# TRENDS IN PROGRAMMING LANGUAGES STANDARDS

Prepared for X3/SPARC

by

J. A. N. Lee

Chairman, Programming Languages Study Group

August 1984

CS84010-R

## 1. Scope

This overview of standards activities covers programming languages per se, and omits intentionally the areas of addenda functions (such as data base management systems (DBMS) and graphics (GKS)), since these would better be the subject of a separate report. This survey, however, does address issues related to the binding of these functional facilities to the programming languages, and the interfaces between them.

## 2. General Overview

The arena of programming languages standards is in a doldrum, with some standards at the end of their development cycle (APL), others just beginning (C) and some in the maintenance process (FORTRAN). The last big step was the publications of the COBOL proposal which caused so much furor. Although FORTRAN has a new proposal ready for public dissemination, there are no new initiatives that will put a great surge into the visibility of this work. To some extent the majority of new initiatives have occurred within ISO/TC97/SC5 by the establishment last October of a number of new proposed work items which concentrate on the relationships between languages and functionalities and the means for the improvement of language conformance.

The international elements of the programming language standards activities are in turmoil as a result of the reorganization of TC97, which has divided the current work in this area into two new SCs and dropped two secretariats which were originally held by the USA. This turmoil results from two perceptions of this change: (1) That the software community (that is, programming languages and functional elements) have been sacrificed by the reorganizing group in order to gain advantages in other areas which are more profitable currently. (2) That the pleas for improvements in the reorganization plan from the workers in these areas are being ignored by the reorganizers, who are themselves merely professional standards administrators related to the major manufacturers who feel that there is little profit in software standards. Misperceptions and inadequate information on the reasons for the reorganization, therefore, introduce strains into the working relationships of participants in the programming language working group of TC97.

An examination of the current status of programming language processor development in the U.S. today will surely reveal that the majority of major companies are in a state of hiatus. They appear to be waiting for some major decision to be made which will motivate them to restart their development activities. It may well be that this

hiatus is being caused by the programming language Ada and the uncertainly of its place in the marketplace of the next few years. Most major producers have processors for the major languages which will suffice (a part from regular maintenance or porting to new systems) for the next few years; users are not clamoring for new language processors. The development work on Ada processors is being funded by the Department of Defense and it is in the interests of the majority of producers to wait out this development process, and if successful, to fund second versions of the same processors for their proprietary use. It is given that such second versions will be vast improvements over their predecessors.

The area in which there is activity is related to personal computers where there is a paucity of true programming software. It is an area in which small companies can make an impact without significant initial outlay. While the majority of personal systems are supporting BASIC and perhaps FORTRAN or Pascal, the major effort is in the development of applications packages such as VisiCalc, dBASE II, Lotus 1-2-3, Symphony, etc. Moreover, the developers of these systems are not generally involved, nor have the capacity to be involved, in standards development operations.

The current work towards a common Operating System Control and Response Language (OSCRL) has languished both domestically and internationally. It is possible that X3 should seriously consider disbanding this project for the lack of a chairman and with the recognition that the current work on JCL-like requirements is being overtaken by events. UNIX-like or Small talk-like systems are the vogue and may constitute a de facto standard which would supplant any committee developed system.

## 3. International Perceptions

Programming language standardization activity is at a stage where there could be an apparent split between the U.S. and the international communities, and also between U.S. domestic activities and those which remain dependent on international activities. Apart from a few multinational corporations there is little to be gained by working on international standards in an environment which is not conducive to progress. There is a distinct feeling amongst the convenors of the Working Groups of ISO/TC97/SC5 (soon to be SC 21 and 22) that they are regarded as the servants of the other Sc's and that their work is not to take the lead, even in their own area of expertise. The world of standards within TC97 is to be driven by the OSI model. Conformance to the whims of the designers of implementations of that

model will drive other standards activities. This is already being seen in the presentation layer with respect to files and file transfer, and is expected to spread to data representations and procedural interfaces.

In 1980 there was published a book entitled Programming Language Standardization, (Wiley), edited by David Hill and Brian Meek. This book concentrated on programming language standards from a European point of view and showed some lack of understanding of the U.S. processes and procedures. In some respects the authors may have been attempting to change the ANSI methods to better conform to the ISO procedures through this has clearly not happened. In preparation for a new edition of this book, the authors of the various papers and the participants in a written discussion chapter were asked to prepare updates to their previous presentations. Preliminary working drafts of these updates reveal a somewhat better understanding of U.S. ANSI procedures, and a sharper focusing of procedural issues. The forthcoming edition of this book could be helpful in clarifying the differences and the alternative for future coordination.

