A Formalism for Translator Interactions

JAY EARLEY* AND HOWARD STURGIS†
University of California, Berkeley, California

A formalism is presented for describing the actions of processors for programming languages—compilers, interpreters, assemblers—and their interactions in complex systems such as compiler-compilers or extendible languages. The formalism here might be used to define and answer such a question as "Can one do bootstrapping using a metacompiler whose metaphase is interpretive?" In addition an algorithm is presented for deciding whether or not a given system can be produced from a given set of component processors.

KEY WORDS AND PHRASES: translator, compiler, interpreter, bootstrapping, language processor, compiler-compiler
CR CATEGORIES: 4.1, 5.29

1. Introduction

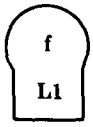
There have been many complex systems built for the implementation of programming languages. Some translate the source language into a different language (often machine language): these are called compilers, assemblers, or translators.[1] Some work directly with the source language to execute the program: these are the pure interpreters. Many have various combinations of the two modes of operation. In this paper we present a formalism which allows us to describe the interactions of the different phases of such systems.

Previous notations with similar aims [1, 2, 8, and 10] have concerned themselves only with translators, ignoring interpreters completely. With the advent of time sharing and conversational languages, interpreters are being used more and more frequently; so we feel that the introduction of a notation which also allows the description of systems involving interpreters is needed. The previous four notations seem to be isomorphic, with Bratman's "T-diagram" the most readable [1]. Therefore, we have adopted his notation and generalized it to reflect our fundamental notions of interpretation and translation.

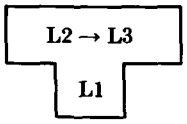
Sections 2 and 3 of this paper introduce our notation and apply it to the description of some interesting systems. Section 4 is a detailed study of bootstrapping using the notation. Section 5 provides a mathematical basis for the notation and proves that the intuitive steps that we have used previously are actually valid on a real computer. We conclude in Section 6 with a decision algorithm for determining what systems can be constructed from a given set of processors.

2. The Notation

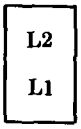
Our formalism contains one fundamental concept: A program written in language L1 which computes function f is represented as follows:[2]



We furthermore distinguish two special cases of function f, one in which f translates one language into another, and one in which f interprets a language. These are defined precisely in Section 5. We are concerned here only with the notation. A translator written in L1 to translate L2 to L3 is represented as

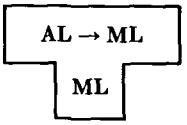


An interpreter written in L1 which interprets programs written in language L2 is represented as



In addition, a machine which executes machine language L1 is represented as

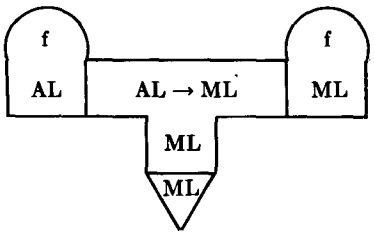
We compose these boxes to form descriptions of systems by placing them adjacent to each other such that vertical adjacency denotes interpretation and horizontal adjacency denotes translation. Thus, if an assembler for assembly language AL on a machine with machine language ML is

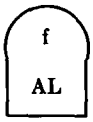
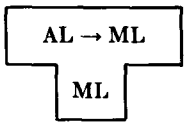


* Department of Computer Science.
† Computer Center and Department of Computer Science.
[1] Some people use "translator" to mean either a compiler or an interpreter. Except in the title of this paper, we use "processor" to mean a compiler or an interpreter and restrict "translator" to mean those programs which translate one language into another.

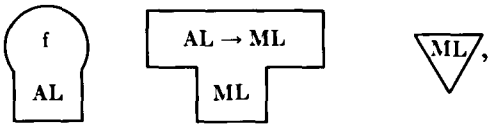
[2] The reader should realize that the function f in roman type in the figure notations is the same function *f* that appears in italic type in the text.

a run of this assembler on a program to compute function f is



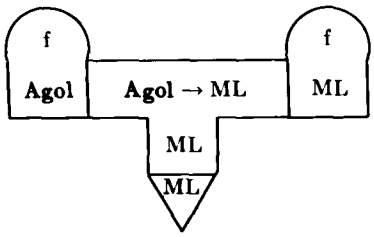
The meaning of this is that the program  is input as data to the translator  , and the program is the output.

This is an example of a "run," which is an important concept in this paper. It represents an actual execution of a program in a computer to produce an output. It also denotes that, given the three boxes



we can produce .

An Algol compiler [3] which goes directly to machine code looks almost the same:



A pure interpreter for a language like JOSS is represented as

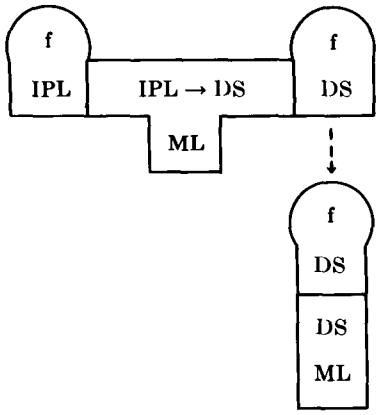
and a run of the JOSS system on a JOSS program to compute function f is as follows:

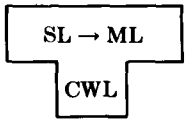
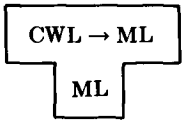
Notice that the bottom box of any configuration representing a run must be a machine. Consequently, we will omit the triangular machine boxes from the rest of the figures.

3. Some Examples

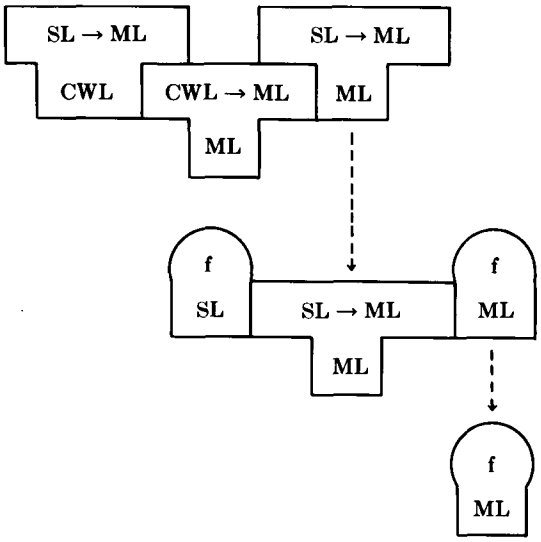
We have now introduced the notation; so let us apply it to some more interesting examples.

