

SAFEGUARD Data-Processing System:

CENTRAN—A Case History in Extendible Language Design

By B. N. DICKMAN

(Manuscript received January 3, 1975)

The history of the design and implementation of CENTRAN, an extendible language, is presented as an example to language designers. The history is viewed in the context of four groups of factors: environmental issues, general design issues, specific design issues, and implementation issues. The paper concludes with an evaluation of the design decisions that were made.

I. INTRODUCTION

There are many papers about the syntax and semantics of computer languages. There are some papers about the compilers for these languages. But there are few papers describing how and why a language was designed and how it was implemented. In presenting the design history of CENTRAN,* we attempt to provide a method that language designers may apply to improve the writing of software.

Previous attempts at building a language for SAFEGUARD either attempted to provide a shell[†] language like PL/I (NICOL), the entirety of which could be implemented or understood only with extreme difficulty, or attempted to provide a complete syntactical uniformity of the machine language structure, like PL360. The attempt to provide syntactical uniformity failed because requisite hardware uniformity does not, in fact, exist. At the assembly language level, the syntax of a language cannot be more uniform than the structure of the object machine.

CENTRAN can be viewed as an extendible language in which several levels of language features exist. At the lowest level, CENTRAN is the assembly language. At the next level, CENTRAN provides a uniformity for the machine by completing incomplete data paths and by providing

* CENTRAN and ETC are different names for the same language (see Ref. 1).

[†] A shell language attempts to provide all the features users would ever want.

uniform register usage. At this level, CENTRAN is still almost one-to-one with the machine code, but provides a more concise syntax for the machine operations by means of, for example, polymorphic operators. At the next level, machine dependence may still exist in the form of hardware register references, but CENTRAN functions as a true compiler. At the highest level of use, CENTRAN programs can be as machine independent as those written in PL/1.

The extended language of CENTRAN approximates PL/1 in control structure and FORTRAN in data structure. In addition to the control structure of PL/1, CENTRAN has CASE, BREAK, and ITERATE statements. BREAK allows a program to exit a DO loop or group gracefully (without use of a GO TO statement); ITERATE causes the next iteration of a DO loop or group to begin. The data structure is similar to that of FORTRAN except that there are based variables, simple structures, and partial word variables. The base language has been described in Ref. 1.

II. LANGUAGE DESIGN PROCESS

The many factors which control the design and implementation of a language can generally be classified into four groups, the designer having increasingly greater control over the resolution of the factors in the later groups.

The four groups are: environmental factors (external resources and constraints), general design issues (decisions to be made based directly on environmental factors), specific design issues (decisions of a topical nature to be made based on the resolution of general design issues), and implementation issues. The resolution of the issues posed in earlier groups are factors in the resolution of issues in the later groups.

2.1 Environmental factors

This group consists of factors over which one generally has little or no control.

2.1.1 Necessity for a new language

First, there is the basic presumption that yet another language is necessary. The need for a new language hopefully arises from external considerations, rather than out of some inner need of the designer or as a result of the "not invented here" syndrome. There must be good justification for designing a new language rather than choosing all or part of an existing language.

It was clearly desirable to write at least some of the SAFEGUARD software in a language higher than machine language. There were many cases in which the possible inefficiencies in code generated by

a compiler could be tolerated. There were also many cases in which it was desirable to produce working programs inexpensively, regardless of the cost in running time and core, e.g., drivers and other test programs. Furthermore, if assured of good programming leverage (object-to-source-code ratio greater than one) from a language, and concise generated code from its compiler, it would be desirable to write all software in that language.

In the SAFEGUARD project, the compiler for the existing high-level language, NICOL, was unstable, and it was felt advisable to develop a language intermediate to NICOL and the assembler language as insurance. Selling CENTRAN as an "intermediate level" language (rather than a high-level language) avoided the presumption of NICOL's demise and avoided promising too much prematurely.

2.1.2 Manpower and implementation schedule

Two rigid constraints on the implementation of a language in an industrial environment are the manpower available and the implementation schedule: PL/1 cannot be implemented on a FORTRAN budget. Furthermore, the feasibility of using a high-level language must be proven before a commitment will be made to the implementation. A working compiler, with programs written in the language, is the most persuasive proof of feasibility.

