pasadena, California 91125
(213) 356-6841

Abstract

We show how to acheive generic functions as in abstract datatypes (such as the Simula CLASS construct or ADA Package notion) for typeless languages, specifically APL. We do this by altering the standard dynamic scoping of names in APL to a scheme we call downward scoping.

Introduction

There are two features which are often proposed as desirable extensions to the current APL language. The first is a capability for user extension of APL's primitive datatypes, the second is the structuring of the global space of names into smaller, more meaningful units. This paper will show that these two extensions are related and can be economically handled by a single mechanism which is upward compatible and consistent with APL's typelessness. We will discuss a scheme which introduces into APL a modest capability for user extension of the available datatypes. This scheme will appear to the programmer much as the feature of abstract datatypes offered by current typed languages such as SIMULA, Smalltalk, or ADA. We do this however in the spirit of APL, not requiring a series of tiresome and redundant type declarations, but rather an assignment of types by type constructors. During the course of this discussion it will become clear how to use the same mechanism to obtain heirarchically structured namespaces. Furthermore, we do this by avoiding the rather unfortunate situation of having to mix both dynamic and static scope rules. These ideas will be incorporated into an experimental implementation of an Array theory interpreter which we are currently in the process of constructing.

Types and their Services

At no time is the APL programmer forced to specify a large collection of tiresome type declarations specifying the types of various identifiers. Rather, type is associated with a value instead of an identifier--and these values are generated from certain type constructors, e.g. monadic iota for the case of integers; or restructured from constants as in

```
10⍴'ABC'
```

Because datatypes in APL are associated with values rather than with identifiers, we call APL dynamically typed. Languages in which type declarations for identifiers are required we call statically typed. ALGOL would be an example of a statically typed language.

It has been said that type declarations are necessary to manage the complexity of software--but the APL programmer knows that in many cases this complexity is borne from the low level of most computer languages rather than any inherent complexity in the problem. Nonetheless, the point must not be ignored. Even APL programmers will encounter complexity problems if the programming task becomes large enough. As the efficiency of APL implementations grows and as compilers for APL become available we fully expect that APL will be used for tasks of the greatest complexity. Let us then examine the complexity management services that explicit datatyping purports to offer.

Services of types

The services of types are three: protection, selection, and summarization. Datatypes offer a protection service against erroneously applying a function which has no meaning on that type of data: e.g. APL's DOMAIN error. If we attempt to apply "+" to two character vectors then the typing mechanism causes such an error.

A second service that types offer is selection of the right definition of a function to apply to the arguments. The machine instructions for addition of two numbers are different for real and integer arguments. The type associated with the arguments trigger a selection of the proper code as well as possibly invoking a datatype conversion mechanism. If this same mechanism selects among user defined functions with differing definitions but with identical names then we may say that a generic function mechanism is at work. APL currently has this capability with respect to only the primitive datatypes built into the language.

Finally, types--especially user defined types--are often used to summarize key properties of an identifier. This is a particularly effective way to mitigate the horrendous complexity that so often crops up in the use of ordinary computer languages. For example, let us say that two matrices ACCOUNTS and EMPLOYEES, are used to represent information in a typical payroll program. It would not make sense to apply a function HIRE to the matrix ACCOUNTS just as it would not make sense to apply a function AUDIT to EMPLOYEES. Now ACCOUNTS and EMPLOYEES might be arrays which have identical representations in computer memory but nonetheless typing reminds the programmer of the purpose of a particular identifier or value. Because we have no way of introducing user defined datatypes into APL this kind of activity is currently closed to us.

Abstract Datatypes

An abstract datatype facility is a feature which has been included in almost all the new statically typed languages. Essentially it is a way to define not only new datatypes but also the allowable functions that operate on them. This is the Simula CLASS construct. With each CLASS, the state configuration of a particular datatype is specified--the so-called data attributes of the class. Furthermore, the possible functions which can operate on that type are also declared--the so-called procedural attributes of the class. Many different procedures with identical names may be associated with different CLASSes, the selection service assures that the right procedure is selected.

