THE IBM PERSONAL COMPUTER APL SYSTEM

M. Tavera, M. Alfonseca and J. Rojas
IBM Madrid Scientific Center
P. Castellana, 4.
Madrid-1. SPAIN.

INTRODUCTION AND HISTORY

The Computer Science Department of the IBM Scientific Center in Madrid has been working for several years in the field of APL-based compilers and APL interpreters.

In 1973, we decided to enhance the software of our System/7 computer with one interactive time sharing system, and chose APL as the target language. System/7 is a sixteen bit minicomputer embodying a disk unit and multiple sensor based facilities, including analog and digital input/output as well as process interrupts. It was our purpose to support all those facilities within our system, in such a way that APL could become the main program development language for those projects in the Scientific Center using this minicomputer.

At that time, System/7 had a maximum storage of only 16k 16-bit words. Therefore, the interpreter was divided into relocatable, 128 word pages. Control transfer between pages was done through a 'virtual storage' supervisor, which made sure that the requested page was present in main storage. The number of pages simultaneously active was a function of the actual size of the System/7 storage.

THE MACHINE INDEPENDENT APL INTERPRETER

In 1976, after the APL/7 system was completed, we received certain suggestions concerning the possibility of starting a similar project for a different minicomputer system. Since the new computer had a completely different machine language, the APL/7 system could not be directly migrated. Furthermore, the problem could recur in some years whenever new minicomputers were available.

In order to prevent this development, at the same time providing a general solution to the problem of the construction of an APL interpreter, we decided to follow a systems programming approach along the following lines:

1. An interpreter is written in an intermediate machine independent language (IL).
2. A compiler is written translating IL programs into the assembly language of a given machine.
3. The IL interpreter is compiled. The generated code is an APL interpreter written in that assembly language and, therefore, directly executable on that machine.

To obtain an APL interpreter for a different machine, only the code generator of the compiler in step 2 must be rewritten, and step 3 must be repeated.

THE INTERMEDIATE LANGUAGE

Once the systems programming approach was chosen for our implementation, we had to select an appropriate intermediate language. Several possibilities were considered, including Macro Languages, Systems Programming Languages known at that time, as well as high level languages. However, none of these was fully adapted to our purposes, and we finally decided to design our own language, which was chosen on the basis of the following criteria:

- It must provide machine independence.
- It should allow easy management of different types of data, required for the implementation of an APL interpreter.
- Compilers for the intermediate language should be easy to build. To optimize performance, they should not introduce an unreasonable overhead in the code they generate.

A special systems programming language (IL) was designed to meet all the requirements of our problem. It is endowed with

- A high level syntax, thus making programming, debugging, and readability easier, at the same time providing machine independence.
- A low level semantics, thus reducing the programming effort required to build compilers that produce highly efficient code. In fact, IL can be considered as a 'High Level Assembly Language'.

The primitive operations of the language are those most commonly encountered in current assembly languages. But they have been given a high level syntax.

The only assumption about the target machine is that its memory is considered to be a vector of units of fixed size, consecutively numbered. The number of bits in a memory unit should be a power of two, but is otherwise undefined. The language supports several types of data objects: numeric and literal constants, parameters, integers of different sizes, floating point objects, pointers, as well as internal and external label names. Some of these objects may be vectors of numbers, addresses or literals. Others (parameters and labels) are only allowed to be scalars.

An IL program consists of two different parts: declarations and executable statements. All variables and parameters appearing in a program must be declared. Their type is implicit in the first letter of the name. Two different declaration statements are supported.

- Assignment of initial values to static variables.
- Assignment of synonyms to a previously defined variable name. Different combinations of data types within this instruction make it possible to define pointer based variables; vector variables of variable length; and variables of different types sharing the same address.

The executable section of an IL program consists of a number of executable IL statements. These are analyzed from right to left. There are no precedence rules, and parentheses are not allowed. The following primitive operations are supported:

- Assignment, increment, decrement, and point to.
- Arithmetic operations: Addition, substraction, multiplication, division, and residue.
- Bit shift to the right, or to the left.
- Bit to bit logical operations: OR, AND, NOT, exclusive OR.

The only control statements are the following:

- Unconditional branch.
- Conditional branch.
- Computed go to.
- 'On overflow' condition.
- Subroutine call and return.