IPL [6] is normally implemented as a partial interpreter; that is, the source language is first converted into internal IPL data structures (DS) by the IPL loader. These are then interpreted to execute the program. We represent this process in two phases as follows:



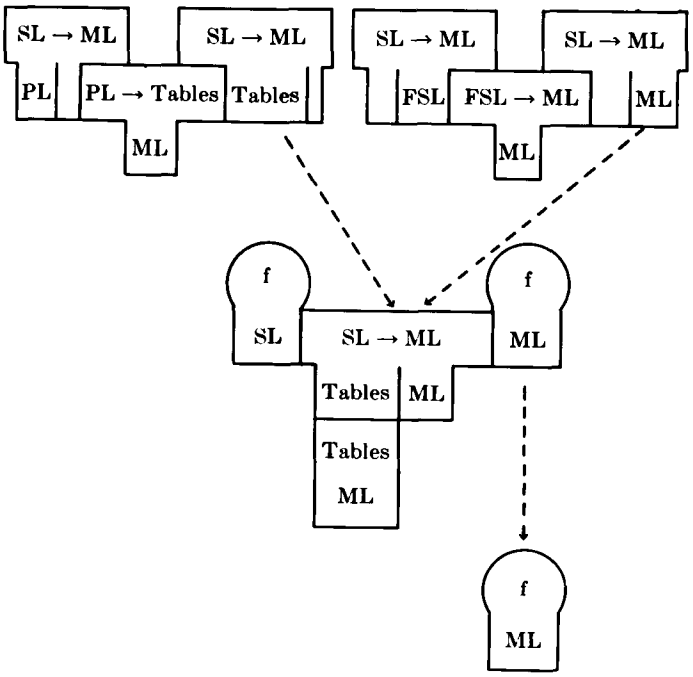
A compiler-compiler has three phases. The metaphase produces a compiler from a description in some higher level compiler-writing language (CWL). Here we are using a translator itself  as input to another translator  , and producing a third translator as a result.

The last two phases look like an ordinary compiler for the source language (SL). Notice that a compiler really has two phases, not just one, but the second is trivially represented by one box in our notation.

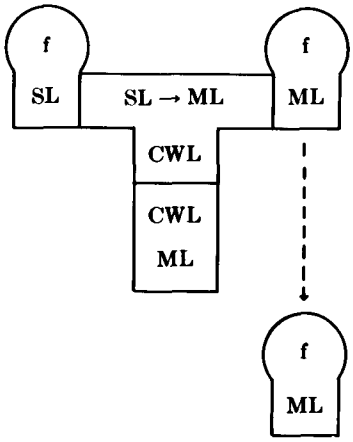


We will use "COMPILE(SL) and RUN" as an abbreviation for the last two phases of the above figure.

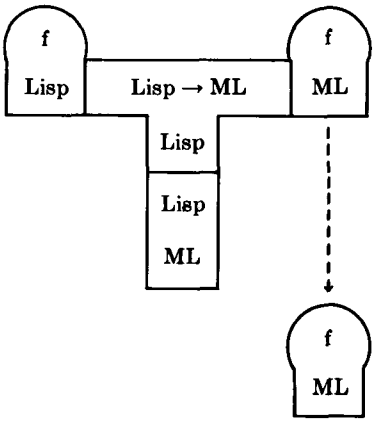
FSL [4, 9] is a compiler-compiler with a more complex structure than this. Its metaphase has two halves: the production language (PL) handles syntax, and FSL handles semantics. FSL is compiled, but PL is partially interpreted. We can fudge our representation a little to produce the following representation of the FSL system.



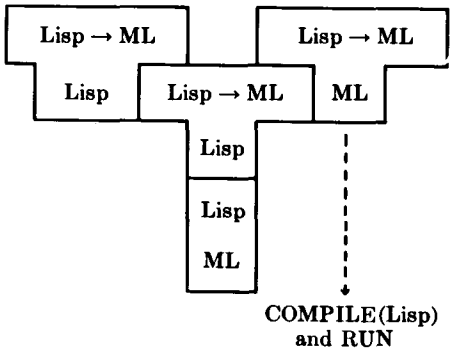
The metaphase of a translator writing system might also be a pure interpreter, directly interpreting CWL to compile the source language.



An interesting variant of this was used to write the Lisp 1.5 compiler [5]. Lisp is actually implemented as a partial interpreter, but since we are abstracting from reality anyway, we choose to abstract the parts of the system being described which are important for our purposes. So for this purpose we choose to ignore the loading phase of Lisp and to treat it as a pure interpreter. In this figure, we see the Lisp interpreter being used to execute a Lisp compiler written in Lisp.

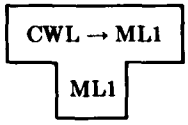


However, since the metalanguage and the source language are both Lisp, an extra phase was added to the system to obtain a machine language version of the compiler. The compiler was commanded to compile itself!

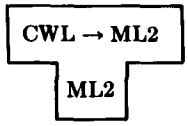


4. Bootstrapping

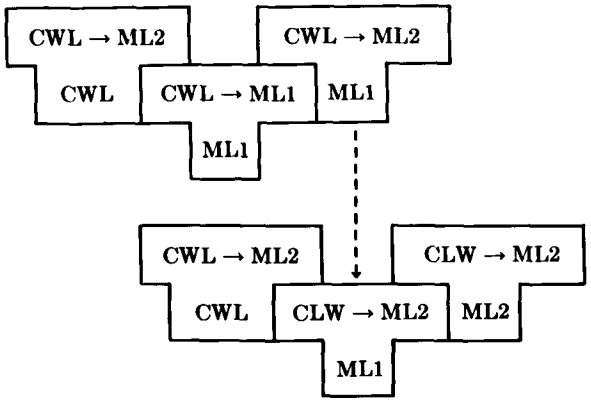
There are at least two quite different kinds of bootstrapping—moving a program from one machine to another, and writing a compiler in earlier versions of itself; we will examine both. The first kind can be illustrated by the following problem. We have two machines, with machine languages ML1 and ML2, and we have a compiler-compiler working in ML1:



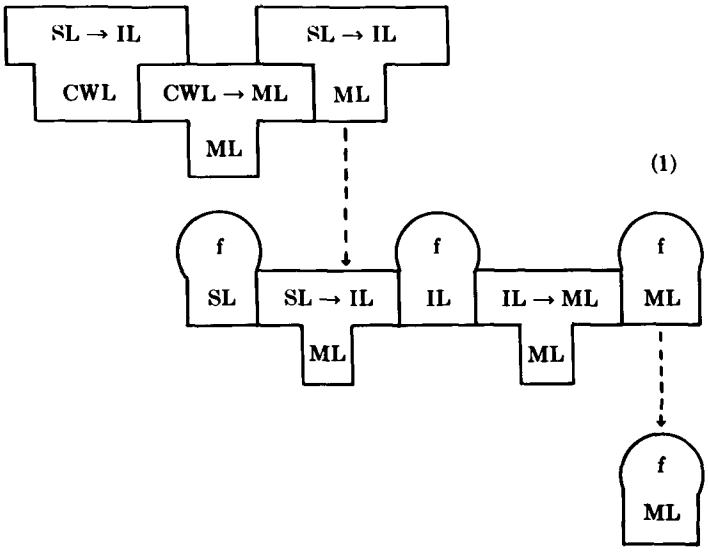
We would like to get a compiler-compiler for ML2 also



with a minimum of effort. This can be done by writing in CWL a compiler into ML2, and then performing the following process:

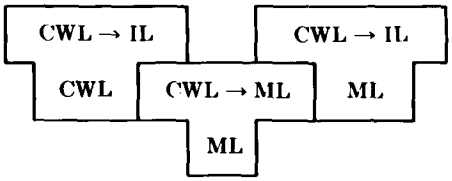


A compiler-compiler often works in two stages as follows: The compiler writer writes in CWL a program to translate his SL into an intermediate language (IL); a machine language compiler for IL has already been provided.

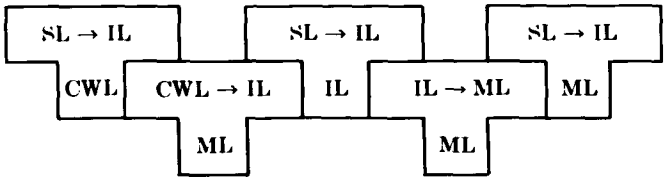


(1)

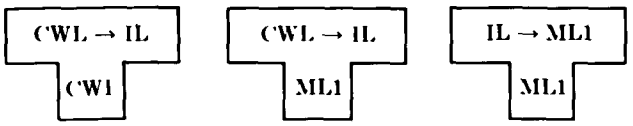
Often in a system such as this, there is actually a phase previous to the above two in which the CWL is implemented in itself.



In this case, the first phase of (1) must be rewritten, showing that the compiler is also compiled in two steps.

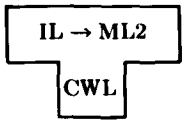


This is good for documentation of the system, and it makes it easier to modify. In addition, if we want to bootstrap the system to a second machine (M2), it involves only writing an IL compiler which compiles into ML2. This can even be done in CWL. The problem is stated as follows: Given



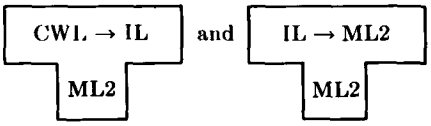
(2)

and having written



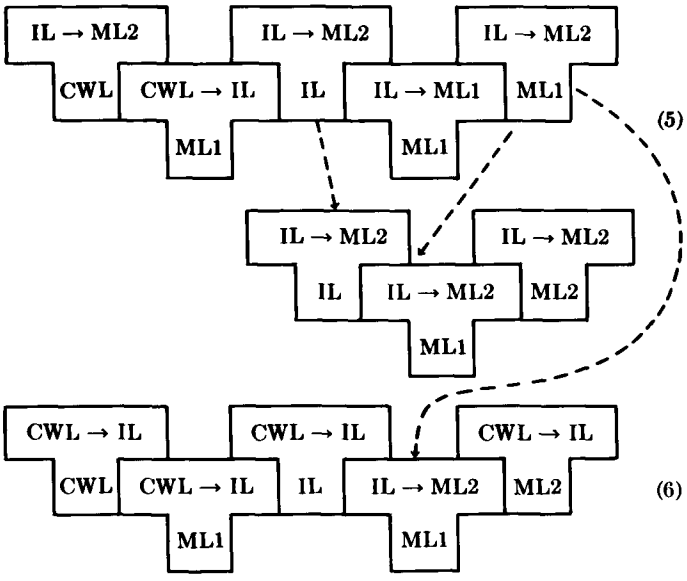
(3)

produce



(4)

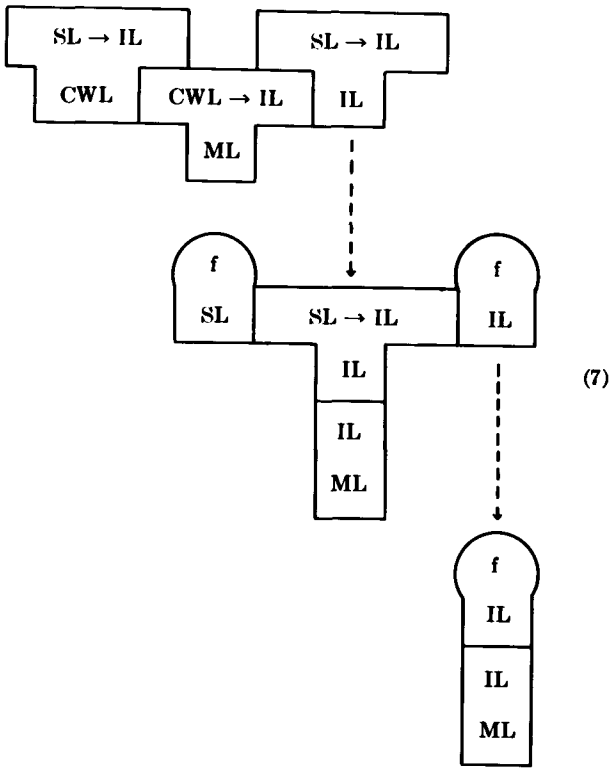
The solution is



(5)

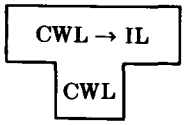
(6)

Notice that this can be accomplished by running on ML1 only. Another kind of metacompiler might have the property that the processor it produces is a partial interpreter; that is, it translates into IL first, and then interprets that. This looks like:



(7)

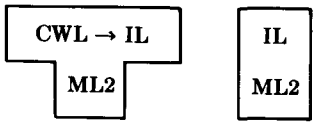
Furthermore, if the system were written in itself, we would have available



The bootstrapping problem for this system is as follows: Given

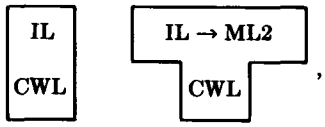
, (8)

produce



. (9)