More significantly, the abstract datatype mechanism offers a way of structuring datatypes in a type hierarchy, in Simula this is implemented via the SUBCLASS concept. In a type hierarchy data and procedural attributes are inherited (and possibly synthesized--via VIRTUAL procedures) by a subclass from its superclass. Thus any given function which may be applied to a class may also be applied to its subclasses. It has been said that a datatyping mechanism is only as good as its subtyping mechanism (Codd 1981). We note that many new languages which purport to offer an abstract dataytype feature do not offer subtyping, for example ADA.

With the abstract datatype mechanism, all the features and operations on a particular datatype are encapsulated in a single module, the CLASS definition, aka datatype declaration. Complexity is managed effectively by breaking up quite large programs into small modules each of which comprise the CLASS declaration containing the functions which act as the sole custodians of objects of that type. For example, ACCOUNTS and EMPLOYEES may be two CLASSes in which every function which manipulates them is defined. In particular, they may both have print functions which both have the name PRINT. PRINT ACCOUNTS and PRINT EMPLOYEES would in general have quite different output formats. This is an example of the generic function mechanism.

It is controversial whether APL could benefit from some form of user definable abstract datatype mechanism. APL is a delicate balance of carefully chosen features meshing together in a manner which reminds one of a fine timepiece. The inclusion of user definable datatypes might spoil this balance. We run the real danger that inclusion of such a feature would add a hodgepodge of ad hoc conventions: a situation painfully familiar to the APL user who is unlucky enough to be forced to program in conventional languages!

Our proposal sidesteps the fundamental question whether user definable datatypes are a desirable feature in APL. We present a mechanism in which much of the advantages of abstract datatypes accrue while doing what we hope is minimal violence to the structure of APL. In particular, we do not require the inclusion of any type declarations whatsoever. Rather, we define type constructor functions--like monadic iota--which construct new datatypes from old. Before we present the proposal we must first discuss the issue of namescoping.

Namescoping and its Services

In this discussion we distinguish between an identifier (or name), its reference (or location), and its value (Wegner 1971). The name is simply a string of characters which appears in the program text. A reference is a pointer to a particular piece of storage in the computer memory. A value is simply the value contained in the storage location to which the reference is referring.

The association of references to identifiers in languages is governed by a series of sometimes extremely complex rules, e.g. in ALGOL (Boyle and Grau 1970) an identifier may be a keyword whose denotation is independent of its context; or it may be a formal parameter or local variable whose reference is on the recursion stack; or in the case of PASCAL or PL/I it may be a field of a record in which case its reference is defined in the environment associated with the type declaration of the identifier it modifies. This service of associating references with identifiers is roughly analogous to the selection service of the typing mechanism. For example, in APL a local variable may have the same name as a global. All instances of that name are associated with the local instead of the global, viz. the local name shadows the global. Thus we say that the dynamic scoping rule has updated the environment to one which selects the local reference instead of the global.

Another service offered by namescoping is that scoping rules hide irrelevant identifiers. Thus while local variables appear during a given function invocation, outside of this invocation they are not defined. This is a rough analog of the protection mechanism offered by types. Indeed, the ability or inability to name an object has been used in operating systems as a very effective protection mechanism, we mention the capability based systems (Fabry 1974).

Finally, environments can be used to summarize the processes that occur within them. A workspace is a universal environment in which names are associated with functions and values. Most workspaces summarize the properties of that particular environment by including HOW and DESCRIBE identifiers and associating with them string constants.

Downward Scoping

The above discussion attempts to make a rather vague analogy between the action of types and the action of namescoping. These two mechanisms are generally considered to be quite different. There are a number of characteristics which differ between the two concepts. Environments are associated with a particular activation instance of a function while types are associated with values (in dynamically typed languages such as APL) or with references (in lexically typed languages). Types do not change during invocation of a function, while the process of parameter binding and localization are the principal mechanisms for modifying environments. Furthermore, under expression evaluation, environments do not change but the datatype of a value does. Finally, the strongest reason for the common belief in the distinctness of these two concepts is that environments are really runtime structures while types (in statically typed languages) may be considered to be compile time structures.

