AD-A063 985 WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIA--ETC(U)
OCT 78 F D CRARY, J M YOHE DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-1892 NL

END
DATE FILMED
4 79
DDC

(9) MRC Technical Summary Report #1892

(6) THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIAL PURPOSE ARITHMETIC PACKAGES

(10) F. D. Crary and J. M. Yohe

LEVEL II

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

(11) October 1978

(12) 21 p.

Received October 13, 1978

(15) DAAG29-75-C-0024

(14) MRC-TSR-1892

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIAL PURPOSE ARITHMETIC PACKAGES

F. D. Crary (1) and J. M. Yohe (2)

Technical Summary Report # 1892
October 1978

ABSTRACT

We discuss the use of a FORTRAN precompiler in the development of packages for nonstandard arithmetics. In particular, the use of the FORTRAN precompiler, AUGMENT, renders the source code more lucid, reduces the number of lines of code in a nonstandard arithmetic package, facilitates modification, and ameliorates the problems of transporting such a package to another host system.

AMS(MOS) Subject Classification: 68A10

Key words: Precompiler
Nonstandard arithmetic packages
Portable software
Software development
Software modification

Work Unit Number 8 (Computer Science)

(1) The Boeing Company
Seattle, Washington

(2) Mathematics Research Center
University of Wisconsin - Madison
Madison, Wisconsin 53706

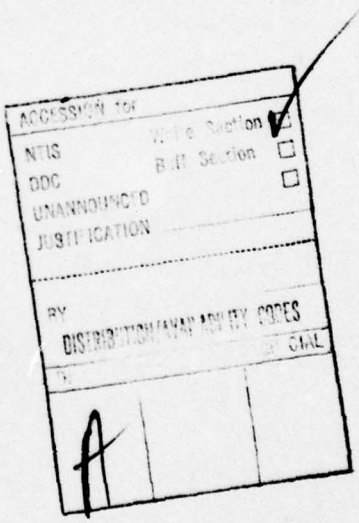
Sponsored by the U. S. Army under Contract No. DAAG29-75-C-0024.

SIGNIFICANCE AND EXPLANATION

The AUGMENT precompiler for FORTRAN [7] is the most recent and most versatile in a series of FORTRAN precompilers which have been developed at the Mathematics Research Center, University of Wisconsin - Madison, over the past decade. Originally, the precompiler concept was envisioned as a means of simplifying the use of special-purpose arithmetic packages, such as multiple precision arithmetic or interval arithmetic, in the research environment.

In recent years, however, AUGMENT has come to play a central role in the development of special-purpose arithmetic packages as well as in their use. Briefly, one may write the majority of the modules of a package in terms of one or more nonstandard data types, later using AUGMENT together with a few primitive modules to bind these data types to specific representations. This technique enhances the clarity of the code, reduces the number of lines of code in the package (in most cases), facilitates modifications (such as increasing precision), and ameliorates the problems of transporting such a package to other host systems.

In this paper, we discuss the advantages of this technique, and illustrate with examples taken from the INTERVAL arithmetic package developed by the second author [19]. In particular, we indicate how the interval arithmetic package was modified in the space of approximately one week to produce a package for triplex interval arithmetic [1], and we also discuss the nature of the modifications that will be necessary to produce a multiple precision interval arithmetic package based on the multiple precision arithmetic package of Brent [2]. Application of this technique to a variety of situations will be apparent.



The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the authors of this report.

THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIAL PURPOSE ARITHMETIC PACKAGES

F. D. Crary and J. M. Yohe

1. Introduction

With decreasing hardware costs and increasing processor speeds, the cost of software development is becoming more and more a function of personnel cost. Furthermore, with the explosion of applications of digital computers, an ever-higher percentage of users place implicit trust in the software they use to support their applications. Today, a computer system is regarded by most users not as an empty box which must be programmed to solve even the simplest problem, but rather as a hardware/software combination which can be used to solve their computational problems. Indeed, many users probably do not know -- and care less -- where the hardware leaves off and the software begins.

For these reasons, it is essential to supply the user with reliable, well-documented software packages. It is no longer profitable, or even feasible in many cases, to re-invent support software. The considerations of efficiency on a particular system have all but been erased by the increasing cost of highly competent personnel and the decreasing cost of hardware. Whereas twenty years ago code that wasted machine time could not be tolerated for many applications, we have now come to the point that code which wastes personnel time can not be tolerated.