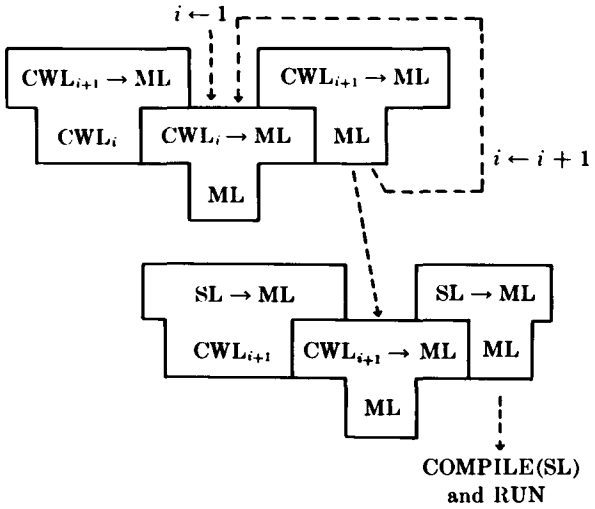
From analyzing possible solutions, we discover that we cannot get away as easily as we did in the previous system. Writing an IL interpreter in CWL will not solve the problem, because it does not provide us a way to get anything in ML2. We must either code things directly in ML2 or write a compiler into ML2. If we code directly in ML2, we must write both boxes in (9) ourselves, so there will be no bootstrapping at all. If we write an IL interpreter and a compiler into ML2 both in CWL



, (10)

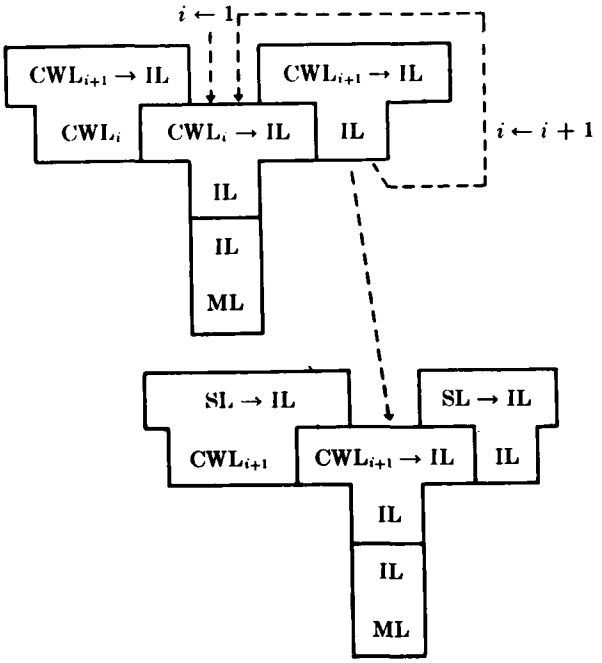
then we can produce both things in (9). But this is somewhat unpleasant because to do this we have had to write a compiler which is in no other way a part of our system. This points out the weakness of having a system that does not include a compiler into machine language. Of course, if IL and CWL are designed correctly, the process of writing the IL compiler in CWL could be little more than defining the meaning of ML2, but this is likely to be the exception, not the rule. In Section 6 we present an algorithm which will determine from any given set of boxes which other boxes can and cannot be produced, and we illustrate its use on this example.

The other kind of bootstrapping is used to write a compiler-compiler in itself. We code a compiler for a very simple version of CWL (CWL_1) in machine language or whatever else is available. Then we code a more advanced version (CWL_2) in CWL_1 and compile it with the first compiler. We can repeat this as many times as we like, and at any point in the process we can use the compiler-compiler as it was intended, to produce a compiler for some other language (SL).



This process may also be viewed as language extension, where the extended language is defined not only in terms of the base language (CWL_1) but also in terms of previous extensions. Therefore this discussion applies not only to compiler-compilers, but also to extendible languages.

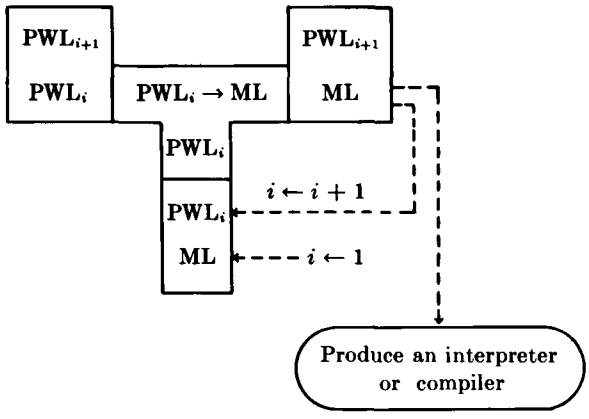
Similarly we can bootstrap or extend a system like that in (7) as follows:



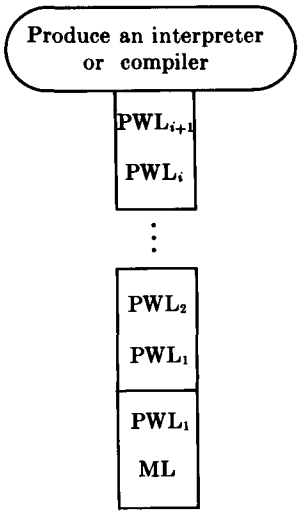
Notice, however, that this can only be used to extend the part of the language which is compiled. We may have wanted to extend IL to add a language feature which we wanted to be interpreted (such as dynamic block structure). In this case the above kind of bootstrapping is not possible, and the extension may have to be written in ML.

We now investigate this further by considering bootstrapping on a system which is a pure interpreter. Suppose we have a metalanguage (PWL) in which we can write both interpreters and compilers, and that we start with an interpreter for it.

One way of bootstrapping this is to also write a compiler for each PWL_i in itself. Then we can extend the interpreter as follows:



This method is awkward, but it does provide a way of extending a pure interpreter without writing in machine language. Another method of extension is possible and less awkward, but it is much less efficient.



Here we are extending PWL each time but downgrading its efficiency enormously each time.

5. Mathematical Basis

We would like to provide a precise basis for our notation, one which not only is rigorous, but also corresponds to the reality of running a computer system in terms of machines, bits, and programs. So we choose a basic vocabulary V, which can be thought of as bits, characters, punched cards, or anything similar. We would like to define the functions f, which we have used in our notation, as simply partial functions on words in V^* (the set of all sequences of elements from V). However, notice that these not only represent functions computed by programs written in certain languages, but also represent functions which correspond to the actions of computers. That is, a machine is a function which reads in some deck of cards in V^* (including both program and data) and writes out a deck in V^*. So it is, in fact, a function which is mathematically similar to a function which we might represent as a program, like the square root function. At first look, it may seem unnatural to lump these two together under the same concept, but on closer inspection we notice the following things.

