SUGGESTION FOR A "MAPPED" EXTENSION OF APL

C. Leibovitz
University of Alberta
Computing Center

Users of APL are under the "spell" of beauty, conciseness and elegance of the language. They have however to go back to Fortran, for instance, whenever they cannot avoid a loop executed a great number of times in order to limit the CPU time used.

The natural desire for enlarging the class[1] of cases in which "it would pay" to use APL, is the origin of a great number of suggestions for modifications to and extensions of APL.

However, an APL interpreter is a complicated collection of interrelated software forming a unity that should not be disrupted. A modification in any part of the collection will have repercussions on the operation of the remainder of the programs and there is no a priori reason preventing these repercussions from being harmful and in need of necessary corrections that may not be welcomed (if at all possible).

It is therefore not enough to show that a given modification is needed; it is necessary to show that it is indeed implementable and has no disruptive character.

Our proposed modification is, in a sense, a "mapping" of an actual interpreter. The logical structure of the mapping is such that we may conclude that: "if there exists an APL interpreter that works, then our mapping will work too."

Review Of A Non-Modified APL Interpreter

Each time a line is entered from a terminal, the interpreter checks the nature of the line: is it for instance a command? or a line in definition mode? or in execution mode?... Let us designate by CHECK the module of the interpreter that finds out the nature of a line and decides what other module is to handle the line. If the line has been entered in the execution mode, it will be executed from right to left. However, for a number of reasons, this cannot be a straightforward procedure:

1. There is no one-to-one correspondence between a "primitive mathematical symbol" and an execution routine. One same symbol may be a monadic function or may be a dyadic one.

2. The mathematical meaning of a symbol may depend on the nature of another symbol placed at its left.

3. There may be brackets altering the normal right-to-left order of operations.

4. There may be mistakes in the line making it unexecutable.

There must therefore exist a module that will analyse the line, will call execution routines in a proper order and provide those routines with the values of the variables.

We are not concerned here with the way in which this is done, it is enough for us to know that it is actually done, i.e. there exists in the interpreter a module, we call it GRAM, that takes care of a line in execution mode. GRAM issues "orders" for space, for fetching values for parameters and variables, for erasing intermediate unnecessary results, for storing needed intermediate results, for finding out which routine is to be called, for calling it, for issuing error messages.

We thus designate by GRAM all the parts of the interpreter that stand between a line recognized in execution mode, and its actual execution. Everything the computer does in execution mode is therefore the consequence of "orders" issued to the computer by GRAM while analysing an entered line in execution mode.

The Need For A Modification

The correct execution of a statement results from the collection of correct "orders" issued by GRAM in a correct sequence. However, the main work done by GRAM is not so much to issue those orders but to find out which orders are to be issued.

In the MAPPED LEVEL (the name we give to our APL modified version), the function is to be stored in such a way that the orders to be executed, and their proper sequence, is known in

advance. The execution of the function thus becomes faster because there is no need for syntax checking time.

Mapped Level

We recall that CHECK examines the nature of a line and delivers it to GRAM if the line is in execution mode. CHECK has of course other alternatives than calling GRAM. We will not modify the existing alternatives; we will add one alternative more that we call Mapped Level. It means that once a line is entered, CHECK will ask an additional question: is it a mapping command? A negative answer will result in the unmodified procedure going on. A positive answer will result in a modified procedure described below.

It may be possible later to allow the use of the mapping command to all users. However, in order to simplify our discussion we will consider the case in which the mapping command is available to a priviledged APL user.

Using the mapping mode, the user can form a library of "mapped functions" that cannot be edited or modified but can be executed by any APL user.

When the user issues the mapping command, he must add two "parameters" which are the name of the unmapped function and the name under which the mapped function will be stored. The function to be mapped either does not call for another function or calls for a number of functions that have already been mapped. The list of all mapped functions is stored in the symbol table in the workspace of the priviledged user.

The mapping command will "deliver" the function to be mapped to a module we call MAPGRAM to indicate that, in a sense, this module is a mapping of the GRAM module.

MAPGRAM will proceed to analyse the lines of the function in the way GRAM would have done it with the following differences.

1. MAPGRAM considers all symbolic names as defined and does not issue value-error messages. Every symbolic name is compared with the symbol table of mapped functions. Depending if the symbolic name exists or does not exist in the table, MAPSYNT will respectively consider it a defined function or variable.