These considerations have led to an increasing emphasis on transportable software. If development costs can be incurred just once for a package or system that will work correctly and accurately on a broad spectrum of equipment, users are willing to tolerate a reasonable amount of inefficiency in return for the convenience of having the development work done for them and the confidence that they can place in a quality product.

Increasingly, it is becoming practical to build on existing software rather than to develop new packages from first principles, even when the existing software might not be just exactly tailored to the application in question. For example, as we shall discuss later in this paper, it makes sense to base a multiple precision interval arithmetic package on Brent's excellent multiple precision arithmetic package ([2]) rather than to develop a new multiple precision package which implements the required directed roundings. The needed modifications can be introduced via a very few new modules, leaving the bulk of Brent's software development work intact.

In order to make the best use of existing software, one must have the tools to make its incorporation in new programs reasonably easy, and one must adopt a design philosophy which will make the use of both the tools and the existing software natural and uncomplicated.

In this paper, we describe one such tool -- the AUGMENT precompiler for FORTRAN ([7]) -- and illustrate a design philosophy which has proved to be a reasonable application of the above criteria. Briefly, we advocate the abstraction of the data type representations to the maximum possible degree in the design and implementation of software packages, and subsequent application

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

of the AUGMENT precompiler to bind the data type representations and extend them throughout the package.

We illustrate this philosophy with examples drawn from the interval arithmetic and triplex arithmetic packages developed by the second author.

We also give an indication of several other applications of AUGMENT which, while not necessarily employing this philosophy, serve to indicate the breadth of possible applications of AUGMENT.

In Section 2, we describe the AUGMENT precompiler briefly. This is done for completeness; although the AUGMENT precompiller is described extensively in [7] and in [5, 6], the present paper requires that the reader have at least a basic understanding of what AUGMENT is and what it does. In Section 3, we discuss the concept of abstract data types. In Section 4, we show how AUGMENT was used in the development of the interval arithmetic package and in Section 5 we indicate how this design philosophy has facilitated modifications to the interval arithmetic package. In Section 6 we give some brief examples of other applications of AUGMENT. Section 7 summarizes the paper.

2. Brief description of AUGMENT

AUGMENT is a program which allows the easy and natural use of nonstandard data types in Fortran. With only a couple of exceptions, it places nonstandard types on the same basis as standard types and allows the user to concentrate on his application rather than on the details of the data type implementation.

AUGMENT gains its power and ease of use through several aspects of its design.

(1) Its input language is very much like FORTRAN. The only changes are the addition of new type names and operators, and the ability to define "functions" naming parts ("fields") of variables which may appear on either side of the assignment operator. Thus it is very easy to use--much easier than learning the calling sequences of a package of subroutines.

(2) AUGMENT is extremely portable. Since it is written in FORTRAN, AUGMENT can be implemented on almost any computer (this should be qualified by saying that adequate memory must be available--one attempt to implement AUGMENT on a DEC PDP-11 was unsuccessful because of the limited address space). The machine-dependencies of AUGMENT are concentrated in eight subroutines which can be implemented in less than 200 lines of (machine-dependent) FORTRAN.

(3) AUGMENT's output is standard FORTRAN which makes it suitable as a cross-precompiler; that is, the AUGMENT translation may be performed on one (large) machine and the results compiled on or for some other machine which is unable to host AUGMENT. This was the approach taken when the interval arithmetic package was implemented on the PDP-11.

There are three major steps in the use of AUGMENT:

Specification of the nonstandard data type
Binding to a representation (implementation)
Application

As is readily seen, the first two steps may be very easy or may be skipped entirely in cases where appropriate research and development has already been performed. We give a brief sketch of these steps below; the remainder of the paper illustrates them in some specific cases.

Specification. The whole process begins with the specification of the properties of a nonstandard type. The specification will need to consider the following questions:

What information will the user see?
What operations will be made available?
How will this type interact with other types?

In many cases, the answers to these questions will be available in previous research or obvious from the nature of the new type. For example, a multiple precision floating point type should parallel the standards as much as possible. In other cases, considerable research may be needed and even an appeal to personal preference (there are at least two distinct schools of opinion on the proper way to define comparison in interval arithmetic).

AUGMENT gives little assistance to this part of the process. The specifications will be guided by the applications envisioned by the person preparing the new type, by the operations known or felt to useful in manipulating the type, and esthetic considerations such as consistency with similar types (if any) already existing in Fortran or previous extensions.

