22
5/2/79
1
24/80 NTIS

UP-32

UCID-18089

Lawrence Livermore Laboratory

MASTER

COMPUTER LANGUAGE EVALUATION FOR NTF SCDS

Roger E. Anderson, Patrick R. McGoldrick, and Robert H. Wyman

April 11, 1979

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Computer Language Evaluation for Mirror Fusion Test Facility

Supervisory Control and Diagnostics System

Roger E. Anderson, Patrick R. McGoldrick, and Robert H. Wyman

Abstract

The computer languages available for the systems and application implementation on the Supervisory Control and Diagnostics System (SCDS) for the Mirror Fusion Test Facility (MFTF) have been surveyed and evaluated. Four language processors, CAL (Common Assembly Language), Extended FORTRAN, CORAL 66, and Sequential Pascal (SPASCAL, a subset of Concurrent Pascal [CPASCAL]) are commercially available for the Interdata 7/32 and 8/32 computers that constitute the SCDS. Of these, the Sequential Pascal available from Kansas State University (KSU) appears best for the job in terms of minimizing the implementation time, debugging time, and maintenance time. This improvement in programming productivity is due to the availability of a high-level, block-structured language that includes many compile-time and run-time checks to detect errors. In addition, the advanced data-types in the language allow easy description of the program variables.

I Background

A network of nine minicomputers form the Supervisory Control and Diagnostics System (SCDS), they communicate via multiported shared memory. Their function is to facilitate automatic and manual experiment control, data acquisition, data reduction, and data analysis for the Mirror Fusion Test Facility (MFTF) from a central control room.

The SCDS will enable researchers to conduct experiments on the MFTF plasma physics experimental system as frequently as every five minutes. Even with such short intervals, the SCDS will allow experimenters to interactively set up and perform

NOTICE
This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

ADDITIONAL COPIES OF THIS REPORT ARE AVAILABLE FROM THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT A SPECIAL RATE. SEE NTIS PUBLICATIONS FOR DETAILS.

344

an experiment, acquire the data, and interactively analyze the data prior to succeeding experiments. A considerable increase in system usefulness over current control systems arises from the high degree of interactivity and the rapid data access and processing provided by the SCDS.

The desired performance specification for the experimental facility will, in most part, determine the size and speed of the computer hardware required for the SCDS. This specification is influenced, to some extent, by an estimate of the amount of software required to implement the operation. Additionally, the magnitude of the software implementation effort must not be underestimated: Recent conferences report software costs in many specialized applications reaching ten times the hardware costs. However, proper software design and proper implementation can affect the software development effort by as much as order of magnitude; thus, any improvement in the efficiency of software development must be considered important.

II Language Effects

Selection of an implementation language has effects in four areas: design, implementation, debugging, and maintenance. If the language has a degree of abstraction that begins to approach that of the problem itself, then it becomes easy to describe and to write a concise, accurate description of both the problem and its solution. In fact, the language choice can make the design easy if the language was designed for the problem to be solved.

Regrettably, few languages have been written for specific problems, and none are specifically tailored to control experimental physics machines. Accordingly, a general-purpose language with many advanced data constructs that are applicable to the problem should be employed. If the language is easy to learn and use, it will allow rapid and accurate coding. If the language has a high degree of abstraction, it will make the implementation easy. Thus, it should be as close to the language of experiment designers and implementors as possible.

Through the use of modern structured programming techniques, the program can be built modularly by more than one implementor, while remaining consistent with the original specification and design. If the compiler specification requires data typing and range checking, accurate output can be obtained without an inordinate amount of effort on the part of the implementer. The output of a modern, high-level compiler should execute if it compiles without error. In this way, the debugging effort is reduced to a minimum. Debugging will consist of checking logical flow and not in looking for transcription errors and misspelled variables.

Finally, the choice of language can affect the maintainability of a program. Program maintenance does not consist of replacing parts that are worn out, but of correcting or replacing the parts that contain latent errors that appear after the system is exercised more fully. Language choice is also important when extensions or modifications to programs occur.

