REDUCTION OF COMPILATION COSTS THROUGH LANGUAGE CONTRACTION

Mary Shaw
July 17, 1972

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. Support was also contributed by the Univac division of Sperry Rand Corporation. This document has been approved for public release and sale. Its distribution is unlimited.

ABSTRACT

Programming languages tailored to particular groups of users can often be constructed by removing unwanted features from a general purpose language. This paper describes the use of simulation techniques to predict the savings in compilation cost achievable by such an approach. The results suggest a function which describes the effect of changes in the power of a language on the compilation cost of an algorithm expressed in that language: when features not actually used by the algorithm are removed from the language the cost of compiling the algorithm decreases moderately, but when features that are needed are removed, the compilation cost increases sharply.

1. INTRODUCTION

The cost of writing and running a program that implements some given algorithm can be attributed to three sources: compilation cost, execution cost, and the cost of the programmer's time and effort. Any attempt to reduce the overall cost of programming must take all of these into account; any work directed at one component must not simply shift costs to the other two. The work reported here addresses the subproblem of finding ways to reduce compilation costs by eliminating unused features from a language and its compiler. The results also suggest techniques for the analysis and reduction of the other costs. This technique for improving compilation efficiency is based on the premise that a programming language is composed of a number of separate features. The approach involves identifying features of a language that are not useful for some classes of programs being written in that language; these features can then be successively eliminated from the language, leaving useful sublanguages as a result. Algol was chosen as the language for study, and a one-pass production-driven compiler [Eva64] was used as the subject compiler.

2. CONTRACTING A LANGUAGE

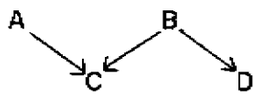
We will rely on the intuitive notion of a feature as a syntactic or semantic component of a language. A contraction is a family of languages produced by successively factoring out groups of features and making any necessary adjustments in the remainder of the language. The individual languages of a contraction must be judged by the usual standards. In addition, the contraction as a whole should be evaluated on the degree to which the member languages are appropriate to existing tasks, the naturalness of the notations used, and the ease of moving from one language to another. To be useful in a real programming situation, a contraction must consist of subset languages appropriate to the tasks being performed. It is often desirable to have a language that is easily taught and provides only the basic tools; the teaching and learning task is substantially simplified when programmers can grow within a single language family rather than outgrowing one language and switching to another, perhaps very different, one. The availability of a variety of strongly related languages can reduce overall costs by eliminating unnecessary complexity and overhead; in such cases it is desirable for the more complex languages to be natural extensions of the language that is taught first. As is usual when a tradeoff between generality and specialization is involved, two costs are associated with any feature. These may be overhead cost of its residing unused in the language and the compiler and the circumlocution cost of using other features of the language to emulate its effect if it is removed. The expected frequency of use of the feature should be a major consideration in the decision about when or whether a feature should be factored out of a language. A compiler is not expected to admit of arbitrary factorization; rather, some few lines of decomposition should set the style for the entire implementation. This will affect the choice of compilation strategy, the degree to which conversational usage is permitted, and other decisions. Once this design decision has been made, some factorizations may be infeasible in that the compiler must be drastically perturbed in order to implement the change. Such factorizations cannot be allowed.

2.1. Formal descriptions of contractions

When several groups of features are factored out of a language, any of them may depend on the presence of another feature or conflict with it. As a result, dependencies and conflicts among the members of a contraction may arise; these lead to partial orderings among the languages. When there are dependencies among all the features to be factored, a linear sequence of subset languages is formed. When the features to be factored are independent, it is not necessary to order them at all. If both dependent and independent features are involved, they impose a partial ordering on the sublanguages of the contraction. Directed graphs are a natural notational device for describing such relations, especially when both dependent and independent features are present. A directed graph describing a contraction will be called a decomposition graph. In the decomposition graph of a contraction,

- each node corresponds to a particular sublanguage,
- each edge connects a sublanguage to one of its immediate descendants (that is, to one which is smaller by a single factorization).

Thus in the decomposition graph



A, B, C, and D are languages; C must be a subset of both A and B; C and D are distinct subsets of B; and D need not be a subset of A.

2.2. Examples

A contraction of some features related to Algol conditional statements is shown in Figure 1. The first factorization eliminates **else** clauses from conditionals. The second restricts the object of the **then** to an unlabelled basic statement, leaving a conditional statement much like Fortran's. The third factorization permits only a **goto** to occur after a **then**, leaving a conditional statement comparable to Basic's. The ellipses before and after the graph indicate that this is a segment of a larger contraction. The example of Figure 1 shows a contraction involving only dependent factorizations. Figure 2 gives one possible factorization of **for** statements; in this contraction some of the factorizations are regarded as independent of others. A factorization of **for** statements might include the following subsets with the relationships shown in Figure 2:

a. full Algol 60 **for** statements
b. control variable restricted to be simple identifier
c. arithmetic expression as **for** list element prohibited
d. **while** clause as **for** list element prohibited
e. **for** list restricted to a single **step-until** element
f. static evaluation of expressions in control element
g. step size always +1; form is now: **for** i := a **until** b **do**
h. initial value always +1; form is now: **for** i **until** b **do**
i. **for** list restricted to the form **while** b **do**