From a European perspective, American standards developers appear to be isolationist, and at times somewhat "untidy" in the software standards they produce. On the other hand, American standards tend to get developed more rapidly than European and international standards, as a general rule. Underlying the apparent differences in attitude toward standards development is the continuing problem of communication. Much too often the European national standards bodies think that they see only the results of the deliberations of the American programming language standards committees, and therefore cannot fully appreciate the reasoning behind the decisions taken, the technical tradeoffs that were weighted, and the political compromises that shaped the final documents.

The ISO practice in developing programming language standards has been to accept a national standards body, such as ANSI, as the sponsor of a language development effort, and then assure that input from other member countries through their language experts, can get a fair hearing in the determinations of the sponsoring nation's standards committee. This practice has been followed for COBOL, FORTRAN, ALGOL, PS/I, CLPT (Text Processing), BASIC and ADA. Some European observers have alleged that this procedure amounts to reducing ISO standardization for programming languages to a simple reference to corresponding National Standards produced by the sponsoring nation. The other member bodies appear to be in a 'take it or leave it' situation under this procedure. The reality, however, is different. The national standards committees that sponsor

the development of a programming language for ISO adoption have been careful to schedule meetings of their technical committees abroad, and to open up the communications with other national member bodies so that inputs and comments can be made early in the deliberations, and can be based on a fuller understanding of the technical tradeoffs and other considerations of implementability, etc.

Another source of tension arising from this practice is the misperception that the sponsoring nation's standards committee may be trying to maintain or create a commercially advantageous postion for that nation's computer industry in international markets. The suspicion has been voiced that the procedure leaves the way open for the computer industry of one nation, or possibly a single manufacturer or cartel of manufacturers, to gain a significant commercial advantage by controlling the content of a standard. This suspicion, however, is totally contradicted by the safeguards in ANSI procedures, and the fact that a particular manufacturer will have only one vote on a technical committee developing a programming language standard. Moreover, the voting records of the technical committees can readily be examined for any evidence that a group (cartel) of manufacturers may be voting as a block to control the content of a standard. Finally, knowledge of the content of the standard at the various stages of its development cannot be kept a secret because the technical committees work in an open manner.

A specific controversy arose over the proposal that DoD's Ada specification be adopted as an ANSI standard, and then as an ISO standard. To come along with an already developed language definition and ask for it to be adopted as-is appeared to some to be a new procedure that totally by-passed the existing TC97/SC5 procedures and practice. On the other hand, European computer scientists and engineers had much more of a role in shaping the design and content of Ada than they have had in any other programming language. Moreover, there is no evidence that programming language standards developed through the ISO process of formal representation from each of the member countries (on a Working Group) has produced or would produce a qualitatively better standard specification. What appears to be certain, however, is that if the ANSI/DoD Ada standard specification were required to go through a full ISO Working Group process from beginning to end, the availability of an ISO standard for Ada would be delayed for many years without any improvement in the quality of the programming language specification. A more flexible and realistic approach for ISO would be to recognize the work done nationally on the current language definition, and submit the document for public review without procedural delays.

## 4. Technical Issues

### a. Interlanguage Compatability

At both the international and nation levels, there is too little contact and communication between the working groups and technical committees responsible for different programming language standards. Each programming language community has evolved in relative isolation from other language communities. When standards development is initiated on a new programming language, the new working group or committee will generally contain persons whose principal interest and experience has been dedicated to that language. These people generally are unlikely to have had any experience with standardization of another programming language. Usually each new committee starts from scratch, develops a different style of representation of the language specifications, and uses a different meta-language. With the exception of PL/I, which is described in a semi-formal notation, all current programming language standards use natural language, with the resulting ambiguities that are unavoidable when natural language is used. Subsequent efforts to clarify ambiguities tend to result in the stylizing of the natural language and a more extensive set of limiting definitions for the natural language terms. As a result, the natural language descriptions become progressively more 'unnatural,' and the definitions of natural language terms used in one programming language's standard specification tend to diverge from the strict definition of these same terms in other programming languages. This progressive degradation and divergence of definitions for English natural language terms becomes a particularly heavy burden for experts for whom English is a second language.