Binding (Implementation). The binding of the abstract specification of the new type to a representation usable through AUGMENT is by means of a "supporting package" of subroutines and functions, and through a "description deck" which tells AUGMENT about the supporting package. In this effort, the implementor must consider the conventions expected by AUGMENT in terms of argument number and order. These conventions are treated at length in the AUGMENT User Documentation [5].

In addition to this, there may remain basic questions of representation. For example, the data structure which the user sees may not necessarily be the best way to implement the type. Suppose that the new type is specified abstractly as a linear sorted list into which the user may insert and retrieve items. In this case, it might be much more efficient to implement as a binary tree with balancing and threading properties appropriate to the expected uses. The user might well remain ignorant of the underlying implementation and see only the linear sorted list.

Very often AUGMENT can be used to assist in the binding process. It may often be the case that only a few of the routines in the supporting package require knowledge of the binding. For example, if a new arithmetic type (such as multiple precision) is being implemented, then only the basic arithmetic and conversion routines would require knowledge of the underlying representations. The mathematical functions (SIN, ABS, etc) could be written using AUGMENT and be representation independent. This would make it easier to reimplement the package with, for example, a different precision. In fact, this has been done with the interval arithmetic packages as will be described later in the paper.

Application. The application of AUGMENT to preparation of a program which uses one or more nonstandard data types is by far the easiest part of the process. In fact, AUGMENT was designed so this would be the case. Given that the supporting package(s), description deck(s), and adequate documentation have already been prepared, the use of the package(s) through AUGMENT consists of just four steps:

(1) Write the program using the new operators and functions.
(2) Supply AUGMENT with your program and the description deck(s).
(3) Compile AUGMENT's output with the system FORTRAN compiler.
(4) Link-edit and run.

Since AUGMENT checks for as many errors as it can, it is usually unnecessary to examine the FORTRAN translation. However, we suspect that most users will wish to examine the output at first to convince themselves that AUGMENT does produce correct code. This desire would be expected to diminish as confidence grows (after all, how often does the normal user examine the output of the system compilers?).

3. Abstract data types

In the planning of most computations, we do not explicitly consider the architecture of the computer that will be processing the program or the specific representation that it will assign to real numbers, for example. In writing the code, however, most languages require that we make decisions early in the coding about such questions as precision, data representation, and so forth. This is not ordinarily onerous, since we are usually somewhat limited in our selection of data types anyhow, and certain variables naturally invite representation as integers, others as "REAL" variables, and yet others as DOUBLE PRECISION. We put the word "REAL" in quotes because we feel that this is an unfortunate choice for reference to a single-precision approximation to real numbers.

It is not unusual for persons trying to run a program written for a long-word machine on a short-word machine to experience problems related to precision. The reason for this is that they have come to depend on a large number of significant digits, a large exponent range, or both; the algorithm taxes the ability of the short-word machine to handle the problem. In such cases, the programmer is not really interested in what the data type is called, so long as it is an accurate approximation to the abstract mathematical system to produce valid results. Nevertheless, the user caught in such a squeeze must usually declare all of the former "REAL" variables to be DOUBLE PRECISION.

AUGMENT has a feature which will aid a person caught in such a trap; it can be instructed to convert any REAL variable to DOUBLE PRECISION. While this feature may save many hours of drudgery for the person who has a requirement for such direct conversion, it is only a hint at what we have found to be one of the major attractions of AUGMENT in writing special-purpose arithmetic packages: the ability to use abstract (unbound) data types throughout the majority of the programming, binding the data type to a specific representation only in the instructions to AUGMENT and in a few primitive modules of the package.

Thus, for example, one might write a package using the data type ETHEREAL (Express THEoretical REALs will do, if you must have an expansion for the acronym), later instructing AUGMENT to convert ETHEREAL to REAL, DOUBLE PRECISION, MULTIPLE PRECISION, or what have you. Other data types may then be defined as vectors or matrices of ETHEREAL numbers, and AUGMENT will be able to allocate the proper amount of space when it knows the binding of ETHEREAL. Moreover, the routines which manipulate the arrays of ETHEREAL numbers may all be written in terms of operations on ETHEREAL numbers; again, AUGMENT will put everything right at precompile time.

The following sections will illustrate this philosophy with concrete examples.

