A PROPOSED GRADUATE COURSE in AUTOMATIC SOFTWARE GENERATION

by
Wendell L. Pope
Utah State University, Logan, Utah

Abstract

The backlog of demand for applications software is exceeding the productivity of programmers using conventional procedural languages. One approach to solving this problem is increasing programmer productivity through the use of automatic software generation systems. This paper presents the background of the software crunch, the reasons conventional programming languages are not expected to meet it, and the possibility of the state of the art being adequate to support a graduate course in automatic software generation.

1. Introduction.

Pressures are increasing to develop more software more rapidly and to give users the ability to generate some of their own applications. The pressures come from the increasing number of computers being installed with the resulting demand for software to make them useful, the limited supply of traditionally trained programmers, the limited capability of educational institutions to train programmers, and from the decreasing cost of computers vis-a-vis the cost of programmers.

"It is estimated that the number of computers used for scientific and commercial applications will continue to grow at 25% per year at least. It is growing faster than that now." [Martin82,p1]. An IBM survey found that "the number of applications in today's data processing centers is growing at 45% per year."[Martin82,p2]. Assuming no change in the ratio of programmers to computers, and no increase in productivity, the number of programmers in ten years time would have to increase over nine times, and the backforty-one times! "Any set of estimates of computing power 10 years hence indicates that the productivity of application development needs to increase by two orders of magnitude during the next 10 years."[Martin82,p2]

Of course the means to increase productivity is being developed, but the results are slow compared to the demand. Structured programming techniques with existing procedural languages have produced productivity gains on the order of 10%[Martin82,p2]. Of more serious concern is the slowness with which improvements move into the world of work and the inability of existing educational institutions to train a sufficient quantity of new programmers. Existing faculties are at full load and beyond, and the number of new faculty members is small compared to the demand. The Snowbird Report of July, 1980 (so-called because the conference that spawned it was held at Snowbird, Utah) states that "there is a severe manpower shortage in the computing field. It is most acute at the Ph.D. level: the supply of new Ph.D.'s is about 20 percent of the demand." [Denning81] Some data indicating the inability of educational institutions to keep up with demand are:

> The total number of Ph.D. Computer Science faculty in the United States increased from 805 in 1975 to 837 in 1979. The net gain, 32, is 2.8 percent of the total of 1130 Ph.D.'s graduated in the same period. Most faculty outflux is into industry, not retirement.
>
> The total number of Ph.D. graduates, the next generation of researchers and teachers, has decreased from 256 in 1975 to 248 in 1979.
>
> In 1979, there were 1300 jobs advertised for these 248 Ph.D.s.
>
> In 1979, fewer than 100 of these Ph.Ds chose academic careers, and they had over 650 academic positions from which to choose.
>
> Undergraduate enrollments have doubled since 1975, while faculty size and lab space have remained nearly fixed.
>
> Class sizes are significantly higher than in other science departments.[Denning82,p370]

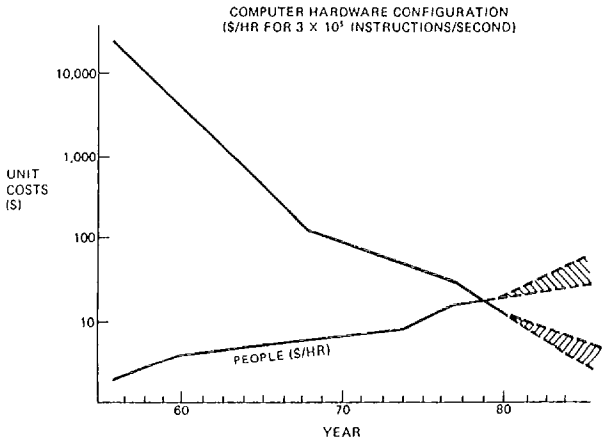
It is well known that software costs now exceed hardware costs.

> The 1973 data processing costs in the U. S. Department of Defense have been estimated in an IDA study as follows:

Hardware	$1.0-1.4 Billion
Software	$2.9-3.6 Billion
Operations	$2.3-3.3 Billion

That is, software above is 3 times the cost of hardware, while software and operations (which are affected by software quality) is more than 5 times the cost of hardware. These figures are typical of industry as well. [Mills80,p798].

As shown in the graph below, on the basis of the cost for 300,000 instructions per second, the per hour cost of personnel exceeded the per hour cost of computers for the first time in 1979 (note that the vertical scale is logarithmic). [Martin82,p3]



2. Language levels.

Intuitively we feel that the time and effort in writing programs increases with their size and complexity, and decreases if we can increase the level of abstraction. These feelings have been confirmed by work done by Maurice Halstead, and he has developed metrics for measuring these factors in programs. See [Halstead79] and [Fitzsimmons78]. He measures the level of languages and finds that most programming languages are around 1.0, assembler languages lower and problem oriented languages higher, for example, PL/I has a level of 1.53. His measure of level for English prose is 2.16.