A cogent argument has been advanced for the use of a uniform formal meta-language for specifying the semantics of programming languages in all ISO (and national) programming language standards documents. This practice, if adopted, would minimize the ambiguities and would allow automated approaches to implementing language specifications. Thus the formal notation for all programming language standards would be the same. TC97/SC5/Working Group 16, on Guidelines for Preparation of Standards for programming languages was established by SC5 at its September 26-30, 1983 meeting in Ottawa to produce "meta-standards" for programming languages. With such guidelines, new programming language standards could be developed in a manner that would result in inter-language comparability.

At the January 30 through February 1, 1984 meeting of Working Group 16, a distinction was made between a formal notation for language constructs, and a full metalanguage

with production rules. It was agreed that in principle a true metalanguage had considerable advantages, and a standard metalanguage for all standards would have further advantages. A simple BNF-style metalanguage combined precision with readability; through was readability just a question of usage and training? Precision was a matter of degree, some syntax commonly being left to non-metalanguage rules; and for complete generality a two-level (van Wijngaarden) grammar was needed. There could be some "sales resistance" to a metalanguage, and in some circles to any metalanguage at all; how could people be convinced that a simple metalanguage was easy to learn and brought benefits of precision with sacrificing readability?

It was recognized that known methods of formal definition of semantics were precise, but at the expense of simplicity and readability. Was this inherent, was it merely a question of educating people, or was it a genuine problem, but one which might yield to further research? It was recognized that, against the disadvantages of ambiguities and contradictions in informal, natural language definitions, had to be weighed the disadvantages of the danger of mistakes occuring through trying to use a difficult and complex notation; both methods were known to be error-prone. However, in principle errors in a formal definition should be detectable by automatic methods, and a formal notation greatly eases translation of the standards into another natural language and reduces the likelihood of introducing discrepancies as a result.

b. Relationship of language standards to other standards

The programming language community has been able to carry on its business for many years without reference to anyone else; they were the masters of both their own fate and those who wanted to interface with them. The tide has clearly turned and unless we can do something to counter a backlash of responsiveness (actually non-responsiveness) from the language communities, we could have a serious problem on our hands. Clearly, independent, stand-alone standards are a thing of the past, but without leadership to coalesce currently disparate activities, the progress of further work could grind to a halt.

The future of programming languages lies not in producing more enhanced standards for existing languages or new standards for new languages; rather programming languages must be inserted into programming environments readily, thus requiring that standards for those environments be developed as a first priority. In the same manner as the open systems community was able to establish a creditability through the development of a reference model which indicated where the components of such a system might lie, so it is important to

develop a model which shows the interrelationships between programming languages, functional systems and software tools, all supported in an environment which permits easy usage.

There is a distinct possibility that the development of standards which pertain to the interfaces (generally called "bindings") between programming languages and functional systems may go the same way as did I/O interface standards -- nowhere! Or possibly the standards which are developed will be specific between languages and functional units so that there does not exist any commonality of interfaces. It would be unfortunate, for example, if the interfaces between (say) FORTRAN and GKS were distinctly different from those between COBOL and DBMS. Moreover, in distributed systems where the communication is achieved through the use of an OSI-based system, it would be unconscionable if the interfaces were different than in a closely coupled, or local, system.

c. Language, System and Environment Reference Models

Once one begins to think about interfaces and bindings between the elements of languages and functions, it is a logical next step to worry about the data which is to be transmitted across these interfaces. The OSI model envisages the existance of a "presentation" layer which interprets between differing representations of data items and thereby resolves this problem automatically. However lacking standards for data items there currently exists no common ground upon which the presentation layer can be built in order to transform two external representations. Work within OSI could force a decision in this area without due consideration for the programming languages or the functional units; this work must be carried on in a coordinated manner.

Work in developing bindings between languages and functional systems should not move on to definitive specifications without first ensuring that the work is in consonance with a "master plan." The hierarchy of tasks which must be undertaken are:

A. Develop a Programming Environments Reference Model

A1. Develop Bindings Descriptions between Programming Languages and

A.1.1 Functional Systems (DBMS, GKS, etc.)
A.1.2 Communications environments (OSI)
A.1.3 Software Tools (debugging systems, application modules)

A2. Examine to contents of Programming Environments to determine where standards are needed for new components, such as

A.2.1 Debugging Systems
A.2.2 Editors
A.2.3 File Systems

B. Develop standards for data items for information interchange

B.1 Numeric Data elements
B.2 String Data, Booleans, simple binary strings
B.3 Structured Data Elements

B.3.1 Regular Arrays
B.3.2 Aggregates/Records
B.3.3 Unbounded data compositions such as Videotext

d. Interlanguage Communication

SC5/WG16 is in the process of developing proposals for New Work Items which would produce "meta-standards" for programming languages which could be retrofitted into existing (and future) standards to uniformly standardize data types (through their proposal only actually deals with interal language data representations) and procedure calling mechanisms.