In the world of ordinary programming programming languages oriented toward batch execution the distinction between compile time and run time is clear. In interactive languages this distinction is blurred The natural question in view of the above analogies is "Is the distinction between types and environments blurred also?". That is, can we fuse the functions of namescoping and datatyping into a single coherent mechanism which provides the services of both? The scheme we call downward scoping does this.

In downward scoping we associate not a type but an environment with each value. Each environment not only may have local variables which may shadow global variables but also may have local functions which shadow primitive functions defined in the global environment through a scoping discipline, viz. APL's dynamic scoping mechanism. Thus we may think of environments as attached to values. This gives an ENVAL object which is very similar to the LISP FUNARG (Allen 1978) except that all objects and not just function objects are associated with environments.

Let us look at a simple example.

```
PRINT EMPLOYEE
PRINT PAYCHECKS
```

Here the parse tree looks like

In downward scoping the environment of an identifier is calculated from the environments passed up the parse tree. This is in contrast to lexical scoping in which the environment of an identifier comes from upward the parse tree, viz. a BEGIN END block. In Knuthian terms the environment associated with an identifier is synthesized from its subtrees (Knuth 1968). Now with EMPLOYEE we have somehow associated an environment in which a local function PRINT is defined. Similarly in the environment attached to ACCOUNTS a different local function PRINT is defined, say with a different output format. How we set up the environments will be explained in the next section. Thus we have simulated the selection mechanism of types by identifier scoping. This gives us a generic function mechanism.

What about the protection services? Let us try to evaluate the expression

```
PRINT UNPRINTABLE
```

where the environment attached to value of UNPRINTABLE has no local PRINT function defined in it. Then instead of getting a domain error, we get a syntax error: PRINT is an unbound identifier and most APL implementations assume it to be an array--thus we elicit a syntax error.

Finally, summarization may be accomplished simply by including a DESCRIBE local variable in the environment attached to a value. So that

```
DESCRIBE VAR
```

will print out a description of the variable VAR.

The downward scoping rule is stated as follows:

> The environment of a function identifier is determined from the environments of its arguments by the following procedure: Look up the function identifier in the environment attached to one of the two arguments which is most deeply nested in the dynamic chain. If both environments are equally nested then resolve to the right.

If the function is monadic then the rule looks up the function definition in the environment attached to its single argument. This corresponds to the SIMULA or Smalltalk rule. For dyadic functions the Smalltalk rules always resolve to the left, treating the other argument as a parameter. In our rule we may resolve either way. If an argument is attached to an environment of nesting level 0, i.e. it is a global variable,then it is a primitive datatype. Thus when a dyadic function is applied to a primitive datatype and a nonprimitive datatype (a value whose attached environment is at a higher nesting level), then the rule invokes the function defined in the nonprimitive environment.

Problems with APL Dynamic Namescoping.

The dynamic scoping rules in APL are similar to a great many other interactive languages, e.g. LISP,SNOBOL. These languages suffer from the annoyances of flat namespaces. That is, environments cannot be effectively structured in a hierarchical manner using the dynamic scoping rule. Certainly, if an identifier occurs purely locally then it may be safely tucked away in some function. However, if there needs to be a communication between two different function invocations then the only way for those functions to communicate is via global variables. Furthermore, if two functions call a common function then the choice is to either define the subfunction a multiple number of times or, as is usually done, to define the function in the global environment. Thus, all of us have had the experience of loading an application workspace to be confro..ted with a bewildering collection of all sorts of bizzare, meaningless names--names which denote internal functions or variables--that should not appear in the global environment. More seriously, if these names are not bizzare and meaningless then they are likely to clash with natural, meaningful names one has already chosen before loading the package. This name clash will then cause unpredictable redefinitions. Of course, such a clash effectively ensures that no two related application packages may be loaded in the same workspace.