The main difference between the two is that while the one (the square root) takes only data as input, the other (the 360) takes both data and program. After all, we could wire up a computer to compute square roots directly, or we could conversely write a simulator for the 360 computer in some language. Furthermore, an interpreter (which is a fundamental notion for us) is precisely a function which takes both program and data as input. If we are running an interpreter, then the machine it runs on has three inputs, two programs, and some data. One can see that this is leading us far afield, so we conclude that "machine" functions and "program-defined" functions are basically the same thing. We will call these *machine functions*.

Now, if a machine is to execute a program correctly, it must be able to separate the program from its data in the input word. Not all machine functions do this, and the effort required to define those that do would substantially complicate our formalism. So since we are not concerned about this particular problem here, we will define it out of existence by specifying that the input to a machine function is not a single deck in V^* but a sequence of decks, of which the first is the program and the rest are the data. The interpreter example should point out why it must be a sequence of decks and not just two.

Definition. Let $\Sigma(= V^*)$ be an infinite alphabet of decks.

Let F (the set of machine functions) be the set of all partial functions f: $\Sigma^* \rightarrow \Sigma$.

Let $\mathfrak{L}$ (the set of languages) be the set of all total functions L: $\Sigma \rightarrow F$.

The partial functions f represent physical translation processes that take place in a computer, while the functions L simply establish a correspondence between decks in Σ (representing programs in L) and machine functions in F (representing the function which that particular program computes). The functions f are partial because the program may loop on some of its inputs. We have chosen to make the functions L total to simplify the formalism, despite the fact that normal programming languages are only defined on certain decks. For any deck w which represents a program not in the language, let $L(w)$ be the machine function which is undefined everywhere.

We now define a convenient special function.

Definition. χ: $F \times \Sigma^* \rightarrow F$ is defined as $\chi(f, w_1)(w_2) = f(w_1w_2)$ for $f \in F$, $w_1, w_2 \in \Sigma^*$.

(Here and elsewhere in this section, equality means that if one is defined, then the other is, also, and they are equal.)

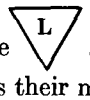
We omit the proof of the following trivial lemma:

LEMMA 1. For $w_1, w_2 \in \Sigma^*$, $f \in F$, $\chi(f, w_1w_2) = \chi(\chi(f, w_1), w_2)$.

We now notice that for every machine function f there is a machine language L_f in which one may write programs for the machine.

Definition. For $f \in F$, define $L_f \in \mathfrak{L}$ such that for all $p \in \Sigma$, $L_f(p) = \chi(f, p)$.

This is the language which treats its programs the same way that f treats the first deck of a string of decks it receives as input.

Conversely, we can define 

as the set of all machine functions f which have L as their machine language.[3]

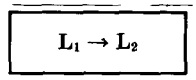
Definition. For $L \in \mathfrak{L}$,

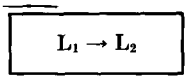
$= \{f \in F \mid$ for all $p \in \Sigma$, $L(p) = \chi(f, p)\}$.

This set may contain more than one element because it does not specify what f does to the empty string of decks.

[3] In this section, f, L, and S that appear in roman type in the figure notations are the same f, L, and S that are given in italic type in the text.

A set of translators is defined as follows:

Definition. 

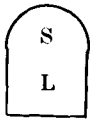
$= \{f \in F \mid$ for all $p \in \Sigma$, $L_1(p) = L_2(f(p))\}$. So 

defines a set of machine functions which translate programs in L_1 to equivalent programs in L_2.

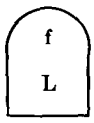
Equivalence of programs p_1 and p_2 is defined by $L_1(p_1) = L_2(p_2)$, that is, each program corresponds to the same machine function. Notice that a set defined in this way really can contain more than one element. For instance, there is more than one mapping from Algol programs to Fortran programs, because for any given Algol program there is more than one equivalent Fortran program. In addition, if L_2 is so weak that it cannot express algorithms for all recursive functions, then a set defined in this way may in fact be empty.

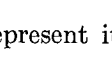
We are now prepared to define our fundamental concept.

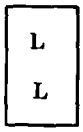
Definition. For $L \in \mathfrak{L}$, $S \subseteq F$,

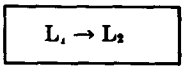


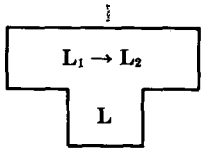
$= \{pw \mid p \in \Sigma, w \in \Sigma^*, \chi(L(p), w) \in S\}$.

It is the set of all programs (or possibly interpreters plus programs) in language L which compute a machine function in S. The three boxes which we have used in our notation are special cases of this. If S is a set with one element f, we represent it as 

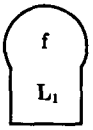
If S is 

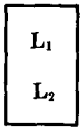
for some L', then this corresponds to 

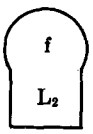
If S is defined as 

, then this corresponds to 

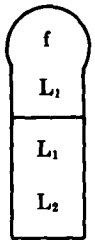
We would now like to prove that the operations of adjoining boxes to represent systems and to produce new boxes which we have used so far in this paper are valid in our formalism. We do this by proving two theorems.

The first says that if 

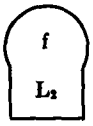
and 

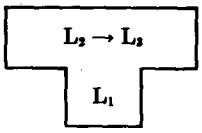
are nonempty then 

is nonempty. We express this

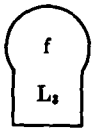
operation as 

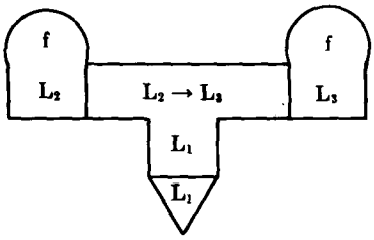
. This allows us to stack up as many interpreters as we like and put any box on top. The second theorem says that if



, and 

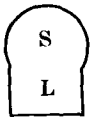
and

are nonempty then 

is nonempty. We express this operation as 

. This allows us to produce new boxes by translation.

LEMMA 2. If $f \in$

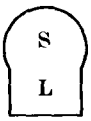
and $w \in$ 

, then $\chi(f, w) \in S$.

PROOF. Let $w = pw'$, $p \in \Sigma$, $w' \in \Sigma^*$.