4. The use of AUGMENT in the construction of the INTERVAL package

The Interval Arithmetic Package described in [18] was motivated by interest in interval arithmetic on the part of several other universities with different computer systems. This interest was sparked by a project sponsored by the U. S. Army Corps of Engineers Waterways Experiment Station, Vicksburg, Mississippi; this project included implementation of interval arithmetic on various computer systems and comparison of the performance of the various systems.

The previous interval arithmetic package at the Mathematics Research Center [12] had been developed in the early 1970's, and was quite dependent on the UNIVAC 1100 series architecture. Extensive reprogramming would have been necessary for each new system, and it was therefore decided to rewrite the package for greater portability.

The revised package needed to be flexible enough to accommodate a wide variety of different computer architectures; indeed, since we wished to make the package adaptable to nearly any host environment, we wanted to leave the representation of intervals and interval endpoints arbitrary throughout the bulk of the package. But because of FORTRAN's popularity for scientific computation, it was the language of choice for implementing the package. Needless to say, ANSI standard FORTRAN does not have the flexibility we needed in order to accomplish the goals we had set.

We wanted to make the interval arithmetic package easily accessible from the user's point of view. This naturally led us to design the package to be interfaced with AUGMENT: indeed, this had been done also with the previous interval arithmetic package. But the requirements for flexibility and transportability led us to conclude that the package itself should be written with the aid of AUGMENT.

Before we discuss the role of AUGMENT in the implementation of the package, it would be appropriate to include a very brief description of interval arithmetic. The interested reader can find more details in [18] or [11].

Interval arithmetic is a means for bounding the error in computation by calculating with pairs of real numbers, the first member of the pair being a lower bound for the true result, and the second an upper bound. The foundations for interval mathematics have been carefully laid by Moore [14], Kulisch [11], and others, so interval mathematics is on firm theoretical ground. There are closed-form formulae for evaluating operations and functions on the space of intervals, so that computation with intervals is reasonably straightforward.

In machine interval arithmetic, one naturally represents an interval as a pair of approximate real numbers. One must then determine what approximation is to be used for real numbers, and develop the necessary software to evaluate the mathematical functions on these representations of real numbers. In most cases, the existing hardware/software systems are not adequate, for one important reason: in order to preserve the integrity of the interval, calculations involving the lower bound, or left endpoint of the interval, must be rounded downward; those involving the upper bound (right endpoint) must be rounded upward. No production system that we know of provides these roundings.

Thus, in designing this package, we were faced with two problems: definition of an appropriate system of real arithmetic approximation, and definition of the interval arithmetic operations based on that approximate real arithmetic system.

The design of the arithmetic primitives for the approximate real arithmetic was relatively straightforward; we used the algorithms given in [17]. The special functions posed more of a problem: straightforward evaluation of these functions can lead to unacceptably wide intervals. We decided to evaluate these functions in higher precision, and use information about the inherent error in the higher precision procedures before rounding the results in the proper direction to obtain the desired real approximation.

In order to preserve the desired degree of flexibility, we introduced the nonstandard data type EXTENDED to designate the higher-precision functions, and the nonstandard data type BPA (mnemonic for Best Possible Answer) to designate the approximation to real numbers used for the interval endpoints. The nonstandard data type INTERVAL was then declared to be a BPA array of length 2.

The BPA portion of the package was then written in terms of BPA and EXTENDED data types wherever possible. In only a few cases was it necessary to bind BPA to a standard data type in the package modules: such functions as the replacement operator obviously need to be bound to a standard data type to avoid recursive calls. There were sixteen (out of 47) modules in this portion of the package that seemed to require such binding.