To illustrate a consequence of this, let L be the level of abstraction of a given algorithm, let V be the volume of the algorithm as written in a given language, let V^* be the potential volume (i.e., the volume in the highest-level representation of the algorithm, which is a constant), let E be the effort to write the algorithm in a given language, and let λ be the programming language level. Halstead defines $E = V/L$ and $\lambda = LV^*$. It can be shown that $E = (V^*)^3/\lambda^2$. The effort in writing a program in PL/I, then in English goes from $E_{PL/I} = (V^*)^3/(1.53)^2$ to $E_{Engl} = (V^*)^3/(2.16)^2$. Further, we have:

$$\frac{E_{Engl}}{E_{PL/I}} = \frac{(V^*)^3}{(2.16)^2}\,\frac{(1.53)^2}{(V^*)^3} = \frac{(1.53)^2}{(2.16)^2} = .5$$

We reason that if a more powerful general purpose language than PL/I is invented, its level would be between 1.53 and 2.16, and the effort in programming could be reduced by at most one-half. We conclude that attempts to gain improvements in productivity through better general purpose programming languages will not produce improvements in the range of an order of magnitude or better. In Halstead's words:

> Similarly, in computer languages, we must expect to reduce the range of applicability if we would raise the level. Consequently, we should expect that the most fruitful direction available is in the design of preprocessors or special purpose languages [Halstead79,p136].

Productivity gains of an order of magnitude or better are claimed for some modern commercial packages classified as fourth-generation languages [Read82].

3. The proposed course.

To give computer professionals the skills to create and use such highly productive, advanced tools a graduate level course is proposed. The course would draw upon state-of-the-art capability in finite automata, formal grammars, data abstraction and automatic software generation. In broad outline, the course would prepare students to create special purpose languages using an automatic lexical analyzer generator and an automatic parser generator to relieve most of the burden of labor associated with language creation. Students would be encouraged to use, as much as possible, the language of the workplace with operations on objects familiar to the workplace. Skills in data abstraction would then be addressed so that meaningful data objects and operations on them can be developed. Finally, current work in automatic software generation systems would be studied so as to integrate the preceding skills into a capability to produce a working system. Along the way, issues of automatic generation of documentation, system reliability and efficiency should be addressed. The following sections discuss these topics.

4. Special purpose languages.

It has for some time been possible to automate the implementation of a lexical analyzer and a parser for a language. The tokens (terms) used in a language can be represented by regular expressions and a lexical analyzer automatically generated to recognize them [Aho77,Lesk75]. The tokens recognized by the analyzer can then be passed to a parser, so a statement can

be recognized. To generate the parser it is necessary that the allowable statements in the language be specified by a context-free grammar. If the context-free grammar is presented in Backus-Naur Form (BNF), then the parser can be generated automatically [Johnson74,Wetherell81]. With this kind of capability, it should be possible to construct, in the language of the workplace, the special purpose language useful to implement applications in that workplace. Such a language would have terms familiar to the workers in the workplace, and would perform operations on objects familiar to them. That is, workers could think in terms of 'running the payroll' or 'hiring' an employee rather than inserting a record into a database. [Ledgard70,p384]

4.1 Very high level languages.

As noted in 2. above, languages increase in level as their level of abstraction is raised. The abstraction of a computing language increases as the objects and operations it deals with move from machine level toward objects in the real world. Further characterization of very high level languages are that they are non-procedural, have aggregate operators and allow associative referencing [Schwartz80,p578]. (Schwartz' section on very high level languages is quite short and ought to be required reading in the course.) Prywes achieves non-proceduraliity in his MODEL system by allowing statements to be entered in any order, even when an order of processing is known. As the statements are entered, the system generates a directed graph whose edges are precedence relations between statements. Once the graph is completed, and passes the system's completeness test, then a topological ordering of the graph produces an ordering of the statements that satisfies the precedence relations amongst the statements [Prywes77,p102].

4.2 Teaching materials.

To do work in this area, students will need to be acquainted with formal languages at Chomsky's levels 3 (regular expressions) and 2 (context-free). A good background is given in Chapter 4 of "The Handbook for Artificial Intelligence" [Barr81,p233] and in Chapter 4 of "What Can Be Automated" [Karp80,p216].

If an automatic lexical analyzer generator or an automatic parser generator must be written, then information necessary can be found in Chapters 3, 4, 5 and 6 of "Principles of Compiler Design" [Aho77].