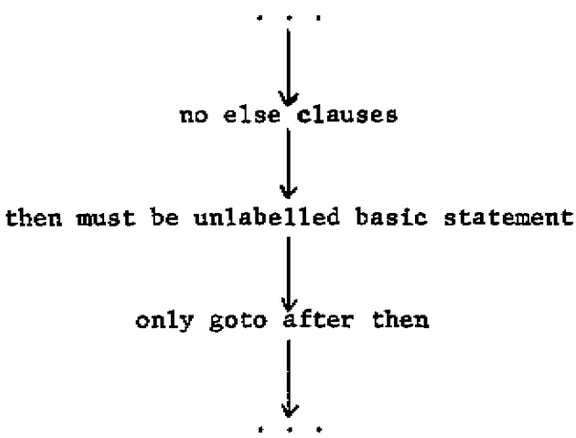


Figure 1. Decomposition graph of one contraction of conditionals

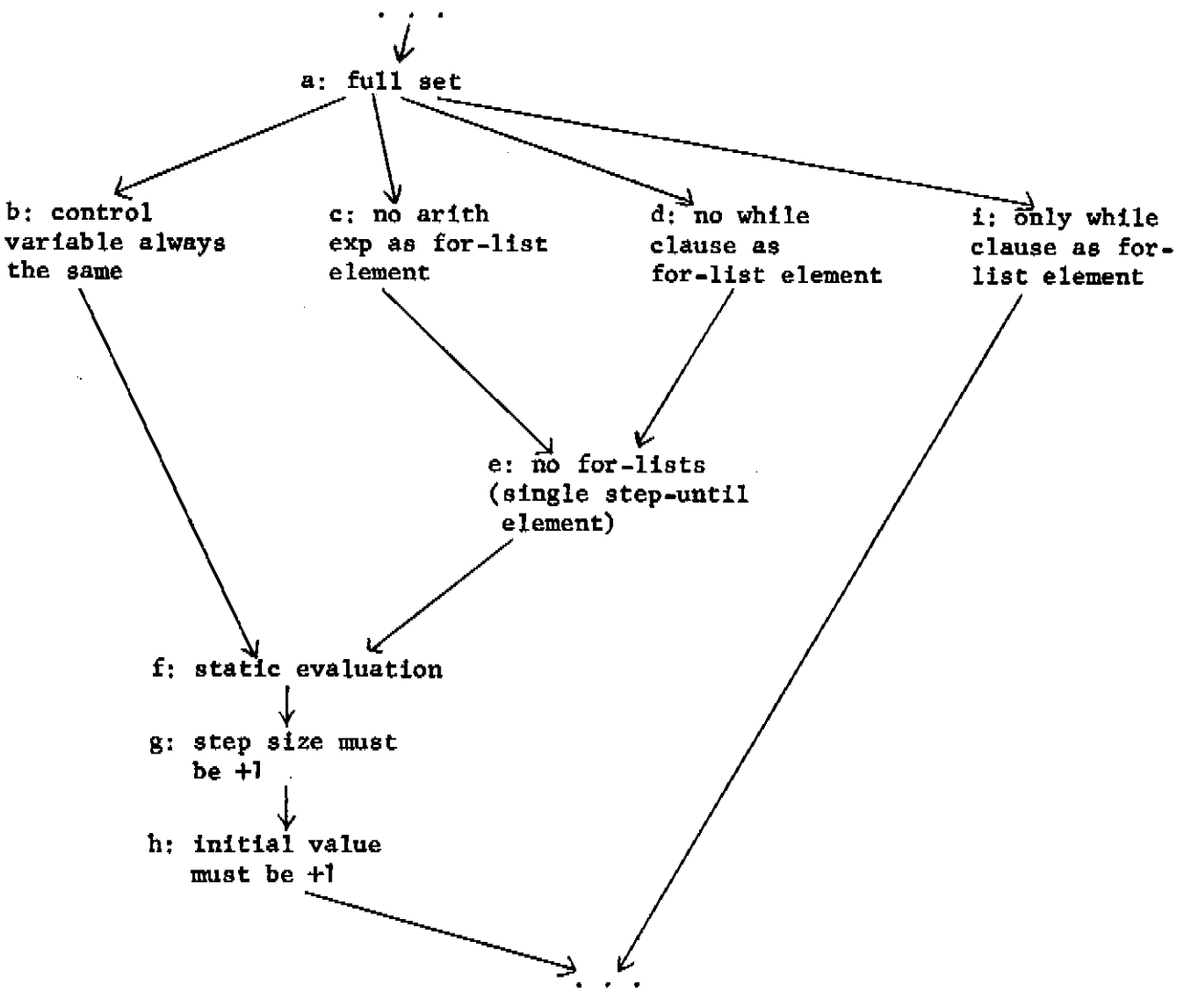


Figure 2. Decomposition graph of one factorization of <u>for</u> statements

2.3. Selection of a contraction for detailed study

A few of the many features that might be factored out of Algol were selected for further study. The resulting family of eleven languages, given in Figure 3, is the contraction studied with the simulation system described in Section 3. The contraction contains two major lines of decomposition. The first involves labels and designational expressions. These are not completely embedded in Algol as data objects, and few programs actually use them in generality. The second and third lines involve procedures. Many features are associated with procedures, and they can be factored out in many ways. The factorizations under consideration touch the most important aspects of the procedure concept: recursion, parameters, call-by-name, and the notion of procedure itself. Three of the languages chosen as members of the contraction correspond to Algol, Basic, and a small teaching subset of Fortran [Ken70]. The languages of the contraction are described below. A three-letter mnemonic is given below for each language. These mnemonics are used for conciseness in the subsequent discussion.