We illustrate the implementation of the BPA portion of the package with the BPA square root routine. The COMMON block BPACOM is used to communicate the desired rounding option to the BPA routine and to communicate any errors (via the variable BPAFLT) to the calling routine. ACC is an integer variable which indicates the number of accurate digits in the EXTENDED routines. The statement R = ER implicitly invokes the conversion from EXTENDED to BPA, which includes addition or subtraction of an appropriate error bound and rounding in the specified direction.

```
      SUBROUTINE BPASQT(A, R)                                          BPA10320
C         R = SQRT(A)                                                  BPA10330
C         NO ERRORS POSSIBLE EXCEPT SQUARE ROOT OF A NEGATIVE NUMBER,  BPA10340
C         WHICH RESULTS IN BPAFLT BEING SET TO 18                      BPA10350
      COMMON /BPACOM/ OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA     BPA10360
      INTEGER OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA             BPA10370
      COMMON /BPACON/ ZRO, ONE, ONM, TWO, PIO2, PI, BPAMNB, BPAMXB     BPA10380
      BPA             ZRO, ONE, ONM, TWO, PIO2, PI, BPAMNB, BPAMXB     BPA10390
      COMMON /BPAACC/ IACC(24), LNACC, LGACC, SNHACC, TNHACC, EXPMXA,  BPA10400
     1                EXPMNA, FRACBD, MAXINT                           BPA10410
      INTEGER IACC                                                     BPA10420
      BPA LNACC, LGACC, SNHACC, TNHACC, EXPMXA, EXPMNA, FRACBD, MAXINT BPA10430
      BPA A, R                                                         BPA10440
      EXTENDED EA, ER                                                  BPA10450
      IF(A .LT. ZRO) GO TO 900                                         BPA10460
      EA = A                                                           BPA10470
      ER = SQRT(EA)                                                    BPA10480
      ACC = IACC(50)                                                   BPA10490
      R = ER                                                           BPA10500
      RETURN                                                           BPA10510
```

```
  900 BPAFLT = 18                                                    BPA10520
      RETURN                                                         BPA10530
      END                                                            BPA10540
```

The INTERVAL portion of the package was then written in terms of INTERVAL, BPA, and EXTENDED data types. In this portion of the package, only the BLOCK DATA module, the error-handling routine (which depends on internal representation to some extent) and a routine which unpacks Hollerith strings are system-dependent.

The following interval square root routine illustrates the implementation of the interval portion of the package. The COMMON block INTCOM is used for communication with the error-handling routine INTRAP. INTFLT indicates whether an error has occurred and, if so, its nature; ID is a code number designating the routine which is calling INTRAP; and TA, TB, and TR contain the arguments to and the result of the calling routine, respectively. Since the data types and numbers of arguments differ for different routines, INTRAP is given (in another COMMON block) the information necessary to decode the contents of TA, TB, and TR properly. Since we do not know a priori which of INTERVAL or EXTENDED may require the greatest amount of storage space, we resort to the EQUIVALENCE statement to ensure that enough storage is allocated for these three variables, regardless of the data type of the variables they may be required to contain. Note that before invoking the BPA square root routine (implicitly, twice; once for the right endpoint, or SUP, of the interval, and once for the left endpoint, or INF, of the interval), the variable OPTION is set to specify the desired directed rounding (RDU for upward directed rounding, and RDL for downward directed rounding). After the computation is complete, INTFLT is set to indicate which faults, if any, have occurred, the variable ID is set to 19 (the code for the square root routine), and INTRAP is called to process any error that may have occurred. Finally, the result is stored and a return to the calling program is executed.

```
      SUBROUTINE INTSQT(A, R)                                        INT18600
C         R = SQUARE ROOT OF A          (MONOTONE FUNCTION)          INT18610
C         THE ONLY FAULT WHICH CAN OCCUR IS THE ATTEMPT TO TAKE THE    INT18620
C         SQUARE ROOT OF A NEGATIVE NUMBER.  THIS WILL BE DETECTED     INT18630
C         IN BPASQT.                                                   INT18640
      COMMON /BPACOM/ OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA   INT18650
      INTEGER OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA           INT18660
      COMMON /INTCOM/ INTFLT, ID, TA, TB, TR                         INT18670
      INTEGER INTFLT, ID                                             INT18680
      INTERVAL TA, TB, TR                                            INT18690
      EXTENDED TAE, TBE, TRE                                         INT18700
      EQUIVALENCE (TA, TAE), (TB, TBE), (TR, TRE)                    INT18710
      INTERVAL A, R                                                  INT18720
      TA = A                                                         INT18730
      OPTION = RDU                                                   INT18740
      SUP(TR) = SQRT(SUP(TA))                                        INT18750
      OPTION = RDL                                                   INT18760
      INF(TR) = SQRT(INF(TA))                                        INT18770
      INTFLT = BPAFLT                                                INT18780
      ID = 19                                                        INT18790
      CALL INTRAP                                                    INT18800
      R = TR                                                         INT18810
      RETURN                                                         INT18820
      END                                                            INT18830
```

Appropriate description decks were then prepared for AUGMENT, binding the nonstandard types EXTENDED and BPA to representations in terms of standard data types (INTERVAL remains declared as a BPA array of length 2). The entire package was then processed using AUGMENT to extend these bindings throughout the package.