5. Data objects and operations on them.

To achieve the goal of operating on objects familiar to the workplace in the language of the workplace, we must have a means of defining the objects and the operations on them in addition to the language design capability discussed in 4. above. The most promising direction for such work seems to be that proposed by Parnas[Parnas72], which has come to be known as data abstraction. Parnas' ideas of encapsulating a data object and the operations allowed on it have been implemented in the Ada package construct[LeBlanc82]. An example of an implementation in Fortran is given by Isner[Isner82]. Fortran does not lend itself to the notion of encapsulating data objects and operations as well as Ada does, but it is at least possible, and with disciplined use, it can be made to work.

Considerable programming skill is necessary to do a good job and achieve the desired results at this stage. The programming language used should probably be the target language used as output in the parsing stage in 4. above.

6. Generation of documentation.

The generation of documentation begins with the analysis of the problem and the design of the special purpose language to be used. As terms (tokens) are collected for use in the language, a dictionary of meanings should be developed and stored in machine readable form so as to be available for inclusion in any documentation of the system. The list of tokens serves as input to the lexical analyzer generator. Output of the lexical analyzer generator should include a clean copy of the list. Next, the Backus-Naur Form (BNF) of the context-free grammar must be developed for input to the parser generator. Output from the parser generator should include a clean copy once the ambiguities and inconsistencies have been removed from the grammar.

The next stage is the development and coding of the data objects and the operations on them. This is a traditional programming project, and documentation should be prepared with rather greater care than is traditional because of the special nature of the objects being created. The documentation should include the design specifications (implementation independent), a description of the implementation chosen, and a listing of the code generated for each data object.

Provision should be made to capture some documentation as the system is running to generate an application. The source statements can be captured and listed together with the target software produced. Provision should also be made for comments to be included with the source statements. These would pass through and form part of the documentation of the generated software.

7. Reliability considerations.

Some measure of reliability should derive from the way software is produced from an automatic softwware generation system. That is, the parser generator assures us that we start with a consistent grammar, and the software generator, by its very nature, produces code in a standard way. Thus, desirable programming

standards can be built into the generator leading to the opportunity to produce consistent, standard code that is more reliable and easier to maintain. This result does not come automatically and care should be exercised at each stage of the development and generation process to plan and provide for reliability.

It would be useful to build into the software generator some of the results of work being done on program correctness proofs and reasoning about programs as mathematical objects. In this regard we note that the COSERS report espouses the axiomatic approach of Hoare[Mills80,p809]. O'Donnell thinks the axiomatic approach is weak. He cites some rules of the Hoare logic that are unsound and observes that "convenient and elegant rules for reasoning about certain programming constructs will probably require a more flexible notation than Hoare's [O'Donnell82,p927]. Culik and Rizki argue that mathematical proofs are more appropriate than logic proofs for computer science education [Culik83]. Backus proposes that an entirely new approach to language development be taken. His functional programming style proposal is based on an algebra of programs that makes it possible to combine parts of programs into demonstrably reliable larger units [Backus78]. Williams has developed a realization of some facets of Backus' proposal [Williams82]. It would be prudent to stay in touch with this work and incorporate results as they become available.

8. Efficiency considerations.

It has been justly observed that automatically generated software tends to be inefficient. The sutuation is reminiscent of the days when higher-level languages were being introduced and having to overcome the resistance of assembly language programmers who knew how to get the best performance out of their hardware. That experience suggests that the target language of the software generator ought to be a higher-level language with a good optimizing compiler. One also ought to plan for a feedback loop that allows inefficiencies to be removed from generated software in a controlled manner with documentation of what has been done. The decreasing cost and increasing speed and capability of hardware is on the side of reducing the impact of the efficiency problem, but cannot be relied on as the entire answer.

Bibliography

Aho, Alfred V. and Ullman, Jeffrey D. "Principles of Compiler Design" Addison-Wesley, 1977.

Backus, John, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs", (1977 ACM Turing Award Lecture) Comm. ACM 21, 8 (August 1978) pp613-641.

Barr, Avron and Feigenbaum, Edward A. "The Handbook of Artificial Intelligence", Wm. Kaufman, 1981. Chapter 4 is "Understanding Natural Language"

Culik, K. and Rizki, M. M. "Logic versus Mathematics in Computer Science Education", ACM SIGCSE Bulletin, 15, 1 (February 1983) pp14-20.

Denning, Peter J.; Feigenbaum, Edward; Gilmore, Paul; Hearn, Anthony; Ritchie, Robert W. and Traub, Joseph "A Discipline in Crisis", Comm. ACM 24, 6 (June 1981) pp370-374.

Fitzsimmons, Ann and Love, Tom "A Review and Evaluation of Software Science", ACM Computing Surveys 10, 1 (March 1976), pp3-18.