FUL - Algol 60 with the restriction that parameters must be specified and without dynamic **own** arrays, numeric labels, or parameter delimiter comments.

SWI - FUL without switches.

DSG - SWI without designational expressions other than simple labels.

NAM - FUL with the restriction that procedure parameters must be called by value and without array, procedure, and label parameters.

REC - FUL without recursive procedures.

XXX - FUL with a combination of the restrictions of NAM and REC.

PAR - XXX without procedure parameters or typed functions.

BLK - FUL limited by the constraints of both DSG and PAR, plus the elimination of blocks (but not compound statements). As a consequence, **goto** out of block and **own** variables are also eliminated.

FOR - BLK with restriction to **for** statements with simple control variable and a single **step-until** element.

BAS - Algol restricted approximately to the Basic level. Remaining features are integer and real types, **for** statements with a single **step-until** element and simple control variable, conditional statements with the **then** clause restricted to a **goto** and no else clause, labels, single array subscripts for fixed base arrays, multiple assignment statements, parameterless procedures, and arbitrary variable names.

TSF - minimal useful subset of Algol. The remaining features are single assignments, **if**-then-**goto** statements, labels, and array accessing with fixed base arrays and a single subscript. This language is roughly comparable to Kennedy and Solomon's Ten Statement Fortran [Ken70].

Three linear orderings can be extracted from this partially ordered contraction. Each is designated by naming one of the languages that occurs only on that linear subordering. Thus there are the following three lines of decomposition:

a. TSF BAS FOR BLK PAR XXX NAM FUL : call-by-name line
b. TSF BAS FOR BLK PAR XXX REC FUL : recursion line
c. TSF BAS FOR BLK DSG SWI FUL : switch line

Some of the results in the following sections are described in terms of these linear orderings.

3. SIMULATION OF COMPILATION COSTS

One way to evaluate the costs of compiling an algorithm in various members of a contraction would be to write a compiler for all of Algol and subset it for the dialects of interest. However, the effort involved in actually building a family of compilers substantially exceeds the usefulness of the resulting measurements, especially since precise costs are not required. A simulation can produce cost estimates at an appropriate level of detail with a set of programs much less complex than a complete compiler. The trends of these costs from language to language and the shapes of the cost functions are the results of interest. Since the simulation is an abstraction, its results will be much less dependent on particular implementation decisions than the results that would be obtained from a full compiler. It is useful to view compilation cost as a function of language size. The estimates of compilation cost produced by the simulation may be viewed as sample values of such a function. Unfortunately, the notion of "language size" or "language richness" is not well quantified; however, it is reasonable that compilers for "larger" languages tend to be larger than compilers for "smaller" languages. Therefore, compiler size is used here as a crude measure of language size.

3.1. Simulation design and validation

The compiler chosen for contraction is a one-pass production-driven system (see, for example, [Eva64]). In such a compiler, the syntactic analysis is controlled by a program written in a picture language (the productions); this part of the compiler (called Phase I) generates a postfix string representing the program being compiled. The postfix representation is processed by a group of semantic and code-generating routines (Phase II) that output the object program. The objective of the simulation was to produce estimates of compilation times with sufficient accuracy that comparisons among the sublanguages and the algorithms would be valid. The accuracy thus required of the simulation exerted a major influence over its design. On the one hand, the simulation had to carry enough detail to yield results which would discriminate among the various languages of the contraction. In particular, this