2. MAPGRAM will analyse lines of a function already tested in the unmapped mode by the user. This function is supposedly syntax-error free (this concept will be discussed later.) Therefore, for proper values of the arguments, GRAM would have issued a number of "orders": fetch, store, reserve storage place, call for execution routine, erase, etc...

MAPGRAM will issue "mapped" orders that could be described by: "copy and store in proper order the 'orders' that GRAM would have issued." For instance, whenever GRAM would have called for storage, MAPGRAM will order to store a copy of the call for storage space; wherever GRAM would have called for a given execution routine, MAPGRAM will order to store a copy of this execution routine.

In short, the mapping of the function will consist of the collection in proper order of copies of fetching routines, store routines, execution routines, etc...

These routines will be linked either by MAPSYNT or by the module LINK active at execution mode for mapped functions. The linkage consists of taking care of the proper order and of the addresses of the intermediate results and transforming the copy of a call into an actual call of a routine. It must for instance insure that the output address of a given execution routine may have to be identical with the input address of the next execution routine.

In short LINK takes care of a mapped function in the execution mode. LINK is called every time the name of a mapped function appears in a line at execution time.

Error Messages

APL delivers two kinds of error messages. The first kind corresponds to what we call a "built-in error". It is delivered when GRAM concludes that there does not exist an execution routine corresponding to the symbols entered in the line. This kind of error will be delivered for instance if there is, at execution time, a symbolic name not yet defined or if a line is entered with mathematical symbols in a non-sensical sequence. The second kind of error messages is delivered by an execution routine when GRAM does find out, at a given stage of execution, that execution routine is to be called and when this routine cannot be executed for the values and number of arguments delivered to it (rank error and domain error for instance).

The built-in errors can be detected during the mapping operation by MAPGRAM in exactly the same way as GRAM is doing it, i.e. by taking over in MAPGRAM the procedure followed by GRAM in this case. The error message could display the faulty line and indicate the place where the error has occured.

This however cannot be done for the second kind of errors. They can be detected at the mapped level only during execution time. The function is then stored differently and there is no record, at this mapped level, of the form in which the function was entered unmapped.

However, this kind of error would have been detected at the unmapped level by an execution routine which could tell the nature nature of the error (rank or domain) and since we have at the mapped level a copy of the execution routine, it is still possible to deliver at this level an error message containing the following information.

a. The nature of the routine that has detected the error (addition or multiplication or iota operator routine etc...)

b. The nature of the error (rank error or domain error)

c. The values of the arguments for which the error was detected.

This means that the copies of the execution routines stored at the mapped level have to be slightly modified in their error message subroutines.

If the user is mapping functions already tested at the unmapped level and if he checks that all functions called by the one he is mapping have already been mapped before, there will therefore be no error message delivered during the mapping process; those are the functions referred to before as Syntax-error-free functions.

The Advantages Of The Mapped Level Suggestion

The Mapped Level modified APL has many of the advantages of a compiler while being quite distinct from it.

It is clear that the execution of the functions will be much faster at the mapped level. The fact that the syntax analysis has been done makes them close to compiled functions. However there is this important difference between the mapped level and a compiler: A compiler delivers an object program in the machine language that can be directly executed. In particular the compiled function should have all the needed instructions for storage handling, whereas a function stored at the mapped level is still in need of the module LINK at execution time.

It is also clear that the interractive feature of APL is not disrupted by the introduction of the mapped level as it would have been with the use of a compiler. In the case of most Fortran compilers for instance, alternating orders of compiling, executing, compiling, executing etc... require successive loadings of the compiler. In our case, the same interpreter will remain loaded in the computer while mapping or executing.

Another advantage is the flexibility of the combination of the two levels; in particular, it facilitates the editing and debugging process. A function can be tested and displayed at the unmapped level; the faulty line is then displayed with an indication of the place and the kind of error. It is then possible to execute parts of the line instead of executing the whole function. Such a facility would not have been available with a compiler. Once edited and debugged, the function may be stored at the mapped level.

ACKNOWLEDGEMENTS

The author is indebted to Dr. W.S. Adams, Dr. D.H. Bent and to Mr. G. Gabel for suggestions and fruitful discussions.

REFERENCES

1. In the Computing Center of the University of Alberta, a 360/67 IBM computer is used (mainly under M.T.S.). The c.p.u. time needed for loading an object program from a file is greater than the loading time needed in the APL case. There is therefore a class of programs that would take less time to be executed with APL than with a FORTRANG generated object program (if loading time is added to the execution time).