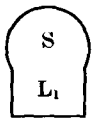
$\chi(f, w) = \chi(f, pw') = \chi(\chi(f, p), w')$ by Lemma 1

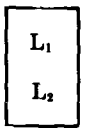
$= \chi(L(p), w')$ since $f \in$

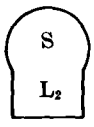
and $\chi(L(p), w') \in S$ since $w = pw' \in$ 

.

So $\chi(f, w) \in S$.

THEOREM 1. If $w_1 \in$ 

and $w_2 \in$ 

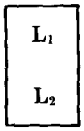
, then $w_2w_1 \in$ 

.

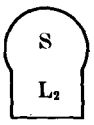
PROOF. Let $w_2 = pw_2'$, $p \in \Sigma$, $w_2' \in \Sigma^*$.

$\chi(L_2(p), w_2'w_1) = \chi(\chi(L_2(p), w_2'), w_1)$ by Lemma 1

and $\chi(L_2(p), w_2') \in$

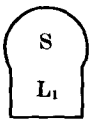
since $w_2 = pw_2' \in$ 

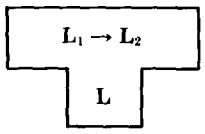
.

So $\chi(\chi(L_2(p), w_2'), w_1) \in S$ by Lemma 2. Thus $\chi(L_2(p), w_2'w_1) \in S$ and $w_2w_1 \in$ 

.

THEOREM 2. If

$pw_1 \in$ 

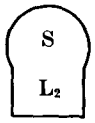
, $w_2 \in$ 

,

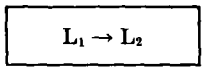
and

$f \in$

then

$f(w_2p)w_1 \in$ 

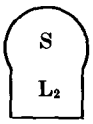
.

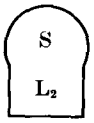
PROOF. $\chi(f, w_2) \in$ 

by Lemma 2.

So

$$L_2(\chi(f, w_2)(p)) = L_1(p)$$
$$\chi(L_2(\chi(f, w_2)(p)), w_1) = \chi(L_1(p), w_1) \in S$$

$\chi(f, w_2)(p)w_1 \in$ 

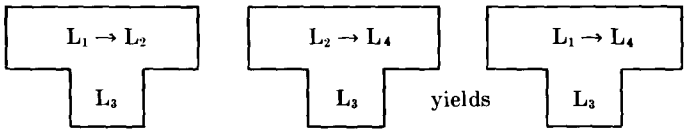
$f(w_2p)w_1 \in$ 

.

Notice that in translating a string of decks, we translate just the first deck and place the translated deck in front of the rest of the string.

Theorems 1 and 2 together provide a basis for the notation we have used in this paper.

One may wonder why we have not included a theorem which allows us to glue translators together since Theorem 1 allows us to glue interpreters together. This might read:



The most important reason for not including this is that any straightforward theorem one might derive from it is false. That is, if we concatenate two decks representing the translators on the left, we do not get a deck representing the one on the right (as is true with interpreters). At the very least, we must include a third program which feeds the output of one as the input to the other. This introduces enormous complications which our formalism, as it is, does not handle.

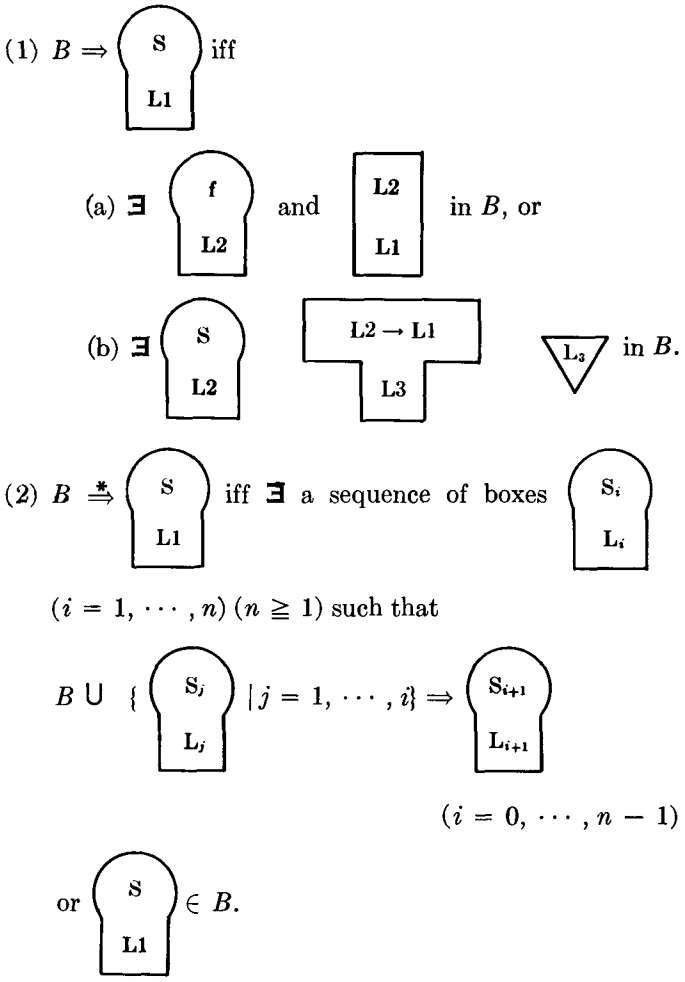
6. A Decision Algorithm

Let us examine the problem presented in diagrams (2), (3), and (4) of Section 4. We are given a certain set of translators and interpreters represented by boxes and we would like to know whether or not we can produce other boxes which we need from them. We can show that certain boxes can be produced by exhibiting a system which constructs them as in diagrams (5) and (6). But we would also like to be able to prove that certain boxes cannot be produced by the methods we have.

In fact we can do better than this. In this section we present an algorithm which, given a set of input boxes and a desired box, will determine whether or not that box can be produced and will supply the construction which produces it if it can be produced.

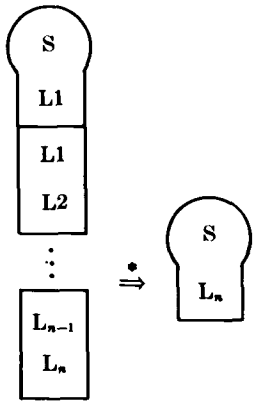
Before exhibiting the algorithm, we will formalize exactly what steps may be used to produce new boxes.

Definition. Given a finite set B of boxes (a box is a subset of Σ^+ or a set over F)



Definition (1) specifies the two ways in which a new box may be produced according to Theorems 1 and 2. Note that we have required specifically a set of machine boxes . This corresponds to the set of physical computers we are using. It actually requires that only the bottom element of a stack of interpreters be written in a machine language.