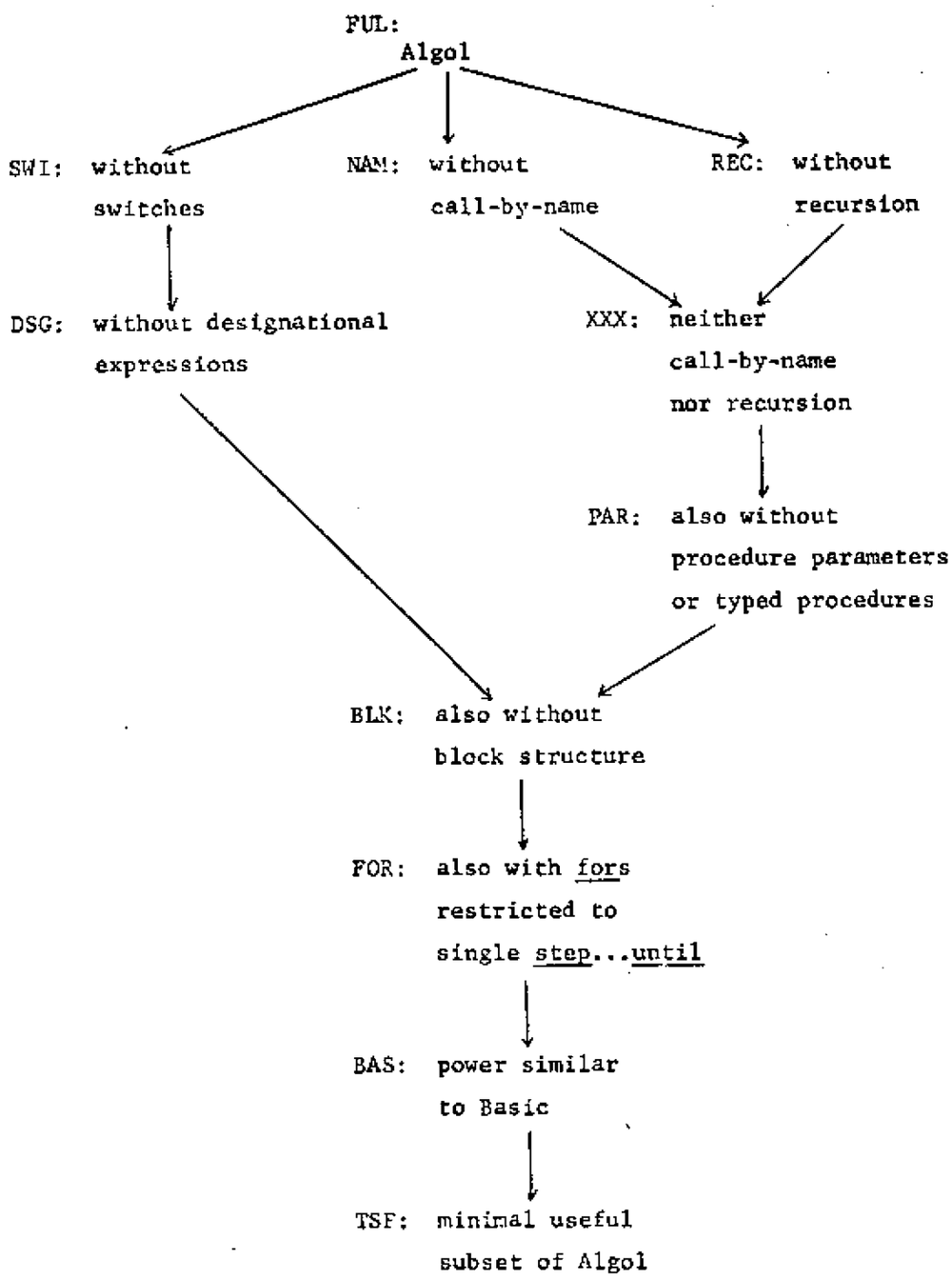


Figure 3. Decomposition Graph for the Algol Contraction

meant that an actual postfix stream had to be produced by Phase I in order to provide sufficiently detailed input for Phase II. On the other hand, a certain amount of abstraction was in order; an experiment to measure the effects of language contraction on compilation cost should not depend heavily on the particular factorizations chosen for the experiment. These considerations led to a system with mixed execution and simulation. An actual Phase I processor performs syntactic analysis and produces the complete postfix representation of the program being compiled along with a record of Phase I costs; this much is not a simulation. The simulation begins with Phase II, which accepts the postfix stream and predicts the costs a compiler would incur in compiling that postfix. Measurements are taken in terms of events that would occur in a real compiler, then converted to a single cost for each compilation at the end of the simulation by taking a weighted sum over the various events. The simulation system itself consists of three programs. The first two correspond to the two phases of the compiler; they collect tallies of events that occur during compilation. The third program converts them to a single estimate of compilation cost by taking weighted sums. A diagram showing the flow of data through the simulation is given in Figure 4. The validation of the simulation included checks on the individual components as shown in Figure 4 and an overall validation. The validation arguments are sketched here; details are given in [Shaw71]. Phase I is a running version of a subsystem that could appear in a real compiler; as such it requires no validation. In Phase II and the analysis phase, the values of the parameters were based on measured values, and the individual contribution of any parameter is small; further, the simulation is not more than one or two levels deep, so errors cannot cascade. Estimates of savings have been conservative throughout the simulation, so errors in the results will tend to underestimate potential benefits. In addition, the simulation results are used only qualitatively, and comparisons are made only among members of the contraction - not with other systems. The overall simulation has been checked with respect to the balance of internal cost distributions and the agreement of ratios between costs of sublanguages with comparable ratios of real compiler costs.

3.2. RUNNING THE SIMULATION

3.2.1. Test algorithms

Six test algorithms were chosen to exercise the features involved in the contraction and fourteen published algorithms [CACM] were chosen to represent the general Algol programming population. The complete simulation results for these programs are given in [Shaw71]; the results for two of the test algorithms (the typical program and the innerproduct routine) are discussed in Section 4. The algorithms were written in full Algol, then rewritten as necessary to adapt to the restrictions of the sublanguages of the contraction. The cost measurements were made on the basis of simulating the compilation of the rewritten algorithms in the languages for which they were rewritten and in the smaller languages for which each rewritten program was still legal. The test algorithms are:

(1) A "typical" program based on the results of several studies of the style of Algol and Fortran programmers [Knu70, Knu.nd, Wich70, Shaw71].

(2) A program to play the children's game "Buzz"; for lists are used extensively.