It was recognized that interlanguage communication was a large area covering several issues. What might be termed the 'dynamic' issues were the province of SC16, being a subset of the problems of open systems interconnection (OSI). 'Static' issues concerned the generation of data items, or files, by one language processor for use by another; and the access from a program written in one language to procedures (or other program segments) written in another. There was of course overlap between 'static' and 'dynamic' issues, e.g., that of data representation. It did include both "simple" types and "aggregate" types. Adoption of a common set of data types would not force all languages to incorporate such types, or for 'untyped' languages to become 'typed.' If, however, a language processor was generating values which were to be accessed by another processor, it could provide a mapping from its internal types onto the common set, while the processor accessing the data could provide a mapping from the common set onto its own internal types. Standardization of mappings for a particular language would be a matter for the standard committees for that language. This problem is so fundamental to interlanguage communication that WG16 should include a recommendation for a new work item to define a common set of data types.

The need for a common procedure calling mechanism was also recognized and it was agreed to include a recommendation for a new work item in this area also in the interim report to SC5. It was agreed to use the term "procedure" to cover both "functions" which returned a value and "subroutines" which did not. The term "module" would be used as a more general term for a subsection of a program capable of separate identification and invocation from elsewhere.

Through the calling mechanism could be separated as a problem from the common data set, both would be needed for an effective means of passing parameters and returning values from function procedures. Again, a processor generating procedures to be called from elsewhere would need to provide a mapping from the common calling mechanism to its internal mechanism, while a processor accessing procedures written in other languages would need to provide a mapping from its internal calling mechanism onto the common mechanism.

e. Secondary Standards in a Hierarchy of Standards

Due consideration should be given to reorganizing standards in this area into a hierarchal system which could achieve two major goals: (1) to remove the multi-purpose nature of the language standards (as they apply to languages, processors and programs) by splitting these uses into separate (but coordinated) standards; and (2) to decrease the complexity of the standards. This might be achieved through the introduction of primary and secondary standards. Primary standards for (say) programming languages would be strict definitions of the language itself, devoid of aspirations to specify conformance, limits and implementation requirements, while secondary standards would prescribe such dependent features as:

- Processor requirements
- Program standards
- Validation procedures
- Subsets and Supersets
- Additional functions (enhancements)
- Interpretations
- Applications Packages

By this method the primary standard could be maintained constant over a longer longer period while the secondary standards could be updated as necessary. Moreover this structure would permit language sponsors to release control to other groups to specify secondary standards (such as other national bodies) while maintaining control over the primary standard. This methodology would be equally applicable to functional systems such as GKS and DBMS. The

concept of secondary standards or some equivalent hierarchial system is worthy of further study.

At the SC5/WG16 meeting in January, the definition of "secondary" standards was discussed. A "secondary" standard was one which did not affect the main language standard (and hence did not affect conformance or otherwise of programs) but placed additional requirements on processors. A secondary standard might require additional functions or features to be provided, not mentioned in the primary standard, or place additional constraints upon the way the processor met provisions of the primary standard - typically by specifying, exactly or more closely, things which the primary standard left processor-dependent. It was important that a secondary standard placed only requirements which did not affect program or processor conformance to the standard. Distinction was made between 'direct' and 'indirect' secondary standards. A direct standard specified requirements upon processors which they had to meet, in addition to conforming to the primary standard; conformance to the secondary standard could be directly measured.