It is for these reasons that proposals to heirarchically structure namespaces is high on the top ten list for APL extensions (Orgass 1977). How can we hierarchically structure our environments so that use of reasonable names in application packages will not be punished by name clash? How how can we avoid the unnecessary proliferation of global names? The usual proposals include some sort of lexical scope mechanism. Unfortunately this complicates the identifier semantics tremendously. Instead let us look for solutions consistent with the practice of APL. We show how one can construct a hierarchical namespace structure with dynamic namescoping only.

Setting up Hierarchical Namespaces

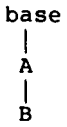
Under the dynamic scope rules we are able to hierarchically structure environments in a very limited manner. The environment of three functions A,B, and C are nested within one another if they call each other,e.g. if A calls B which in turn calls C then one may picture a skinny tree (with branching ratio 1) as the hierarchical structure of the environments. It is clear that we would not be able to set up any other structures.

We would like at least to be able to structure our environments as trees. Since the environment scoping follows the call chain, this is accomplished simply by extending the allowable control structures. Thus we extend the possible control forms to include not only a function call but also a coroutine mechanism. The introduction of coroutines into a dynamic scoping discipline allows one to structure environments in other than a strict first in first out sequence.

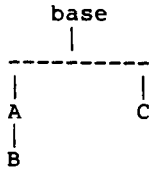
In our proposal let all the current primitive function definitions be included in the BASE. The base is the control level at which the user normally interacts, viz. the interpreter. Let us say that the user types the expression "A 1" then during the execution of A, the following environment is set up:

and any expressions evaluated during the execution of A reference identifiers either defined locally or predefined in the base. When the function returns, the current environment is restored to the base.

Now let us look at an example which includes a coroutine. Say the user types an expression containing A which locally defines and then calls B, furthermore B, instead of performing a simple return, performs a coroutine RESUME. This sets up the environment shown as follows:



Note that although B has relinquished control, its environment has not been popped from the dynamic chain. Function B has been suspended but its Llc although retained is not accessible. We are again at the top level of APL. Now if an expression which calls a function C is typed to the interpreter the following environment structure will be set up during the execution of function C.



Using the dynamic scoping discipline we see that B is able to access all the identifier definitions in A and in the Base, while A and C are able to reference only either local definitions or definitions in the base. A and C are not able to reference the locals defined in each other. Thus we have set up a tree of hierarchical definitions through the coroutine control structure. Note that C is able to call A, since the definition of A occurs in the base, but that C is not able to call B since it was locally defined in A.

Using the coroutine call and resume mechanisms we can structure namespaces into any arbitrary unordered tree.

Downward Scoping

We attach environments to values via the downward scoping rule. This in essence is the 'type' of the data value--it contains a summary of all the relevant operations and state structure of that value--which may contain comments.

Type Hierarchies: Subclassing

Datatypes have more than a simple structure. They can also be hierarchically structured. In SIMULA this corresponds to the SUBCLASS construct. Let us examine an example of a SUBCLASS object.

```
CLASS A ....;

A CLASS B ....;

B CLASS C ....;
```

The Proposal

We now possess a scheme which accomplishes generic procedures and abstract datatyping mechanisms without imposing an artificial type structure on typeless languages. It does this by combining the notion of type with that of scoping environment. Basically, the proposal mimics the action of types by assigning environments to values. In statically typed languages types are explicitly declared. In abstract datatypes this declaration includes procedural attributes which express the allowable functions which can operate on the data. In this proposal the type declaration would be set up via the dynamic scoping rules of environments. We build up environments hierarchically during the execution of a type constructor and then attach them to the value calculated via the downward scoping rule. A coroutine RESUME is executed at the end of each type constructor to ensure that the environment of the type is retained, viz. the local functions and variables which serve as attributes of the type are kept.