In order to adapt the resulting package to a different host environment, or different precision, or both, one writes the necessary primitive routines, adjusts the declarations in the description deck as necessary, and reprocesses the package with AUGMENT. That this procedure is effective is attested to by the relative ease with which this package was adapted for use on the IBM 370, Honeywell 600, DEC-10, PDP-11, and CDC Cyber systems.

5. Adaptations of the INTERVAL package:

We discuss two adaptations of the INTERVAL package: the first of these is the creation of a package to perform Triplex arithmetic, and the second is a package to perform interval arithmetic in multiple precision.

A. The TRIPLEX package:

Triplex is a variant of interval arithmetic in which a main, or "most probable", value is carried in addition to the endpoints. In West Germany, which could be regarded as the focal point of work in interval mathematics, triplex arithmetic is widely used in preference to ordinary interval arithmetic.

One of the visitors at the Mathematics Research Center was a Professor from the University of Karlsruhe, West Germany, who wanted to have access to triplex arithmetic during his appointment at MRC. We decided to see how difficult it would be to modify the INTERVAL package to perform triplex arithmetic.

The difference between triplex and interval arithmetic is conceptually quite simple: at the same time one computes an operation or function on the interval endpoints, using the interval mathematics formulas, one evaluates the same operation or function on the main values, using standard real arithmetic, rounding the results to the nearest machine number. Assuming that the real arithmetic is reasonable, the main value will always be a point in the interval given by the endpoints.

The task before us, then, was to add code to all of the interval routines to compute the main values, rename the modules of the package, adjust the formats to accommodate the third value, and, of course, change the representation of intervals to accommodate the main value.

The addition of the extra code was straightforward and ordinary; we simply added the appropriate lines of code to each routine to compute the main value. We should note, however, that this did not disturb the existing code, inasmuch as storage and retrieval of the endpoint values had already been defined not in terms of first and second array elements in the interval number, but rather in terms of the field functions INF and SUP respectively (AUGMENT allows the use of such field functions, even when the host FORTRAN compiler does not).

The modules were renamed by suitable use of a text-editing program on the INTERVAL file. Each INTERVAL routine began with the prefix INT; we wanted the triplex routines to have the prefix TPX. We simply instructed the editor to find all occurrences of INT which were not part of the words INTEGER, PRINT, etc. and change them to TPX.

The representation problem was handled simply by changing the word INTERVAL in the type declaration statements to TRIPLEX. No other changes were necessary in the bulk of the routines, since AUGMENT automatically extended the new binding throughout the package. The only places where hand coding was required were the few primitives (such as the replacement operator) where we needed to handle triples (or portions of them) rather than pairs. Again, some of these are obviously necessary to define basic operations.

The triplex square root routine below illustrates the types of changes to the interval package that were necessary to produce the triplex package:

```
      SUBROUTINE TPXSQT(A, R)                                          TPX20050
C         R = SQUARE ROOT OF A          (MONOTONE FUNCTION)            TPX20060
C         THE ONLY FAULT WHICH CAN OCCUR IS THE ATTEMPT TO TAKE THE      TPX20070
C         SQUARE ROOT OF A NEGATIVE NUMBER.  THIS WILL BE DETECTED       TPX20080
C         IN BPASQT.                                                     TPX20090
      COMMON /BPACOM/ OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA       TPX20100
      INTEGER OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA               TPX20110
      COMMON /TPXCOM/ TPXFLT, ID, TA, TB, TR                             TPX20120
      INTEGER TPXFLT, ID                                                 TPX20130
      TRIPLEX TA, TB, TR                                                 TPX20140
      EXTENDED TAE, TBE, TRE                                             TPX20150
      EQUIVALENCE (TA, TAE), (TB, TBE), (TR, TRE)                        TPX20160
      TRIPLEX A, R                                                       TPX20500
      TA = A                                                             TPX20180
      OPTION = RDU                                                       TPX20190
      SUP(TR) = SQRT(SUP(TA))                                            TPX20200
      OPTION = RDN                                                       TPX20210
      MAIN(TR) = SQRT(MAIN(TA))                                          TPX20220
      OPTION = RDL                                                       TPX20230
      INF(TR) = SQRT(INF(TA))                                            TPX20240
      IF(BPAFLT .NE. 0) TPXFLT = 66                                      TPX20250
      ID = 19                                                            TPX20260
      CALL TPXRAP                                                        TPX20270
      R = TR                                                             TPX20280
      RETURN                                                             TPX20290
      END                                                                TPX20300
```

Changing the I/O formats proved to be more time-consuming and error-prone than all of the rest of the changes put together. This, however, was a mechanical task.