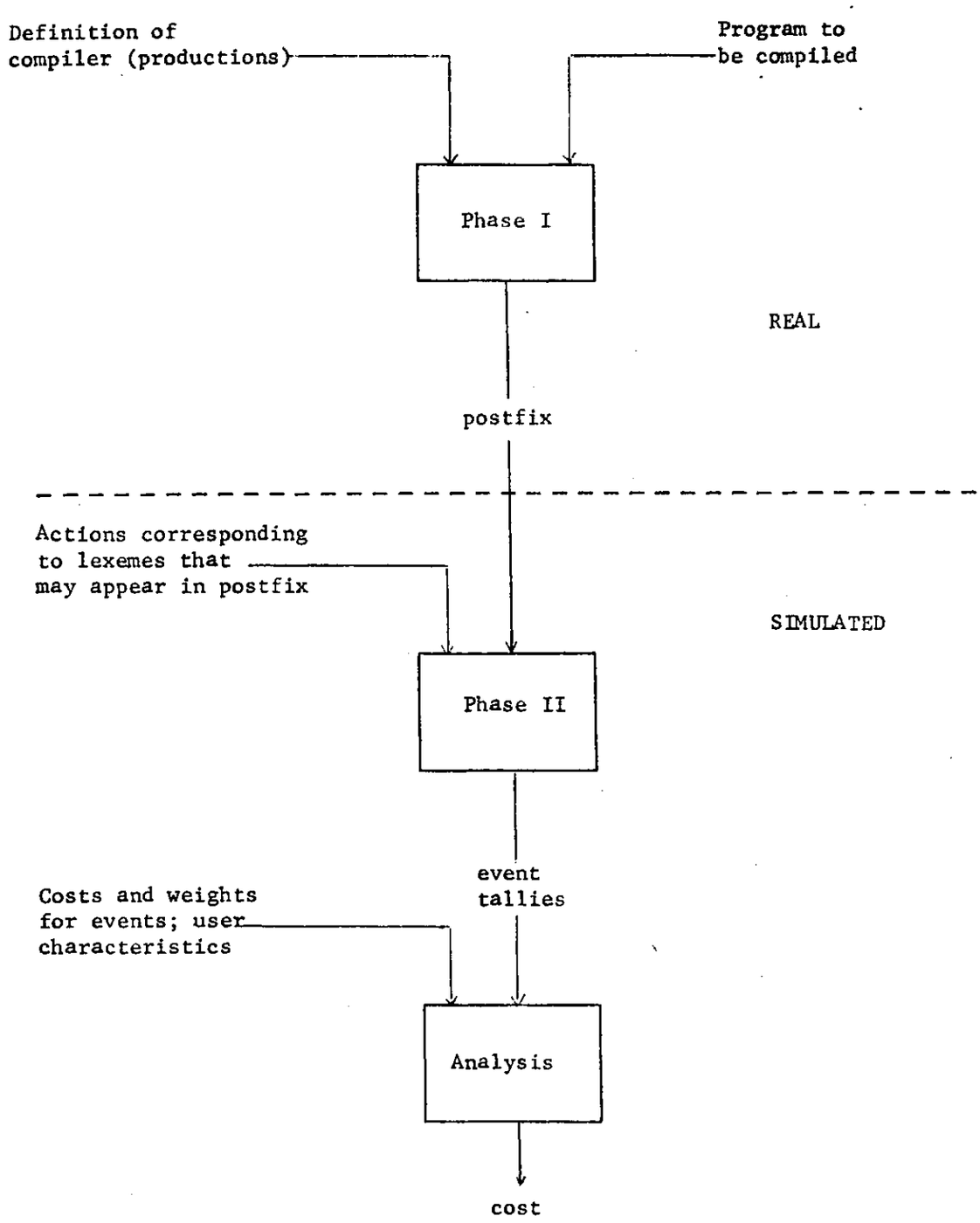


Figure 4. Data Flow of the Simulation

(3) The parsing algorithm for a simple precedence phrase structure language given by Wirth and Weber [Wir66a]; it relies heavily on while clauses.

(4) A simulator for a small computer that uses switches and designational expressions to interpret the operation codes.

(5) An integration routine that adapts to the slope of the integrand by calling itself recursively.

(6) The Innerproduct routine of the Algol report [Naur63] with its use of call-by-name.

The published algorithms were selected to represent a variety of applications. They are:

84	Simpson integration
128	Summation of Fourier series
141	Path matrix
143	Treesort 1
149	Elliptic integral
151	Find vector in sorted list
162	XYmove plotting
169	Newton interpolation
212	Frequency distribution
220	Gauss-Seidel solution to simultaneous linear equations
229	Elementary functions by continued fractions
246	Graycode
294	Uniform random number generator
373	Number of double restricted partitions

3.2.2. Description of a language to be simulated

The subsets of Algol that belong to the contraction sequence are described above. The simulation requires four pieces of data for each language in the contraction:

- the productions and other tables that define the language;
- the set of test algorithms, recoded as necessary to be correct in the sublanguage;
- the cost table that drives Phase II, with one row for each lexeme that Phase I may emit;
- the cost/weight/probability table that drives the final analysis.

Given this information, the simulation system will estimate a compilation time for each of the test algorithms in the given language.

3.3. Estimation of compiler size

Compilation cost depends on the amount of space required to run the compiler as well as on the execution time. Estimated relative sizes for the compilers of the

contraction are given in Table 1. Size estimates for Phase I were based on records made by the production loader when the compiler for each language was simulated. The estimates for Phase II space savings are based on identifying blocks of instructions in an existing Algol compiler that would become unnecessary if certain features were removed from the language. The sizes of a number of tables in the compiler were assumed to depend on the number of special characters in the language. These estimates were combined to obtain overall size change predictions; the complete derivation is given in [Shaw71].