In this example C is a SUBCLASS of B which in turn is a SUBCLASS of CLASS A. Thus among the attributes accessible to C are those accessible to B. B in turn may define locals and inherit attributes from A. How is this to be accomplished in our dynamic scoping scheme? We cannot simply call the procedure defining C from B and B from A for although the nesting of the attributes would be correct by the standard dynamic scope rules, the global names would not be right. Specifically, B and C would not be accessible from the base. Thus we could not construct an object of type C simply because we need to

call A and B first. If we were to call B from C and A from B then the constructor for C would be accessible from the base but the scoping would be backwards--an attribute of class C would not be able to access those of A. The solution is contained in the following code.

```
∇Z←C
  ∇Z←SUBSUBCLASS;DATA ATTRIBUTES OF C
    ∇FUNCTION ATTRIBUTES OF C
    ∇
    SUBSUBSUBCLASS
  ∇
  B
∇

∇Z←B
  ∇Z←SUBCLASS;DATA ATTRIBUTES OF B
    ∇FUNCTION ATTRIBUTES OF B
    ∇
    SUBSUBCLASS
  ∇
  A
∇

∇Z←A;DATA ATTRIBUTES OF A
  ∇FUNCTION ATTRIBUTES OF A
  ∇
  SUBCLASS
∇
```

We assume in the base that there are defined functions named SUBCLASS,SUBSUBCLASS,SUBSUBSUBCLASS,etc. each of whose definition looks like

```
∇Z←SUB ···· SUBCLASS
 ⎕RESUME ∇
```

Let us now follow the execution of an invocation of the type constructor C. First the function C is called and the following definition is set up:

```
base: SUBCLASS,SUBSUBCLASS,
  |       SUBSUBSUBCLASS,
  |       SUB...SUBCLASS
  |
  C: SUBSUBCLASS
```

then C calls B and the following definition is set up:

```
base: SUBCLASS,SUBSUBCLASS,
  |       SUBSUBSUBCLASS,
  |       SUB...SUBCLASS
  |
  C: SUBSUBCLASS
  |
  B: SUBCLASS
```

C then calls A to get the following definition:

```
base: SUBCLASS,SUBSUBCLASS,
  |       SUBSUBSUBCLASS,
  |       SUB...SUBCLASS
  |
  C: SUBSUBCLASS
  |
  B: SUBCLASS
  |
  A: data attributes,
     function attributes
```

A then calls SUBCLASS which defines the attributes for B.

```
base: SUBCLASS,SUBSUBCLASS,
  |       SUBSUBSUBCLASS,
  |       SUB...SUBCLASS
  |
  C: SUBSUBCLASS
  |
  B: SUBCLASS
  |
  A: data attributes for A,
  |  function attributes for A
  |
  SUBCLASS: data attributes for B,
            function attributes for B
```

This goes on until finally SUBSUBSUBCLASS is about to be called in the SUBSUBCLASS routine.

```
base: SUBCLASS,SUBSUBCLASS,
  |       SUBSUBSUBCLASS,
  |       SUB...SUBCLASS
  |
  C: SUBSUBCLASS
  |
  B: SUBCLASS
  |
  A: data attributes for A,
  |  function attributes for A
  |
  SUBCLASS: data attributes for B,
  |       function attributes for B
  |
  SUBSUBCLASS:data attributes for C,
              function attributes for C
```

Finally SUBSUBSUBCLASS is called, it has not been shadowed by any other definition so the predefined function is executed. Thus we simply resume. So finally we are left with the following environment.

```
----base: SUBCLASS,SUBSUBCLASS,
|   |       SUBSUBSUBCLASS,
|   |       SUB...SUBCLASS
|   |
*   C: SUBSUBCLASS
    |
    B: SUBCLASS
    |
    A: data attributes for A,
    |  function attributes for A
    |
    SUBCLASS: data attributes for B,
    |       function attributes for B
    |
    SUBSUBCLASS:data attributes for C,
    |         function attributes for C
    |
    SUBSUBSUBCLASS
```

This example illustrates the semantics of how the subclassing mechanism would be set up. Syntactically, this is an unacceptable mode of definition. We leave it up to the reader to devise some pleasant syntax for the above actions. For example, Smalltalk syntax would do nicely (Ingalls 1978).

