SETL Newsletter 142 January 29, 1975

What Programmers Should Know

J. T. Schwartz

In this short sermon, I will summarize some of the views concerning the programming process to which I have been led by my SETL project involvement, formulating these views as recommendations concerning the intellectual equipment and cast of mind which a creative, high level programmer should attempt to acquire. I have in mind here programmers (or designers) who originate programs, rather than programmers (alas! the vastly more numerous group) whose work is the extension and repair of programs poorly done and documented in the first place, and the adaptation of these programs to shifting system interfaces. And I will stress the 'higher' rather than the commonplace aspects of the programmers' intellectual armament.

A programmer should understand:

1. *Algorithms*, i.e. various important algorithmic inventions using which significant processes can be performed with special efficiency. Examples are heapsort, fast Fourier transform, parsing techniques, fast polynomial factorization methods, etc. He should understand that a formal concept of program performance exists, and have some familiarity with the combinatorial techniques used to analyze algorithm performance. In this connection, it is also important to understand that there exist processes which no program can carry out rapidly, and others which no program can carry out at all.

2. *Semantic frameworks*, which allow individual algorithms to be organized into large program structures. He should understand the use and significance of such fundamental semantic inventions as subroutine linkages, space allocation, garbage collection, recursion, coroutines, and various structures useful for organizing processes acting in parallel. He should be familiar with object/operator algebras which are of general significance or which play an important role in significant application areas: sets and mappings, strings

and patterns, Curry combinator and lambda calculus, etc. He should understand the way in which semantically significant languages make these frameworks and algebras available, and the way in which the syntactic features of a language facilitate the use of its underlying semantic capabilities.

3. The programmer should have a conscious view of the *programming process*, understand the way in which programs, in their earliest origins, coalesce out of less organized intellectual structures, and understand the objective/psychological influences which can either facilitate the development of a final, efficient and reliable program version or abort this development. Accumulating complexity should be understood as a central peril to successful program construction, and techniques for managing and minimizing this accumulation should be appreciated. Particularly important among these techniques are the orderly multilevel development of more and more efficient program versions through a sequence of progressively less high language levels, and also prespecification, for each major application, of a well-tailored set of application-specific primitives, expressed as macros, structure declarations, or auxiliary subroutine definitions. Simple clean logical structure should be perceived as a central goal of programming; and each simplification seen as a victory, each complication as a defeat. The programmer should learn to structure his programs in spare, logically clean ways which keep open the possibility of subsequent functional expansion.

4. The step which leads from a high-level program representation to a lower level and more efficient version of the same program should be seen and approached as a process of *manual optimization* to be carried out in a mechanical spirit. For use in this process, the programmer should have knowledge of a wide variety of optimization approaches and optimizing transformations, adapted to the various language levels at which optimization will be directed, and ranging from high level global program restructurings to machine level inner-loop bit-tricks.

5. The manner in which the global properties of an algorithm determine the *data structures* appropriate for the représentation of the objects which it manipulates should be understood. The programmer should have a wide variety of data structures at his disposal, and understand the efficiency with which these structures can represent more abstract data objects and operations.

6. The fact that very small *inner loops* are often critical for program efficiency, and that conversely most of a program lies outside its efficiency critical paths, should be understood, which implies that it is important to measure actual program behavior before committing to the optimization of any particular section of code. (Note that the optimization of large noncritical program sections represents an unwarranted expenditure of program resource.) He should be familiar with the tools for measuring program behavior, which various languages, operating systems, preprocessors, and program editors provide.

7. The programmer should understand the techniques which can be used to adapt programs to run well in specific *operating environments*; this implies knowledge of data staging, overlay, paging, and virtual memory techniques. The principal factors which affect program performance in these environments should be understood, as should the way in which programs can be structured to isolate environment dependencies and preserve inter-environment portability.

8. The *correctness of a program* rests on a web of logical relations, implicit in and guiding the program's development; this set of relationships, if made manifest and formally complete, would constitute a formal proof of the program's correctness. An essential part of program development is to guard the integrity of this web as successively more specific program versions are developed, to structure programs so that

the logical assumptions on which it rests do not become unmanageably complex, and to check the logical integrity of the program systematically and repeatedly as it is developed. The fact that some programming language constructs aid in the preservation of logical integrity, while other more dangerous tools tend to tear a program's underlying web, should be appreciated.

The process of debugging is that of searching, in the possibly very large execution-event space of an ill-behaved program, for *primary anomalies*, i.e., places at which good input leads immediately to bad output; these are the events which point to program errors. The debugging tools which make it possible for this large space to be searched should be mastered; bugs should be recognized as inevitable and programs prepared in ways which facilitate their detection and removal; but debugging should be seen as a process for repair of a relatively small number of tears in an extended and delicate logical fabric, rather than a process which can bring order into a heap of disconnected strands. During program debugging, the programmer should always understand the degree to which the tests which he has administered 'cover' all the possible lurking-places of bugs, and should design tests systematically for maximum coverage. The types of program constructs likely to give rise to bugs, and the types of bugs typically to be expected, should be understood, and the kinds of static and dynamic consistency-checking likely to uncover bugs rapidly understood also.

Finally, the several techniques of formal program-correctness proof should be known, and the implications of these techniques for the construction of relatively bug-free programs and for bug detection comprehended.

9. Finally, we list various important *hand-skills and habits* of an elementary but important sort which the programmer should have. He should know the interactive, editing, and program maintenance aids available to him; program carefully, check conscienticusly, and document scrupulously, always remaining

aware of himself as a team member whose expensive product must reliably serve others. He must realize that programming is a highly unstable process, in which a disorganized effort can consume ten times, or even a hundred times, more resource than a well-devised effort with the same goal, and that especially in programming, work is a *signed* quantity, and mere activity, no matter how energetic, no proof of significant contribution to a goal.