FUL	1.00
SWI	0.96
DSG	0.94
NAM	0.97
REC	0.97
XXX	0.94
PAR	0.82
BLK	0.74
FOR	0.70
BAS	0.59
TSF	0.44

Table 1. Compiler size estimates

3.4. Procedure

The simulation described above was run for each of the eleven languages of the contraction with an appropriate version of each six test algorithms. The raw simulation output represents compilation time only. Real computing costs depend on space as well as on time; the space-time product is often used as a measure of core occupancy. Overall costs are represented most realistically by weighting these measures according to a billing formula. Given such compilation costs for an algorithm in various sublanguages, it is useful to view the costs as a function of language size. Validation of the simulation indicates that the results do not depend heavily on either the billing formula or the scaling of language size (see [Shaw71] for details). The following were chosen for definiteness. The cost for compiling each of the programs representing the six test algorithms was based on the compilation time predicted by the simulation programs, the compiler sizes as given in Table 1, and a billing formula based on the rates

$ .01 per run second	for compute time
$ .002 per 1000 words/sec	for core residence

The resulting costs are scaled so that the cost of compiling each algorithm in full Algol (FUL) is 1.00. As noted above, compiler size is used as a rough indicator of language size. Further, the languages of the contraction under study are only partially ordered, so direct comparisons between some pairs of languages are not meaningful. To avoid that problem, the results in the next section are presented in terms of the three separate lines of decomposition defined in Section 2.

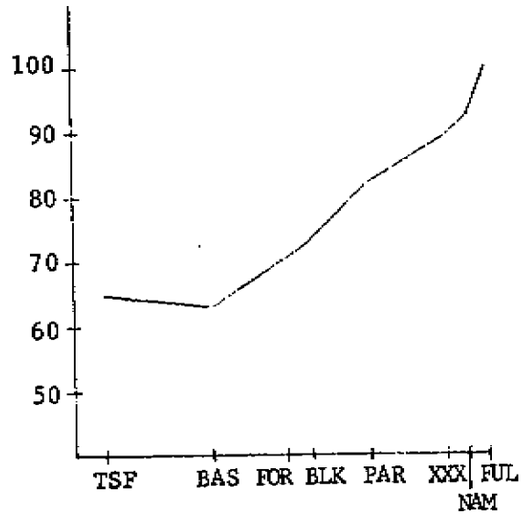
4. RESULTS

Figures 5 to 8 graph compilation cost as a function of language size for representative simulation results. Figure 5 shows the cost function for the "typical" program along the call-by-name line of decomposition; the cost changes along the other lines are much the same. For this program, compilation cost drops evenly as the language is contracted from full Algol (FUL) to the subset comparable to Basic (BAS). The savings arise because this program, constructed to reflect typical usage habits, does not take advantage of much of the richness available in Algol. However, when the final factorization (to TSF) is made, the cost increases. This increase is attributable largely to the loss of the for statement. The cost shifts for compiling the Innerproduct routine along the various lines of contraction are very different. This algorithm relies on dynamic evaluation of parameters to select successive elements of the arrays being multiplied, and removing that facility requires the programmer to write a substantial amount of code to simulate the same effect. Figure 6 gives the cost function for the contraction along the call-by-name line. Call-by-name parameters are removed in the first factorization, and the cost function increases sharply as the language moves from FUL to NAM. When the contraction follows the recursion line, as in Figure 7, compilation cost drops from FUL to REC as recursion is removed and peaks sharply at XXX when call-by-name is deleted. In the third case (Figure 8), features related to switches and designational expressions are factored out first, and then all the features involved in the two procedure lines are removed at once. The cost of removing call-by-name still shows up as the language is contracted to BLK, but the cost jump is less pronounced than in the other two cases because a number of other, unused features are removed at the same time. For the portion of the contraction below BLK, the Innerproduct algorithm behaves like the "Typical" algorithm: the cost drops until BAS is reached, and then increases to TSF. These effects were common to virtually all the programs simulated: sharp peaks corresponding to features used to good advantage by an algorithm and a minimum cost at about the level of Basic.

5. CONCLUSIONS AND EXTENSIONS

5.1. General form of the compilation cost function

These and similar results suggest a general form for the compilation cost function. The results show two types of costs involved in the contraction of a language. If a language feature is present but rarely or never used, all programs incur the overhead cost of keeping the feature available. On the other hand, if a feature is needed but is not available, there is a cost associated with the inefficiency of having to rewrite the program to accomplish the same thing in the remainder of the language. Moreover, the overhead cost of an unused feature is small relative to the cost of emulating an unavailable feature of the same magnitude. These costs have opposing effects: the overhead costs tend to drive the language to be small while the emulation costs drive the language to be large. The choice of language to be used by any group of programmers should thus be driven by the characteristics of their particular collection of problems. The general function describing compilation cost as a function of language size (as measured by compiler size) is sketched in Figure 9. The break where the feature is added represents the emulation cost of the deleted feature. The



a) along call-by-name line

Figure 5. Billing Cost vs. Language Size for Typical

overhead cost of other unused features causes the decrease in cost with decreased language power. Since this function is driven by the overhead cost when no useful feature is removed and by the emulation cost when a useful feature is involved, the general function depends on both the languages of the contraction and the application for which the contraction is intended.