THE APL/IL INTERPRETER

A complete APL system cannot be made fully machine independent. There remain several operating system dependent functions, too close to the machine level, which must be left outside our procedure. We call the set of all these functions an APL supervisor. They include

- System initialization.
- Time sharing control, if needed.
- Sign on/off procedures.
- Terminal input/output.
- Support of workspace libraries.
- Timer routines.

The APL supervisor amounts to 5% to 10% of the total programming effort. The remaining functions in the APL system, equivalent to 90% to 95% of the programming effort, make up what we call the Interpreter. This has been programmed in IL, and debugged on a System/370 computer.

THE LANGUAGE

Our machine independent APL interpreter follows the APL standard, with the following specifications, differences and enhancements:

- Implementation dependent limits
  1. The maximum number of characters in the name of an APL object is 12.
  2. The maximum rank of an APL variable is 63.
  3. The maximum number of lines in a function is 1000.
- Restrictions.
  1. The value of Quad HT has no action on terminal input/output.
  2. Quad TT always gives a result of zero.
  3. The following system commands are not supported:
     -)COPY and)PCOPY. They are replaced by)IN and)OUT.
     -)GROUP,)GRP and)CRPS
     -)CONTINUE and)CONTINUE HOLD
  4.)FNS and)VARS list the corresponding names in Quad NL order. A first letter cannot be mentioned.

- Enhancements.

  1. Multiple line deleting is supported in function definition.

  2. Ambivalent defined functions (dyadic-defined functions that can be used monadically). The function itself may discover the valence of the call by applying system function Quad NC to its left argument.

  3. New APL primitives:

     - Dyadic grade-up and grade-down.

     - Picture format, a powerful output generating facility.

  4. New APL system functions:

     - Execute alternative (Quad EA), the error trapping system function.

     - Peek and Poke of memory contents, plus the execution of machine-code subroutines, through a new ambivalent system function (Quad PK).

     - Standard data transfer form (Quad TF), which provides conversion between the internal form and the transfer form of APL objects.

  5. New APL system commands:

     -)RESET clears the state indicator and is equivalent to as many right arrows as asterisks are there in the indicator.

     -)OUT produces a file of APL objects in transfer form.

     -)IN copies into the active workspace some or all of the APL objects included in a transfer file.

  6. Four data types are supported: boolean (packed one bit per element), integer (two bytes per element), real (in floating point format, eight bytes per element), and literal (one byte per element). The system automatically performs conversions of numeric data-types to minimize storage space.

THE ELASTIC WORKSPACE EXTENSION

Since the machine independent APL interpreter can be applied to many different computers, including those with an address space limited to only sixteen bits, something had to be done to increase the size of the active workspaces in such machines. Therefore, we devised the concept of an elastic workspace.

The APL active workspace is considered to be divided into two parts: one (the main workspace, MWS) where normal computations take place. The other (the elastic extension EWS) where APL objects not currently in use may be stored. This extension may be located either in main storage or in a fast direct access device, depending on the actual implementation.

Under normal circumstances, the system forgets about the existence of the elastic extension, and works only on the main section. Whenever a workspace full condition arises, the following procedure is followed:

- A garbage collection is performed.

- If the workspace full condition has subsided, the operation is reattempted, and the system goes on as before.

- Otherwise, the user defined APL objects (functions or variables) in the MWS not currently needed, are copied into the elastic extension, and erased from the MWS. The location of the APL object in the elastic extension is stored in the symbol table of the main workspace. A new garbage collection is then performed and the operation is again reattempted.

- If one of the objects in the EWS is again needed, it is copied into the main workspace.

- Finally, whenever one of these objects is changed, redefined or erased, the new value, if any, is generated in the MWS, and any old copy in the EWS is deleted.

The elastic workspace is managed dynamically, with automatic reusability of any free space it may contain. When an object is deleted there, the space it occupied becomes available for other purposes.

APPLICATION TO REAL ENVIRONMENTS

Our interpreter writing system has been applied to generate APL interpreters in three different environments:

1. IBM System/370, which has been used as a test case.

2. IBM Series/1.

3. IBM Personal Computer.

THE IBM PERSONAL COMPUTER APL SYSTEM

Conversations with Entry Systems Boca Raton towards the application of our procedure to the IBM Personal Computer began in September 1981. The authors began

working in the interpreter in December 1981, when we received the first Personal Computer at the Madrid Scientific Center.