The number of combinations and permutations of input data and information to a computer program is so great as to preclude exhaustive testing. As a consequence, design deficiencies result in latent program bugs that appear in program use. If language is modular and easily understood by a nonauthor, its maintainability is greatly enhanced. Differently stated, such a program is corrected in a minimum of time by almost any programmer. These features also apply when design changes are made because of new experimental tasks or new insights.

The following features are desirable in a language used to implement scientific data acquisition, data reduction, and experiment control:

- High level. The language should have a sufficient degree of abstraction to directly express the control constructs and data structures without resort to "tricks" or highly specialized operations.
- Completeness. The language should be both defined and supported (maintained).
- Modular, block-structured. The language should be sufficiently block-structured that source modules are easily written and put together. No module should contain more operations than can be readily understood by the programmer as a single, higher level operation.
- Efficient. The language should be efficient in memory utilization, execution speed, and compilation speed. In some cases, these may be mutually exclusive, but memory utilization and execution speed are the most important.
- Library capability. The language should be capable of using pre-existing library routines to take advantage of previously defined and developed procedures.
- Error checking and diagnostics. The language should provide for extensive internal checking to prevent errors, both at compile time and at run time. When errors are found, concise, clear diagnostics should be given to facilitate correction. Errors found at run time should relate back to the source line number, not only to the memory location where the error occurred.
- Capability. For systems work, the language should have the capability of using all hardware features of the computer. These should be implementable in a manner that does not produce side effects or detract from block-structured design.

- Understandable. The language should be easily understood without the use of extensive study or research. The language should promote good documentation and be as close to self-documenting as possible. This also promotes ease in learning the language for the novice.
- Reentrant. The language should be implemented in a manner that produces reentrant code. This allows commonly used procedures to be stored in reentrant libraries and executed without requiring individual copies or additional memory space.

III Languages Evaluated for Interdata 32-bit Computers

Common Assembly Language (CAL) - CAL is the lowest level language processor available from Interdata. It is an assembly language, i.e., there exists a one-to-one correspondence between language statements and machine instructions. It does contain macro expansion capabilities and structured data types. Macros can provide language extensions, while structured data types allow data-structure extensions. CAL can produce reentrant code, provided the programmer follows three rules:

- . The code must not modify itself.
- . The code must obtain arguments via registers.
- . The code must not store variables in its own code space.

CAL is capable of expressing anything the machine can do. However, it still requires more lines of source code than any of the language processors.

Extended FORTRAN - FORTRAN is the most widely used high-level language for scientific computation. It was one of the earliest language processors developed and it evolved as a means of converting scientific algebraic equations into computer programs. It has been extended to include multiple data types such as

integer, real, complex, Boolean, array, and multiple-precision. The currently available compiler does not include the modern structured-control constructs. FORTRAN does little type-checking and allows direct interchange of data types via EQUIVALANCE statements. Modern compilers are fast, they produce code that is nearly as small as theoretically possible and that still executes efficiently. Probably the greatest assets of FORTRAN are that it is widely known and that extensive standard libraries are available for it. The Interdata implementation of FORTRAN VI does not produce reentrant code.

CORAL 66 - CORAL 66 is a compiled language that is based on the algebraic language ALGOL 60. ALGOL 60 was the second programming language development to create an algebraic language. ALGOL had some very good features, but it never received wide acceptance--probably because its poor input-output capabilities. CORAL is a block-structured language with more modern control structures and local variables than its predecessor. It has additional data types and includes type-checking to help prevent errors. The Interdata version has additional data types to make use of hardware instruction facilities on the computer. Even though it has library capabilities, it still is very limited in its input-output features. This may be because CORAL is a United Kingdom Standard and must be capable of running on a wide variety of computers. The language has been developed to be capable of system programming, and the use of inline assembly code allows all computer features to be used.

CORAL 66 does not produce reentrant code, even though block-structured languages are prime candidates. The variable scoping rules imply variable creation on a stack. Access can then be made via a register pointer to the stack which assures reentrancy. However, the Interdata implementation puts the variables in an impure segment that is not permitted in a reentrant library.