5.2. Appropriateness of Basic

Basic was designed as a problem solving language for beginners. It has succeeded as such a language, widely accepted as easy to learn and teach. Results such as the ones presented here suggest one reason this is so. The cost functions

Billing Cost vs. Language Size for InProd

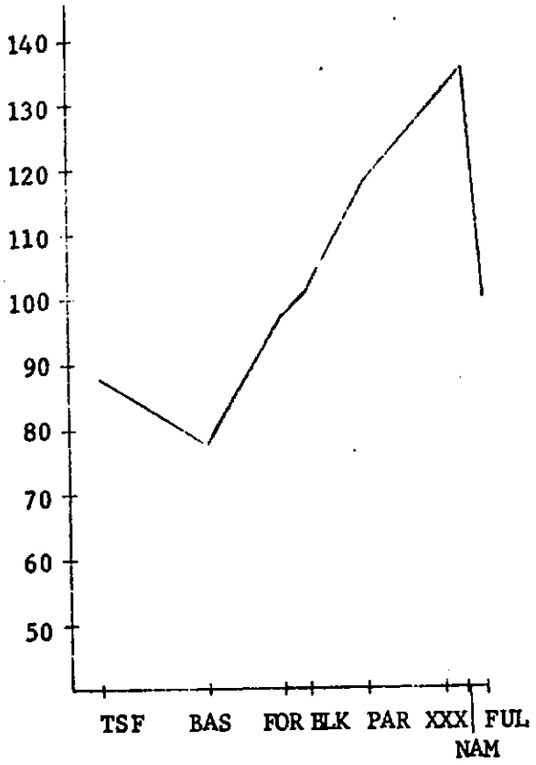


Fig. 6. along call-by-name line

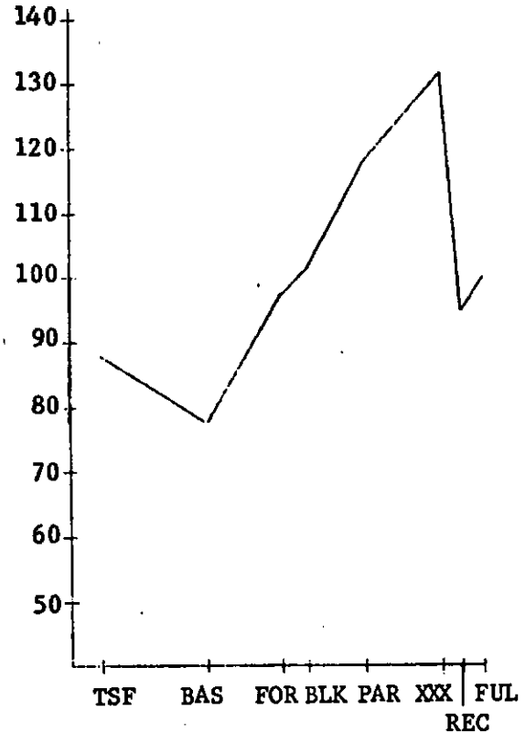


Fig. 7. along recursion line

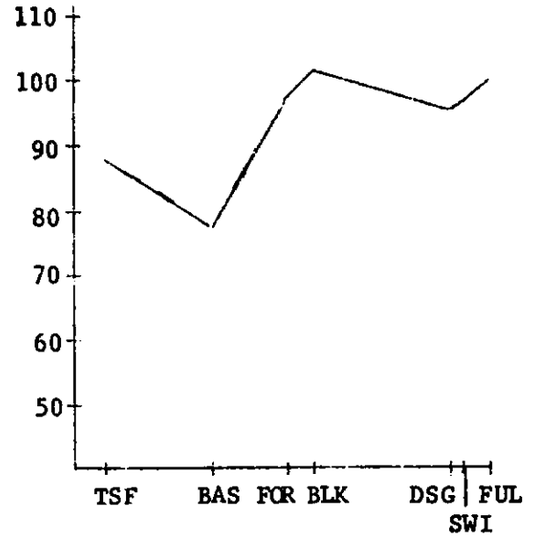


Fig. 8. along switch line

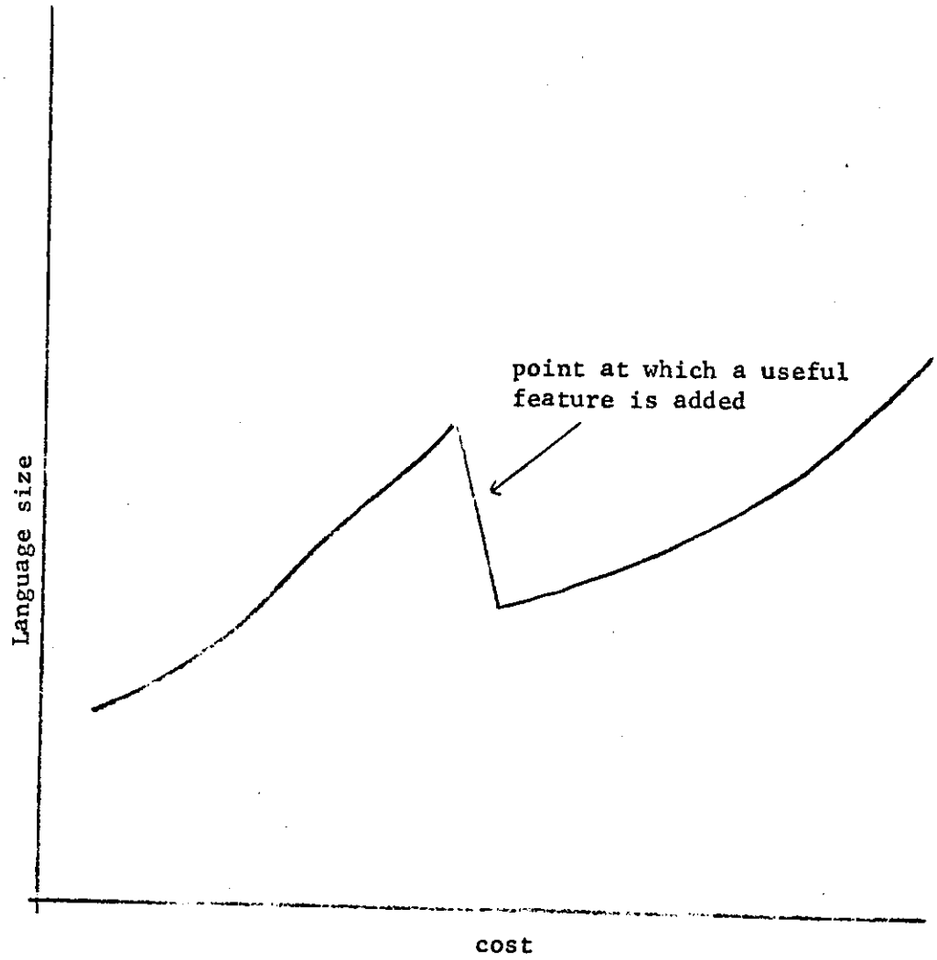


Figure 9. General Form for Compilation Cost Function

produced by the simulation often show minima or relative minima for the language BAS, whose power is similar to that of Basic. This indicates that Basic is a good language for many problems, and that it is an appropriate small language for a contraction.

5.3. Extension to total programming cost

Before this approach to compiler design and implementation can be applied to the generation of actual systems, similar analyses of programming and runtime costs must be performed. Programming effort is a very personal measure, and correspondingly hard to quantify. A programmer's notions about what are "natural" notations and facilities are influenced strongly by the languages he has used and with which he feels at home. This cost should be directly related to some measure of expressibility of the language: the more natural the notations for any application, the easier it should be to write algorithms related to that application. If a language is too small, the programmer will be frustrated by having to write many (to him) simple operations; if it is too large, he may be confused by the variety of options or simply ignore large segments of the language. Even in a contractible language, if a programmer is using too large a member of the contraction, he will have to put up with unnecessary overhead and may have to write extra instructions to avoid invoking features he does not want. Runtime costs should be susceptible to the same type of analysis as used here for compile-time costs. Just as many language features have costs directly allocatable in the compiler, so many features have costs directly allocatable to runtime facilities. For example, runtime overhead for block administration, the stack, the display, and type testing can be eliminated by suitable factorizations of Algol. Like the compilation cost estimates, the precise results will depend on the application intended as well as on the sublanguage. Finally, these cost functions should be combined to obtain estimates of total programming costs. By evaluating human cost in some appropriate (as yet unspecified) way and taking a weighted sum with the compilation and execution cost functions, we may obtain such total cost functions for various classes of problems. The expected result is that the total cost functions have minima somewhere within the available range of language sizes. This shape will indicate that overall costs can be reduced by judicious selection of the language subset to be used for each application. The available information supports this expectation.

5.4. Expanding a contractible compiler