The modification of the INTERVAL package to produce a TRIPLEX package was accomplished in little more than one week of elapsed time, documentation ([1]) excepted. This measures the time from when we first started modifications until we had obtained a completely successful test run, performed all of the housekeeping functions, and had the package ready for production use.

B. The Multiple Precision INTERVAL package:

One of the goals of the original design of the INTERVAL package was to allow for increased precision in cases where that was desired. When the multiple precision arithmetic package of Brent [2] became available, it was only natural to consider using that package as a basis for the multiple precision version of INTERVAL.

The first step in this process was to develop an AUGMENT interface for Brent"s package. This we did in collaboration with Brent, and the interface is written up in [3].

The steps in developing the multiple precision version of the interval package were:.

1. Determine the representation to be used for the real approximations. (Brent's package allows a great deal of flexibility in this regard.)

2. Write the primitive arithmetic operations, basing these on Brent's routines, but providing directed roundings (Brent's package does not have this capability).

3. Use Brent's package as the EXTENDED arithmetic package.

4. Write the necessary BPA primitives. The real format for BPA numbers will be closely related to the format of the Brent multiple precision numbers, but will have fewer digits and smaller exponent range.

5. Write an additional module which will set the necessary constants based on the run-time precision chosen for the BPA numbers. This will use features of Brent's package.

6. Rewrite the description decks as necessary

7. Reprocess the package with AUGMENT.

The development of the multiple precision version of the interval package took somewhat longer than the development of the TRIPLEX package, due primarily to the need to write the BPA primitives (these were the same for TRIPLEX as for INTERVAL) and the rather extensive format changes in the error handling routine that were necessary to accommodate the flexible representation afforded by Brent's package.

6. Other applications of AUGMENT

In the foregoing, we have illustrated the flexibility that may be gained by using abstract data types. We now consider some extensions of this concept, and some other applications of AUGMENT.

a. Recursive data type definitions: The versatility of AUGMENT may not be readily apparent from the above examples. AUGMENT allows data types to be defined in terms of one another, and this opens up some unique possibilities. The first author once used AUGMENT to aid in the writing of a program to sort information that was stored in the form of trees. Since the data was multiply-layered, the values of one tree's nodes would often be other trees. This problem was addressed by creating two data types: TREE and NODE. One field of a TREE was the root NODE, and one field of a node was a TREE. The development of the program using these new data types was straightforward.

b. Analytic differentiation of FORTRAN functions: This package, described in [10], allows one to obtain the Taylor Series expansion or the gradient of a function which can be expressed as a FORTRAN program. The technique is shown to be applicable to functions defined in terms of loops and conditional expressions as well as to the more conventional case of functions defined by a single FORTRAN statement.

c. Dynamic precision calculations: In certain types of applications, the precision required for the calculations is a function of the data. ANSI standard FORTRAN is not an appropriate language for such applications, since the precision of a given variable is both limited and predetermined. AUGMENT allows the definition of data types which are lists, however, and by using this feature, precision can be determined dynamically. One example of this technique is the dynamic precision package of Fullerton [8]; another is the SAC-1 system of Collins and others [4], which was developed without the aid of AUGMENT but will, if our information is correct, soon be interfaced with AUGMENT by another group.

c. Simulations: AUGMENT has been used to simulate one computer on another. The technique for doing this is straightforward; one defines a nonstandard data type which represents the simulated machine, and prepares a nonstandard package which copies the arithmetic characteristics and data formats of the target computer. This has been used to provide a means for algorithm development and word-length sensitivity analysis for airborne computers prior to their production.

e. Algorithm analysis: AUGMENT can be used to provide information such as operation counts in the running of programs or portions thereof. One simply defines a nonstandard data type which, in addition to performing the standard operation, increments a counter. Such statistics can be collected for entire programs quite easily by using the *CONVERT feature of AUGMENT, which allows one to convert every instance of one data type (standard or nonstandard) to another type; or, by inserting statements to print and reset counters in the source code, one can analyze specific portions of the program.

f. Image processing: The picture processing package described in [9] is probably one of the most unusual applications of AUGMENT we have yet seen. Various new operators allow the construction of composite pictures from smaller parts, and mathematical functions have even been defined on type PICTURE. For example, if P is type PICTURE, then SIN(P)**2 would yield a contour plot

of the intensity of P, and (SIN(P))**4 would yield a contour plot with sharper contours.