Sequential Pascal - Sequential Pascal (SPASCAL) is a subset of Concurrent Pascal (CPASCAL). It is that part of CPASCAL that does not require concurrent (simultaneous) processing or the signaling and synchronization associated with them. CPASCAL was designed by Per Brinch-Hansen to put concurrent processes into the concept of the language Pascal that was developed by Niklaus Wirth. Pascal was developed by Wirth as a heavily typed, block-structured language with modern control structures. It was originally designed as a language to express algorithms and was based on ALGOL and ALGOL W.

The rich data types that are defined or definable in Pascal and the strong type-checking have extended its use considerably. Key words express algorithms in an English-like manner and lead to easy learning and understanding. Strong type-checking and error checking lead to early discovery of blunders and transcription errors, and thus minimize debugging time.

The SPASCAL subset of CPASCAL retains most of the features of Pascal. A few special characters have been changed, but this is a minor variation. Most significantly, Sequential Pascal does not allow nesting of PROCEDURE definitions; this limits the implementor's flexibility in variable scoping.

SPASCAL produces truly reentrant code. Both local and global variables are referenced via pointers that are kept in registers. Accesses are made to main memory via indexed addressing off these pointers, which results in a pure procedure.

The greatest limitation of Sequential Pascal is its limited input-output capabilities. However, the Kansas State University (KSU) implementation has several extensions that reduce most of these limitations. The most important of these are the use of EXTERNAL PROCEDURES and EXTERNAL FORTRAN PROCEDURES. External procedures

give the capability of assembly language procedures, so that the language can express anything the computer can do. However, the assembly language procedure is buffered from the programmer since it is obtained from a library. Thus, the procedure can be tested and verified before use. External FORTRAN procedures give the programmer access to existing FORTRAN libraries. This takes advantage of the wealth of existing, tested procedures. This is a bridge to extensive current facilities until eventual replacement by libraries written in PASCAL.

Present deficiencies observed in SPASCAL and in Pascal are the lack of a standard language definition and the lack of STRUCTURED DATA TYPES to predefine constants. Both of these problems are recognized, and a standards effort has been started. Many extensions have been suggested. It should be noted that wide interest in the KSU implementation of Sequential Pascal is providing the support needed to make it a reasonably complete and reliable language implementation.

IV Evaluation and Comparison

All of the language processors mentioned in Section III are commercially available. All have routine maintenance and custom modification available from an external source. All have been operated on the SCDS system. Programs have been compiled and executed. All are capable of describing the application and systems problems for the Interdata computer system.

Although these languages are presented here in order of increasing degree of abstractness (this is somewhat dependent on application), the best choice for the MFTF system is the KSU Sequential Pascal. However, the specific comparative strengths and weaknesses of each are given below.

The CAL assembly language is the lowest level processor for the system. It does have macro and structured data-typing capability to enhance its use, but any type-checking would need to be explicitly included in the program. Because the probability of error is proportional to the amount of source code (and is apparently independent of the language), it would be most error-prone. In addition, CAL has no inherent block structure, hence structured programming techniques would put the entire enforcement burden on the programmer. However, assembly language remains available for the other three language processors to expand their capability.

The Extended FORTRAN processor is a considerable improvement over assembly language because it is a compiler language. The number of source lines of code is greatly reduced, with a corresponding proportionate decrease in errors. Its greatest shortcoming is that it is not a structured language, hence enforcing structured programming techniques is again left to the programmer. Lack of user-definable data types and type-checking allows the programmer to make inadvertent errors and errors of transcription. Finally, the availability of EQUIVALENCE statements and GOTO statements may easily lead to obscure and muddy code.

The CORAL 66 language is a good early attempt at a structured language, but it lacks the user-definable data types to allow increased abstraction. The poor input/output facilities would require a large amount of programmer effort to make the system readily useable. It includes a relatively small amount of compiler data-checking and even performs implicit-type-conversion in assignment statements. The reserved-word format is awkward, and it does not lead to easily readable source code. Some of the control constructs are also a bit awkward, and the address manipulation is not as convenient as pointers in Pascal. In general, it is a reasonable approach, but not the best available.