This comes about because we can obtain the following by repeatedly applying definition (1) part (a):



Definition (2) specifies the way in which a box can be produced by any number of applications of definition (1) (including 0 applications if the box is already in the set).

We now present an algorithm for generating the set of all boxes which can be produced from a given set.

ALGORITHM. Given a set B of boxes, compute Closure(B) as follows:

A: For each pair of boxes do the following: If the pair matches the conditions of definition (1) part (a), let b be the box resulting from applying that definition. If $b \in B$ go on. If $b \notin B$, let $B = Bu\{b\}$ and start over at A. When all pairs have been exhausted without starting over, repeat the process for all triples and definition (1) part (b). When these are all exhausted without starting over, the algorithm terminates.

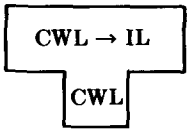
THEOREM 3. Closure(B) = $\{b \mid B \overset{*}{\Rightarrow} b\}$.

PROOF. Closure(B) must be finite for the following reason: Any box in Closure(B) must consist of a language which originally existed in a box in B and a function which originally existed in a box in B. Since both these sets are finite, their Cartesian product is finite. Therefore the closure algorithm always terminates. The rest of the proof is obvious.

Thus we generate all the boxes which can be produced from a given set by a simple exhaustive search. More efficient algorithms may be devised, but we do not examine them here for two reasons: (1) we are mainly concerned with showing that a decision procedure exists; and (2) the set Closure(B) will be small enough in practice that a faster algorithm will not gain much.

We can now, of course, decide whether or not a given box can be produced from a set B by computing Closure(B) and testing the box against each element of Closure(B) as it is generated.

We now provide an example of the use of this algorithm. Consider the problem given in (8) and (9). We have concluded correctly that writing the boxes in (10) will allow

us to produce the boxes in (9). Suppose, however, that we did not have box 

.

This box is not part of the original system in (7). We just assumed that it had come about through a prior bootstrapping process. Suppose we ask whether or not the problem can be solved without this box. We will run the closure algorithm on it in order to answer this question. The results are in Figure 1. The initial set B is given at the top. We then show each application of Definition 1 and the box it produced. We have omitted all applications which produce a box in B or one previously produced.

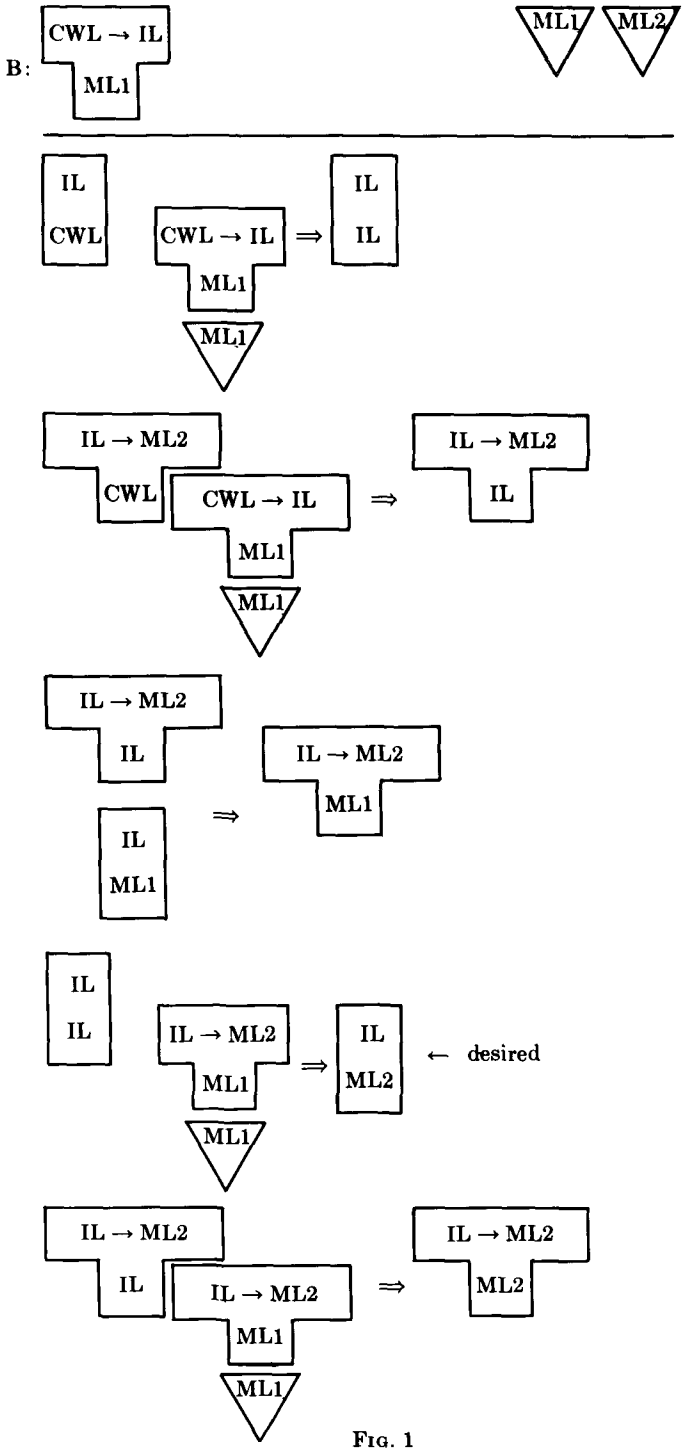
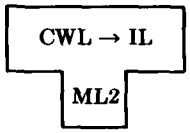
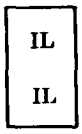


FIG. 1

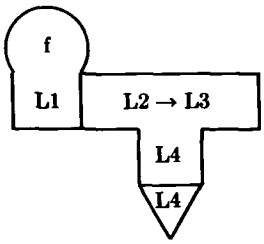
Notice that one of our desired boxes

did get produced, while the other 

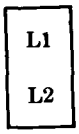
did not. Notice also that the seemingly innocuous box 

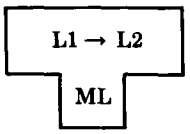
was crucial in obtaining the one desired box. This is because it could be translated into a different language.

This algorithm provides a decision procedure for what boxes can be produced by Definitions 1 and 2 only. It cannot say that a desired box cannot be produced by any means. This is because we have no way of knowing what will happen if a program written in $L1$ is given as data to a translator which translates programs written in $L2$. That is,

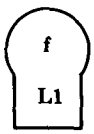


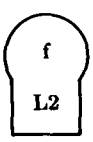
we have no idea what the result of this will be.