The above illustrations should serve to indicate that the role of AUGMENT in development of mathematical software is limited primarily by the user's imagination.

7. Conclusion:

We have indicated a number of ways in which the AUGMENT precompiler for FORTRAN can be and has been used to aid in the development of mathematical software. Other applications will undoubtedly be found for this precompiler, since it is both versatile and powerful.

Some of the emerging languages in research environments have some or perhaps all of the capabilities alluded to in this paper. For example, CLU [13], ALPHARD [16], and EUCLID [15] are capable of handling abstract data types. But at this writing, such languages are largely unavailable to most users.

Perhaps production languages of the future will include facilities implementing such techniques as abstract data type definition and dynamic precision determination; we hope so. But at the present time, AUGMENT seems to be one, if not the only, avenue open to the user who needs this sort of power and flexibility in his work.

REFERENCES

1. K. Boehmer and J. M. Yohe, A triplex arithmetic package for FORTRAN, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report (to appear).

2. Richard P. Brent, A FORTRAN multiple-precision arithmetic package, Assoc. Comput. Mach. Trans. Math. Software 4 (1978), 57-70.

3. Richard P. Brent, Judith A. Hooper, and J. M. Yohe, An AUGMENT interface for Brent's multiple precision arithmetic package, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report #1868, August, 1978.

4. G. E. Collins, The SAC-1 list processing system, The University of Wisconsin - Madison, Computer Sciences Department, Technical Report #129, July, 1971.

5. F. D. Crary, The Augment precompiler I: User information, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report #1469, December, 1974 (revised April, 1976).

6. F. D. Crary, The AUGMENT precompiler II: Technical Documentation, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report #1470, October, 1975.

7. F. D. Crary, A versatile precompiler for nonstandard arithmetics, Assoc. Comput. Mach. Trans. Math. Software (to appear).

8. W. Fullerton, Absolutely portable special function routines, Talk presented at the NSF/ERDA Workshop on Portability of Numerical Software, June 21-23, 1976.

9. W. Fullerton, Private communication.

10. G. Kedem, Automatic differentiation of computer programs, Assoc. Comput. Mach. Trans. Math. Software (to appear).

11. U. Kulisch, An axiomatic approach to rounded computations, Numer. Math. 18 (1971), 1-17.

12. T. D. Ladner and J. M. Yohe, An interval arithmetic package for the UNIVAC 1108, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report #1055, May, 1970.

13. B. H. Liskov and S. N. Zilles, Programming with abstract data types, Proceedings of a Symposium on Very High Level Languages, in SIGPLAN Notices 9 (1974), 50-59.

14. Ramon E. Moore, Interval Analysis, Prentice - Hall, Inc., Englewood Cliffs, NJ, 1966.

15. G. Popek, J. J. Horning, B. W. Lampson, R. London, and J. Mitchell, Notes on the design of EUCLID, Proceedings of the ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 28-30, 1978.

16. W. A. Wulf, ALPHARD: toward a language to support structured programs, Carnegie - Mellon University, Technical Report, April, 1974.

17. J. M. Yohe, Roundings in floating-point arithmetic, IEEE Trans. Computers C-22 (1973), 577-586.

18. J. M. Yohe, The INTERVAL arithmetic package, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report #1755, June, 1977 (revised September, 1977).

19. J. M. Yohe, Software for interval arithmetic: a reasonably portable package, Assoc. Comput. Mach. Trans. Math. Software (to appear).

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 1892
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIAL PURPOSE ARITHMETIC PACKAGES

5. TYPE OF REPORT & PERIOD COVERED: Summary Report - no specific reporting period

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): F. D. Crary and J. M. Yohe

8. CONTRACT OR GRANT NUMBER(s): DAAG29-75-C-0024

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin
610 Walnut Street
Madison, Wisconsin 53706

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: #8 Computer Science

11. CONTROLLING OFFICE NAME AND ADDRESS:
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, North Carolina 27709

12. REPORT DATE: October 1978

13. NUMBER OF PAGES: 17

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Precompiler
Nonstandard arithmetic packages
Portable software
Software development
Software modification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We discuss the use of a FORTRAN precompiler in the development of packages for nonstandard arithmetics. In particular, the use of the FORTRAN precompiler, AUGMENT, renders the source code more lucid, reduces the number of lines of code in a nonstandard arithmetic package, facilitates modification, and ameliorates the problems of transporting such a package to another host system.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)