The Sequential Pascal compiler has the best features available for the Interdata system. However, there was concern that its source is a noncommercial organization. Subsequent installation and test showed it to be a viable product with few bugs. KSU interaction and response to inquiry or change has been as good or better than that with commercial vendors.

The greatest weaknesses with the language implementation are the lack of good input/output facilities, lack of structured constants, and the lack of a defined standard. However, the language extension to external procedures circumvents many of these problems. In addition, the KSU implementation of PREFIX (a predefined set of constants, types, and procedures that are included into the source code) make most of the interface to the operating system transparent to the user. The lack of structured constants is a problem that can be solved with an extension to the language. In addition, it is the only compiler evaluated that produces reentrant code.

Comparison of the object code execution rate shows the SPASCAL code to be equivalent to FORTRAN code, when optimization is included and run-time error checking is deleted. Execution time is similar, with SPASCAL 4% faster than FORTRAN without run-time error-checking, and FORTRAN 6% faster than SPASCAL with run-time error-checking. Note, however, that FORTRAN has no run-time error-checking. It seems quite reasonable to pay 6% for this optional feature. (See appendix for more comparison details.)

V Conclusion

Four currently available language processors for the Interdata 32-bit computers have been investigated. For programming efficiency in the MFTF SCDS application, the Sequential Pascal compiler from KSU appears to be the best language processor

for the task, although the task could be performed with any of the available processors. This language should provide a functional system with the least programming and debugging efforts, and in the most easily maintainable manner.

Bibliography

- Assembly Language Programming (CAL), Pub. No. B29-640, Interdata, Inc., Oceanport, NJ, 1977.
- Fortran VI Reference Manual, Pub. No. 29-240, Interdata, Inc., Oceanport, NJ, 1977.
- CORAL 66 Reference Manual, Pub. No. 29-587 R02, Interdata, Inc., Oceanport, NJ, 1978.
- Jensen, K. and Wirth, N., Pascal User Manual and Report, 2nd Ed. (Springer-Verlag, NY, 1978).
- Brinch-Hansen, P., and Hartmann, A. C., Sequential Pascal Report, Information Science, California Institute of Technology, July, 1975.
- Brinch-Hansen, P., The Architecture of Concurrent Programs, (Prentice-Hall, Inc., Engelwood Cliffs, NJ, 1977).
- Young, R., Pascal/32 Language Definition, Technical Report, Dept. of Computer Science, Kansas State Univ., Manhattan, KA, 1978.
- Wallentine, V. and McBride, R., Concurrent Pascal - A Tutorial, Technical Report CS-76-17, Dept. of Computer Science, Kansas State Univ., Manhattan, KA, 1976.
- Hankley, William J. and Rawlinson, Jerry D., Sequential Pascal: Supplement for FORTRAN Programmers, Technical Report CS 76-18, Dept. of Computer Science, Kansas State Univ., Manhattan, KA, 1977.
- Ball, Michael S., "Unpublished notes on Code Generation and Optimization of SPASCAL for Interdata Machines," Naval Oceanographic Systems Lab., San Diego, CA.

APPENDIX

Whetstone^a Results

	Code size (bytes)	Const size (bytes)	Total size (bytes)	Whetstone number (10 ³ Whetstone inst/sec)
SPASCAL ^b	3720	424	4144	418
SPASCAL ^b (optimized)	3556	304	3860	429
SPASCAL ^b (optimized, no run-time checks)	3438	304	3742	473
FORTRAN VI	-	-	3754	454

^aCurnow, H. J. and Wichmann, B. A., "A Synthetic Benchmark,"
The Computer Journal, Vol. 19, No. 1.

^bSPASCAL programs called FORTRAN library procedures for
functions (SIN, COS, etc.) and to perform output of reals.

Note: Sizes do not include run-time library procedures because
the same run-time libraries were used for both
SPASCAL and FORTRAN VI.