Examples of Generic Functions at Work

We now present an example of user definitions of generic functions. Suppose that the user wishes to include the complex numbers as an APL datatype. Using the current proposal the programmer would define the function J, which takes two arguments and returns a complex number. Here is the code for J:

```
∇Z←R J I
   ∇X←α × B·
   X←(Z[1;]×RE B)-Z[2;]×IM B
   X←X J (Z[1;]×IM B)+Z[2;]×RE B
   ∇

   ∇X←A × ω
   X←(Z[1;]×RE A)-Z[2;]×IM A)
   X←X J (Z[1;]×IM A)+Z[2;]×RE Z
   ∇
   .
   .
   .
   ∇X←ρ ω
   X←1↓ρZ∇

   ∇X←RE ω
   X←Z[1;]∇

   ∇X←IM ω
   X←Z[2;]∇

   ∇DESCRIBE ω
   'THIS IS A COMPLEX ARRAY' ∇
→ERROR×ιI MATCH J
Z←(R EXPAND I)ρR
Z←Z,[.5](R EXPAND I)ρI
SUBCLASS

ERROR:'SHAPE ERROR'
∇
```

The above routine is written in a pseudo-APL syntax. The routine called J generates a complex array, e.g.

```
Z←1 0 ¯1 0 J 0 1 0 ¯1
```

generates a vector with items the four units of the ring of complex integers. Each indented function definition is intended to represent the dynamic definition of the various extended functions. Note that multiplication is defined twice, once for the left argument and once for the right argument. The shape function is also redefined to return only the shape vector with the first coordinate removed. This assures that a complex vector will display the correct shape. The functions MATCH and EXPAND are intended to test whether the shapes of the two arguments are compatible: a scalar or vector of length one is compatible with any array, otherwise they must match. EXPAND calculates the correct shape for the vectors. Also in the base we must define two new functions RE and IM which are the identity and the zero function respectively. This complex number example illustrates the generic function capability. The multiply and shape functions are redefined for complex objects but not for ordinary arrays in the base. Downward scoping keeps track of which function to apply.

One should mention that the user, in order to get complex datatypes, must redefine practically all of the functions, plus all the operators. This illustrates why such comprehensive datatype extensions such as complex numbers and nested arrays are best done in the interpreter instead of by the user. Ordinary languages are able to get away with user extensions because relatively few primitive functions and no operators are defined on any given datatype. APL has an unusually wide set of functions and operators, asking the user to redefine them all for each comprehensive datatype is really not the right thing to do. Thus we envision the downward scoping scheme not as a way to eliminate the datatype extensions currently under consideration (viz. complex numbers (Penfield 1979) and nested arrays (Gull and Jenkins 1979)) but rather a way for the user to implement a number of temporary datatypes useful to manage his program complexity in a convenient way.

Advantages/Disadvantages of Proposal

The first disadvantage of the downward scoping proposal is that it is syntactically complicated--particularly for the case of type hierarchies. We leave it to the reader to determine what syntactic form should be adopted for type hierarchies as well as dynamic nested definitions of functions.

The second disadvantage is that the proposal is not really a complete extension. Major datatype definitions such as complex numbers require complete redefinition of almost all the functions as well as the operators. We point out that any user datatype definition scheme will suffer from this same problem to varying degrees. In any language with a rich set of functions and operators, basic datatype extension is best done in the interpreter. APL is a delicate balance of carefully chosen functions that mesh well. Requiring user extension for basic datatypes results in a patchwork of incomplete and arbitrary restrictions. This can do much to spoil the beauty of this balance. Our proposal on the other hand gives us a method for making temporary extensions to datatypes as a means of handling program complexity. Granted, it is a weaker extension facility but it also has the advantage of not disturbing the delicate balance of functions and operators.