It was agreed that the term 'functionality standard' referred to standards which addressed issues which were independent of the particular language in which the functionality was embedded. The classic example was graphics, were from different languages on the same system users would wish to drive the same devices, which should provide the same functionality to all, and would sensibly do this by all language processors sharing the same device-driver software at a lower level. Advice would be needed from WG17, but one obvious problem was the danger of functionality standards and their binding rules being biassed towards a particular language or kind of language. It was suggested that, for historical reasons, GKS was slanted towards FORTRAN and work towards data base standards slanted towards COBOL.

f. Conformance and Validation

The validation of compilers and interpreters--programming language processors--is becoming more important. The validation of COBOL compilers by the U.S. Federal Government is a well established practice through the services of the General Services Administration. A similar validation service has been set up in the United Kingdom. The validation of Ada compilers and language processors is continuing as an integral part of the Department of Defense's Ada language program. These approaches confirm that the development of tests for the compliance of language processors with language standards can and should be accomplished in parallel with the development of standards. Moreover, the contents of programming language standards

should be such that the development of validation suites will be facilitated.

At the January 1984 meeting of SC5/WG16, it was noted that there was no uniformity in terminology concerning violation of the requirements of standards. There was a recognized division between errors of syntax and errors of semantics but the dividing line varied from language to language, processor to processor, and even context to context for the same language processor. Between languages, a simple assignment (like I:=J) could in one case cause a syntax error (because of incompatible type declarations) or in another case cause a semantic error because, though a type conversion was in general available, it could not be applied to the (dynamic) value of J.

For the same language, an interpretive processor might have to detect dynamically something which a compiler could detect before execution. In the case of division by zero, possibilities for the right operand were

- -a value dependent on input data but which could never be zero
- -one dependent on input data which could be zero but was checked within the program
- -one dependent of input data which could be zero but was unchecked
- -a expression which was identically zero
- -a variable assigned with a zero value immediately before
- -a constant identified with zero declared value
- -a literal zero constant

All but one of these could be checked statically by logical analysis of the program text, without execution, but the task of doing so varied from the straightforward to the highly complex. Furthermore, all depended on semantic knowledge during the statical checking (through, for example, the language syntax could explicitly exclude literal zero from the possible constructs for the right operand of the division operator).

It was agreed to adopt the term "error" for incorrect program constructs which were statically determinable solely from inspection of the program text, without execution, and knowledge of the language syntax. The term 'exception' would be used for all other program faults, i.e., those which in general were detachable only dynamically. The term 'violation' would be reserved for a failure of a processor to meet a requirement of the standard. Various terms like 'undefined' and 'not predictable' tended to be used or interpreted in different ways. The approach to building a permissive standard tended to cause more things to be left

processor-dependent as a simple way of resolving any difficulties, and for the wording even of specific requirements to be looser than need be, because of the prevailing attitude that in general the interpretation of the standard was a matter for the implementor. It even happened sometimes that requirements were deliberately specified in an ambiguous way (with conscious through unacknowledged intent) in order to achieve a notional though not actual consensus.

WG16 hopes that the adoption of this terminology would avoid confusion and ambiguity caused by the use of terms like 'compile time error,' 'run time error' etc. It was agreed that the guidelines would need to cover requirements within standards for the detection, handling and reporting of errors and exceptions, and the distinction between those which were fatal and those which were non-fatal.

The European Computer Manufacturers Association (ECMA) last fall outlined some of the conceptual and practical problems associated with validating compilers and languages processors for conformance with programming language standards. Particular difficulties have arisen in defining conformance, notably in connection with programming language standards. An early example of such a standard would apply to the language, and could consist of definitions of what could legitimately be said in that language and how it was to be expressed. Typically, it would prescribe a set of commands that could be written and made to apply to certain defined classes of variable or operand, what those commands meant and what the proper results of executing the commands should be. In effect, such language definitions postulated a hypothetical machine that could execute directly programs written in the language, though usually such a machine did not exist.

One should consider the precision with which it is possible to determine whether the compiler, together with the hardware on which it is designed to run, precisely emulates the hypothetical machine implied by the standard, command by command. In the strictest sense, this cannot be done by testing. High-level commands are written in terms of general operands described by symbols, whereas any real (emulating) machine operates logically and arithmetically on bit patterns. A correspondence between the transformation of actual bit patterns and the transformation of the ideal operands is difficult to define precisely and unambiguously. For this reason if for no o;ther, it is usually impractical to verify exactly the operations of individual commands. Moreover, unless the emulation is by an interpretive process one command at a time, the sequence of machine level instructions generated by a compiler for each high level command will depend on the sequence of other high level

commands in which it is embedded. It is thus possible in practice to verify only the results of several commands taken as a sequence on selected ranges of operands. The combinations of sets of machine level sequences of instructions and of values of operands are limitless, and only a small fraction of all possible command sequences can be explored in the testing. Experience has shown that extensive testing is necessary if troublesome misinterpretations of a language standards are to be detected.