Notice that this algorithm will answer the question "Can language $L1$ be executed given that we can execute language $L2$?" because this is equivalent to asking "Can 

be produced?" The algorithm will also answer the question "Can $L1$ be translated into $L2$?" even though it is not equivalent to "Can 

be produced?"

The reason they are not equivalent is that translators cannot be glued together like interpreters since the translation may take more than one step. However, the question can be answered by adding 

to B and asking whether or not 

is in Closure(B).

7. Summary

We have introduced a formalism which allows us to explicate certain rather gross properties of language processing systems. As it is, the notation should be useful for designing the outlines of complex programming systems and their implementation, and it should be especially good for documentation. The formalism should also provide a mathematical basis which can be extended to handle more detailed properties of such systems. Some specific inadequacies where it could be extended follow.

1. It does not describe the amount of compilation or interpretation, unless it is coupled with precise definitions of the languages involved. For instance, in (7) we have no idea whether IL is close to machine language or to the source language. IL could be little more than assembly language, or just a trivial modification of the source language, or anything in between. Of course precise definitions of SL, IL, and ML would clear this up.

2. It does not permit the description of such processes as incremental compilation.

3. It does not permit the formal description of systems involving programs which consist of two or more pieces written in different languages, such as FSL.

Acknowledgment. We have benefitted from comments by J. Gray and J. Reynolds in preparing this paper.

RECEIVED JANUARY, 1970; REVISED JUNE, 1970

REFERENCES

1. BRATMAN, H. An alternate form of the "UNCOL diagram." *Comm. ACM 4*, 3 (Mar. 1961), 142.
2. BURKHARDT, W. H. Universal programming languages and processors: A brief survey and new concepts. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 1, Spartan Books, New York, pp. 1–21.
3. EVANS, A. An Algol 60 compiler. Proc. ACM 18th Nat. Conf., 1963.
4. FELDMAN, JEROME A. A formal semantics for computer languages and its application in a compiler-compiler. *Comm. ACM 9*, 1 (Jan. 1966), 3–12.
5. MCCARTHY, J., ET AL. *Lisp 1.5 Programmers Manual.* MIT Press, Cambridge, Mass., 1968, pp. 76–77.
6. NEWELL, A. *IPL-V Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1961.
7. SHAW, J. C. JOSS: A designer's view of an experimental online computing system. Rand Corp. P-2922, Santa Monica, Calif., Aug. 1964.
8. SKLANSKY, J., FINKELSTEIN, M., AND RUSSELL, E. C. A formalism for program translation. *J. ACM 15*, 2 (Apr. 1968), 165–175.
9. ITURRIAGA, R., STANDISH, T., KRUTAR, R., EARLEY, J. Techniques and advantages of using the formal compiler writing system FSL to implement a Formula Algol compiler. Proc. AFIPS 1966 Spring Joint Comput. Conf. Vol. 28, Spartan Books, New York, 241–252.
10. STRONG, J., ET AL. The problem of programming communication with changing machines: A proposed solution. *Comm. ACM 1*, 8 (Aug. 1958), 12–18; and *1*, 9 (Sept. 1958), 9–15.

ALGORITHM 395
STUDENT'S *t*-DISTRIBUTION [S14]

G. W. HILL (Recd. 17 Nov. 1969 and 23 Mar. 1970)
C.S.I.R.O., Division of Mathematical Statistics, Glen Osmond, South Australia

KEY WORDS AND PHRASES: Student's *t*-statistic, distribution function, approximation, asymptotic expansion
CR CATEGORIES: 5.12, 5.5

real procedure *student* (*t*, *n*, *normal*, *error*); **value** *t*, *n*; **real** *t*, *n*; **real procedure** *normal*, *error*;
comment *student* evaluates the two-tail probability $P(t \mid n)$ that t is exceeded in magnitude for Student's [1] t-distribution with n degrees of freedom. The procedure provides results accurate to 11 decimal places and 8 significant digits for integer values of n, with approximate continuation of the function through noninteger values of n (over 6 decimal places for $n > 4.3$).

The procedure *normal* (χ) returns the area under the standard normal frequency curve to the left of χ, so that a negative argument yields the lower-tail area. The user-supplied procedure, *error*(n), should produce a diagnostic warning and may **go to a** label, terminate, or return a distinctive value (zero or -1.0) as a signal of error to the calling program.

Student's series expansion of the probability integral is supplemented by a faster asymptotic approximation for large values of n and by a more precise "tail" series expansion for large values of t.

The value of χ, defined as the normal deviate at the same probability level as t, may be approximated by an asymptotic normalizing expansion of Cornish-Fisher type [2].

$$\chi = z + (z^3+3z)/b - (4z^7+33z^5+240z^3+855z)/10b^2$$
$$+(64z^{11}+788z^9+9801z^7+89775z^5+543375z^3+1788885z)/210b^3-\cdots$$

where $z = (a\times\ln(1+t^2/n))^{\frac{1}{2}}$, $a = n - \frac{1}{2}$ and $b = 48a^2$ [3]. This is well approximated by the first three terms with the third term's divisor replaced by

$$10b(b+0.8z^4+100).$$

The *student* probability is double the normal single-tail area, corresponding to the deviate χ.

The maximum error in the probability result for all values of t is displayed as a function of n in Figure 1, for this approximation, for the first few terms of the asymptotic expansion and for Fisher's [4] fifth-order approximation used in Algorithm 321 [5] for $n \geq 30$.

For small n and moderate t the result is calculated as $P(t \mid n) = 1 - A(t \mid n)$ using Student's cosine series for $A(t \mid n)$, rearranging formulas 26.7.3 and 26.7.4 of the NBS Handbook [6] in nested form

$$A(t|n \text{ odd}) = \frac{2}{\pi}\left[\arctan(y) + \frac{y}{b}\left\{1 + \frac{2}{3b}\left\{\cdots \frac{(n-5)}{(n-4)b}\cdot\left\{1+\frac{(n-3)}{(n-2)b}\right\}\cdots\right\}\right\}\right]$$

$$A(t|n \text{ even}) = \frac{y}{\sqrt{(b)}}\left\{1 + \frac{1}{2b}\left\{\cdots \frac{(n-5)}{(n-4)b}\left\{1+\frac{(n-3)}{(n-2)b}\right\}\cdots\right\}\right\},$$

where $y = \sqrt{(t^2/n)}$ and $b = 1 + t^2/n$. In the nested form, terms are treated in reverse order to the summation in Algorithm 321 and Algorithm 344 [7], reducing the number of operations required and reducing build up of roundoff error. Explicit decre-