Finally, there is a technical snag in the definition of the downward scoping rule. In APL it is impossible to tell if an identifier references an array or a defined function until its looked up in an environment. During the parse of an expression such as A F B, how does the interpreter know whether this is an

instance of a dyadic function applied to two arguments or two monadic functions succesively applied? The interpreter decides which case holds upon scanning F, according to whether F is bound to a dyadic or monadic function in the current environment. The downward scoping rule complicates this procedure. This is because the environment in which to look up F is not determined until the deeper of the two environments associated with A and B are determined. This causes some messy backtracking to occur in the implementation.

There are three advantages of the above proposal. First, this proposal is an (almost) upward compatible extension. Almost all existing APL programs will execute using the downward scoping discipline. This is simply because all environments of identifiers are either in the base or on the top of a call chain. Without a way to hierarchically structure the environments downward scoping collapses to dynamic scoping. The cases which are not extendable are those functions in which local function definitions are made via the system function "FIX" (Jaffe 1979). These cases can be suitably isolated by a simple syntactic scan.

A second advantage of this proposal is that the hierarchical environment structures handle the name clash problem associated with flat namespaces. Thus a persistent problem in APL is solved not by introducing an ad hoc mechanism to handle it but rather introducing a feature which adds real power to the language while solving the problem.

Finally, a third advantage of this proposal is that the coroutine control mechanism is added to the language. In contrast to other control extension proposals motivated by current and past doctrines of structured programming, the coroutine mechanism has been found in practice to be actually useful in simplifying many programming tasks, especially a number of interactive programming paradigms (Lindstrom 1978).

Conclusions

Viewed in its most beneficial light a type is a characteristic of data which is used to disambiguate a function identifier associated with a function application. Unfortunately for most typeless languages this forces a type declaration--something which we would like to avoid in APL. The use of downward scoping enables environments to disambiguate function names simply through a scoping mechanism which is a straightforward extension of the dynamic scoping discipline.

References

J. Allen (1978) Anatomy of Lisp, McGraw-Hill, New York.

J.M. Boyle and A.A. Grau (1970) "An Algorithmic Semantics for ALGOL 60 Identifier Denotation" J. ACM v.17,No.2 April 1970, pp.361-382.

E.F. Codd (1981) "The notion of type in databases" Proceedings of the workshop on data abstraction, databases and conceptual modelling. Pingree Park, Colorado June 23-26, 1980. SIGPLAN Notices v.16, No.1 p.47.

R.S. Fabry (1974) "Capability based addressing" Comm. of the ACM v.17, No.7, pp. 403-412.

W.E. Gull and M.A. Jenkins (1979) "Recursive Data Structures in APL" CACM v.22, No. 2, pp. 79-96.

S.B. Jaffe (1979) "Applications of Local Functions in APL" Quote Quad v.10, No. 2, pp.26-29.

D.H. Ingalls (1978) "The Smalltalk-76 Programming System: Design and Implementation" Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, pp.9-16.

D.E. Knuth (1968) "Semantics of Context Free Languages" Math. Systems Theory v.2, No.2, pp.127-145.

G. Lindstrom (1978) "Control Structure Aptness: a case study using top-down parsing" Conf. Software Eng., Atlanta, Ga., pp.5-12.

P. Penfield Jr. (1979) "Proposal for a Complex APL" APL79 Conference Proceedings, Rochester, New York, Quote Quad v.9, No. 4-Part I. pp.47-53.

R. J. Orgass (1977) "The 1E6?1E6 APL Workshop: another view" Quote Quad v.8, No.2, pp.8-11.

G.M. Seeds, A. Arpin, and M. LaBarre (1978) "Name Scope Control in APL Defined Functions" Quote Quad v.8, No. 9, June 1978, pp. 15-19.

P. Wegner (1971) "Data Structure Models for Programming Languages" Proc. Symp. on Data Structures in Programming Languages, SIGPLAN Notices v.6, No.2, pp.1-54.