i) the drafting of the standard itself may not be clear or complete, (e.g., the test limits have not been defined), so that its very interpretation is contentious,

ii) no satisfactory method of testing conformance was (or could have been) laid down so that technical disputes arise in interpreting test results,

iii) the product in question fails to embody every aspect of the standard, even though it conforms in respect of those aspects it does embody,

iv) the product does conform in every defined respect but it incorporates additional features of a similar kind to those specified and which could, for example, affect interchanges of data or programs, interworking between equipments or interchange of units of equipment,

v) the standard left options to the implementor that mean that a desired form of interworking can be frustrated.

The first two of these reasons imply some weakness in standards that could possible have been eliminated in the drafting process and that in future are more likely to be avoided now that stress is being placed on the importance of doing so. However, as has been explained in connection with programming languages, comprehensive testing for conformance may be strictly impossible, so that while the position can certainly be improved, absolute conformance may not be a valid concept in every instance.

The last three reasons however relate to decision by the designer or supplier of a product intended to conform to, or to interwork with, other products in accordance with some standardized protocols or procedures. The level of standardization achieved may be entirely adequate and acceptable in some contexts, but not others. It is also clear that both the force and the value of statements of conformance depend on the extent to which the standard was

drafted with a clear intention of making conforming products identical, interchangeable or interworkable. Very often the variation in the drafters' intentions here has been enormous. In the case of programming language standards, the drafters were clearly unable to meet the objective of completely standard entities (programs or compilers) that would interwork with no difficulty at all.

One can envisage high-level test programs or a suite of test programs so devised that all features of a language are systematically exercised. If these test programs when submitted to a compiler produced a machine level program that ran and produced the intended results, one could infer that the compiler was (in general) capable of compiling source code written in accordance with the language standard.

For compilers the term "validation" is used increasingly to mean checking to see that a compiler will properly handle programs written in a given high-level language. The issue is not one of testing for absolute conformance since that is not achieveable. Obviously, a validation process will not often given an unequivocal result, more a quality rating. The only unequivocal result would be an abject failure.

## 5. Future of Programming Language Standards

Some observers are stating that conventional languages of the sort that have been standardized up to the present are obsolescent. The languages that are superseding these procedure-oriented languages are the expert, knowledge-based systems that have been developed in the artificial intelligence community, and the applications generators, problem specification languages, data base query languages and the whole class of facilities called "fourth generation" languages. Moreover, in the future programmers will not program computers; end users will simply communicate with them.

It may be a bit premature to hold a funeral for programming language standards, and especially for the discipline of language standardization. It is not too difficult to visualize the new horizon of language usage in communicating with computers that will encounter its own characteristic standards problems. For example, if end users, communicating in natural language with computers, find that different computers respond somewhat differently to the same natural language inquiry, then the need for standards is obvious. The field of Computational Linguistics has made much progress in defining the types of issues that need to be addressed in standardizing specialized sub-languages. In addition, commonality in the meaning of icons and symbols used in communicating with

computers will also be a standards issue. The CODASYL COSCL Committee is already looking into this issue.

At the January 1984 SC5/WG16 meeting in London several other areas for new standards activities were identified. The meeting drew up a list of possible areas for functionality standards, for future consideration as possible work items. Those identified were:

- real time facilities
- sound output
- sound input
- visual input and image processing
- screen management
- telecommunications
- screen management
- telecommunications
- real arithmetic
- file handling

Through some of these (like real time) might be regarded as being well served at the moment by special-purpose languages or functionality within general-purpose languages, the needs of future languages should not be forgotten; nor the advantages of developing commonality of facilities as existing standards were revised.

In addition, standards for programming environments are just beginning to be recognized as generally important. The Ada program of the Department of Defense is working on developing a standard programming support environment (APSE) and EWICS TC 2 is working on a programming support environment for real-time BASIC. Work is needed to identify the best form standards for programming environments should take, their component software tools and utilities, and how such a project can be validated. At the London SC5/WG16 meeting in January, it was recognized that the category of functionality standards could be extended to include processor functionality as well as program (i.e., language) functionality. Possible areas for future consideration were identified as:

- debug/trace facilities
- help systems
- program analyzers (e.g., profilers and cross-reference lists)
- editors and pretty-printers

and indeed the whole program support environment area.