In addition to building contractible compilers, it should be possible to build expandible ones. In expandible compilers, alternative forms of various features and alternative forms of language development will be available. after Algol is contracted to as small a language as appears useful for meaningful tasks, some constructs can be replaced, but not necessarily in the same order or with the same features as were removed. By selecting different features to add to the small language, it should be possible to generate families of languages similar in structure to the original, but different in detail. The choice of factorizations will, of course, affect the resulting language families.

6. REFERENCES

[CACM] - Communications of the ACM, Algorithms section, various dates.

[Eva64] - Evans, Arthur, "An Algol 60 Compiler", Annual Review in Automatic Programming, vol 4, Pergamon Press, 1964, pp. 87-124.

[Ken70] - Kennedy, Michael and Martin B. Solomon, Ten Statement Fortran Plus Fortran IV, Prentice-Hall, 1970.

[Knu70] - Knuth, D. E., An Empirical Study of Fortran Programs, Computer Science Department, Stanford University, CS-186, 1970.

[Knu.nd] - Knuth, D. E., private communication. Reproduced in [Shaw71], Appendix A.

[Naur63] - Naur, P. and M. Woodger, eds., "Revised Report on the Algorithmic Language Algol 60", CACM 6,1 (January 1963), pp. 1-20.

[Shaw71] - Shaw, Mary, Language Structures for Contractible Compilers, Computer Science Department Report, Carnegie-Mellon University, December 1971.

[Wich70] - Wichmann, B. A., Some Statistics from Algol Programs, National Physical Laboratory, Central Computer Unit Report 11, August 1970.

[Wir66a] - Wirth, Niklaus and Helmut Weber, "Euler: A Generalization of Algol and its Formal Definition", CACM 9,1 (January 1966) and CACM 9,2 (February 1966).

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Computer Science Department Carnegie-Mellon University Pittsburgh, Pa. 15213	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

REDUCTION OF COMPILATION COSTS THROUGH LANGUAGE CONTRACTION

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Scientific Final

5. AUTHOR(S) *(First name, middle initial, last name)*

Mary Shaw

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
July 17, 1972	20	9

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-70-C-0107 b. PROJECT NO. 9769 c. 61102F d.681304	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH OTHER	Air Force Office of Scientific Rsch (NM) 1400 Wilson Blvd. Arlington, Va. 22209

13. ABSTRACT

Programming languages tailored to particular groups of users can often be constructed by removing unwanted features from a general purpose language. This paper describes the use of simulation techniques to predict the savings in compilation cost achievable by such an approach. The results suggest a function which describes the effect of changes in the power of a language on the compilation cost of an algorithm expressed in that language: when features not actually used by the algorithm are removed from the language the cost of compiling the algorithm decreases moderately, but when features that are needed are removed, the compilation cost increases sharply.

DD FORM 1 NOV 65 1473

Security Classification