Since the IBM Personal Computer is based on the INTEL 8088 microprocessor, we wrote a compiler to translate IL code into 8088 assembly language. However, we had to take a decision concerning the possibility of supporting the mathematical coprocessor (the 8087 chip) to enhance the performance of floating point operations. Although the chip was not fully available at that time, we decided to run the risk. Therefore, we wrote the compiler to translate IL code into both 8088 and 8087 instructions. We wrote the compiler in APL and executed it under VSAPL at an IBM/370 computer. The total cost of this step in the translation process was about one person-month.

Once the compiler was available, we translated all modules of the APL interpreter into assembly code for the Personal Computer and sent them to our Personal Computers via a communication link. Then we assembled and link-edited all those modules, together with four assembly written routines making up the machine independent supervisor.

We finished the first version of the product in May 1982. It was then that we decided to include several enhancements, such as picture format, dyadic grades, and several auxiliary processors. During the remainder of 1982 and the first quarter of 1983, we implemented the additions, performed a large testing and debugging cycle, and documented the system, all of this in Madrid. The final version of the IBM Personal Computer APL book was completed by February 1983 in Boca Raton.

SYSTEM FEATURES

The IBM Personal Computer APL system works on both the IBM Personal Computer and the IBM Personal Computer XT and supports the APL keyboard, the US keyboard and all the European National keyboards.

Personal Computer APL workspaces can have variable size according to the actual size of the available memory in the machine. The addressing capability of the 8088 microprocessor is 20 bits, but only 64 K bytes are directly accessible at a given time. The use of the elastic workspace concept has been applied to overcome this drawback thus providing access to workspaces over half a megabyte of size.

The APL system has been designed in such a way that the APL interpreter, the auxiliary processors and the interface between the former and the latter (the shared variable processor), are completely separated. This feature presents two advantages. First, the auxiliary processors can be individually loaded, and second, the users can design new auxiliary processors to answer needs that may not be currently supported by the system.

At load time APL can be invoked with a list of auxiliary processors to be used during the session. The shared variable processor is automatically loaded only if at least one auxiliary processor has been requested.

An application can also be started by specifying an APL system command at load time.

A number of auxiliary processors have been provided with the system:

- AP80 supports the IBM Graphics Printer to produce APL characters, and can be used either as a system log to provide a record of the work session, or to selectively print a desired APL object or result.
- AP100 manages DOS and BIOS interrupts.
- AP205 gives full-screen input and output capability, and supports a full-screen, defined function editor.
- AP210 supports DOS file management from APL functions.
- AP232 permits asyncronous communication with VM/370, and supports the upload and download of files.
- AP440 can be used to generate music.

It is possible to work with both graphics and monochrome monitors, (although the APL characters can only be displayed in the former), and switch automatically between them.

The IBM Personal Computer APL works either on the IBM Personal Computer or on the IBM Personal Computer XT, under the DOS 1.1 or 2.0 operating systems, and needs a minimum of 128K of random access memory, one diskette drive, the IBM Personal Computer Math Co-Processor, and the color/graphics monitor.

Optionally, it supports also the IBM 80 CPS Graphics printer, the monochrome monitor, the IBM Personal Computer Expansion Unit, and the IBM Asynchronous Communication Adapter.

CONCLUSIONS

The efficiency of the systems programming approach has been tested and found acceptable. The procedure makes it possible to develop full APL interpreters at a cost of a few person-months, as compared to the several years required to build them from scratch.

REFERENCES

(1) M. Alfonseca, M. Tavera, R. Casajuana, 'An APL Interpreter and System for a Small Computer', IBM Syst. J., 16, 18 (1977).

(2) M. Alfonseca, M. Tavera, 'A Machine Independent APL Interpreter', IBM J. Res. Develop., 22, 413 (1978).

(3) M. Tavera, M. Alfonseca, 'IL. An Intermediate Systems Programming Language', SCR.01.78, Centro de Inv. UAM-IBM, Madrid (1978).