Halstead, Maurice H. "Advances in Software Science", Chapter 4 in "Advances in Computers", Vol 18, M. Yovits, Ed., Academic Press 1979, pp119-172.

Isner, John F., "A Fortran Programming Methodology Based on Data Abstraction", Comm. ACM 25, 10 (October 1982), 686-697.

Johnson, Stephen C. "YACC: Yet Another Compiler-Compiler", Computing Science Technical Report #32, Bell Laboratories, Murray Hill, N. J., July 31, 1978.

Karp, Richard M., panel chairman for "Theory of Computation", Chapter 4 in "What Can Be Automated?", B. W. Arden Ed., MIT Press, 1980. The section we refer to is "Language and Automata Theory", pp216-235.

LeBlanc, Richard J. and Goda, John J., "Ada and Software Development Support: A New Concept in Language Design", IEEE Computer, May 1982, 75-81.

Ledgard, Henry F. and Taylor, Robert W., "Two Views of Data Abstraction" Comm. ACM 20, 6 (June 1977), 382-384.

Lesk, M. E. "LEX: A Lexical Analyzer Generator", Computing Science Technical Report #39, Bell Laboratories, Murray Hill, N. J., October 1975.

Martin, James "Applications Development Without Programmers", Prentice-Hall, 1982.

Mills, Harlan D., panel chairman for "Software Methodology"; Chapter 11 in "What Can Be Automated?", B. W. Arden Ed., MIT Press 1980, pp791-820.

O'Donnell, Michael J. "A Critique of the Foundations of Hoare Style Programming", Comm. ACM 25, 12 (December 1982) pp927-935.

Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules", Comm. ACM 15, 12 (December 1972) pp1053-1058.

Prywes, Noah S., "Automatic Program Generation"; Chapter 2 in "Advances in Computers", M. Rubinoff and M. C. Yovits, Eds., Vol. 16, Academic Press, 1977, pp57-125.

Read, Nigel S. and Harmon, Douglas L. "Language Barrier to Productivity", Datamation, February 1982, pp209-212.

Schwartz, Jack, panel chairman for "Programming Languages", Chapter 8 in "What Can Be Automated", B. W. Arden, Ed., MIT Press, 1980. The section we refer to is "Very High Level Languages", pp577-583.

Wetherell, Charles and Shannon, Alfred, "LR-Automatic Parser Generator and LR(1) Parser, IEEE Transactions on Software Engineering, SE-7, 3 (May 1981), pp274-278.

Williams, John H. "Notes on the FP Style of Functional Programming" in "Functional Programming and Its Applications: An Advanced Course", J. Darlington, P. Henderson and D. A. Turner Eds., Cambridge University Press, NY, 1982, pp73-101.

****

MANAGING PROGRAMMING ASSIGNMENTS - continued from page 28

has remarkably efficient string-handling machine code (programs can be compiled, not just interpreted). So our final version is a fairly simple one where the range string is progressively munched away. If the next piece munched is itself a groupname, its definition range is appended at the leading end of the string instead. Simple, eh Watson?

And each time the Group definition table is referred to in the munching away of a range string, a counter is bumped. Over 32 references mean 'circular definition'. Like all (we hope) other error situations when using #LOG, the error is trapped and dealt with graciously so the user need never re-run the program from scratch (which would lose him all the stored values in main memory).

The range string is simply used to tag a "boolean" array or bit map covering programmer numbers 0 to 214. Any range specification given further along the string overrides any prior specification, which makes for convenient updating of group definitions.

And finally the supporting files. Every instructor has two data files: LIST.DAT with the list of his passwords (in his alloted range, say from 150 to 210) to which he adds student names. When homeworks are printed out, the student name is printed at the top in big letters. Then GROUP.DAT which holds the instructor's semi-permanent group definitions (names with ranges) and which he is free to update at any time.

The Project Manager (me) who allocates blocks of programmer numbers has a couple of special programs. One has "privilege" (even though none of the Walsh instructors has a 1 as first number)--the program was submitted to the System Manager who approved it and has the power to attach a privilege code to a program, once compiled. This program allows the Project Manager to change passwords, or change the disk quota, of any programmer.

Another program finds out what the passwords are, and transfers a block of them to a given instructor's LIST.DAT file. The same program makes a copy of the student names from this LIST.DAT for the benefit of the Group Manager.

Finally, the very first item of virtual array LIST.DAT contains the first and last programmer number of the block allocated to an instructor and the date allocation was made. This is really for information: an instructor could not use programmer numbers outside his allocation because the passwords have not been copied into his LIST.DAT array.

Now you would think that students would object to such a "superspy" but they don't--because no-one takes advantage of it to hurt them in any way. They soon learn that two things must be adhered to: give the right file name to the next program, and have it in by deadline.