For CENTRAN, the requirement existed to produce something useful within six months because the project was well under way and user software development could not wait. Only two full-time people and one person half time were available for design and implementation. There was no promise of increased manpower or lengthened schedule. Only one of these people had previously designed and implemented a compiler. It was necessary that the structure of the compiler be clean enough and simple enough for the available manpower to implement. The extendibility features of CENTRAN played a role here in assuring that the basic structure of the compiler could be implemented in a short time. Using the SWAP² macro facilities to write the compiler also contributed to the quick implementation of the language.

Within three months, a skeleton compiler was written that was able to successfully compile sample programs with which to show the feasibility of CENTRAN. A computer listing can be powerful magic, even among the initiated, and compiler development support was soon forthcoming.

2.1.3 Hardware

The hardware on which the programs are to be run is more of a constraint in the design of a language than is usually realized. Going

from one generation of hardware to another has revealed machine dependencies and influences in language design. It has often been said that there should be more feedback to hardware design from language design, but until the state of software technology reaches that of hardware technology, hardware will be a fixed factor in language design.

The language designer has the final word on how the hardware appears to the user. He has the satisfaction of knowing that one purpose of a computer language is to compensate for "errors" in hardware design, such as to make the machine seem more uniform in structure than it actually is or to make explicit by syntactic equivalence the classes of machine operations. For example, the designer may use "+" to add a constant to a variable as well as to add two variables, even though the "+" may be implemented as two different machine operations.

The SAFEGUARD Central Logic and Control (CLC) computer was the target machine for CENTRAN. At a low level, CENTRAN supplied a uniformity to the CLC instruction set that did not in fact exist. For example, there were no machine operations to move data from certain registers to others without first moving the data to an intermediate register. CENTRAN "completes" incomplete data paths by generating the appropriate code. Of course, at the highest level of CENTRAN use, no references to hardware registers are necessary.

2.1.4 Software environment

The degree to which the software environment (e.g., loaders, binders, and operating system) is a fixed factor may affect the mechanics of program production and perhaps even the design of the language itself.

At the time CENTRAN was being designed, a large body of support software already existed. It was tedious matter to reassemble all SNX programs each time the object module format changed, and so it was decided that CENTRAN would conform to SNX object module format. As a result, certain desirable language features could not be included (e.g., multiple location counters) because they could not be represented in the object module.

2.1.5 User population

The two attributes of the intended user population, programmer proficiency and programmer background, affect the design of the language. For CENTRAN, the user population (in addition to Bell Laboratories people) consists of several subcontractors. The programmers exhibit a wide range of ability and experience.

Programmers have an emotional investment in the first language they learn; it is difficult to teach a programmer a second language. On the SAFEGUARD project, most of the experienced programmers were assembly language programmers and had a strong bias toward writing in machine code. This phenomenon has been noted in a more general context by Weinberg.³ CENTRAN attempted to make the transition to a high-level language more palatable by keeping the machine accessible if so desired. The assembly language, SNX, is actually a proper subset of CENTRAN.

CENTRAN may have made the transition to a high-level language too easy—some programmers still think in machine language when organizing their programs, leading to a potential rigidity of structure and lack of language leverage.

2.2 General design issues

While the environmental factors generally are not under the control of the language designer, some degree of design creativity can be expressed in the resolution of the general design issues. These issues are: whether to create a new language or adapt an existing one, what the degree of machine independence and the language level are to be, how important ease of learning and ease of use are, whether the language should in some sense be “complete,” and whether the language design should express present technology or the state of the art.

2.2.1 Creation of a new language or adaptation of an existing language

In determining whether to create a new language or adapt an existing one, the designer must beware of contracting either or both of two diseases: the “not-invented here” and the “it’s-more-fun-to-design-my-own-language” syndromes.

In the case of a language for SAFEGUARD, the language compiler for NICOL 3 was found to be nonviable. An alternative seriously considered was to code, debug, and unit-test all programs in PL/1 using IBM computers and then to hand-compile the programs into SNX so that they could run on the CLC. This may well have been the course taken if CENTRAN had not been produced on schedule.

There was, however, an “almost existing” low-level language, the CLC assembly language SNX. It included the SWAP macro facilities, possibly the most sophisticated in existence (see Ref. 2), most of the interfaces with the operating system, and an object module generator that almost met requirements. By building on the existing SNX assembler, the designer and implementers gained a certain built-in compatibility with existing SNX SAFEGUARD programs, familiarity with

the format, and most important, because of manpower and development-time constraints, free maintenance. However, the approach lost block structure (since the assembler did not have it), efficiency with respect to compile time (since the macro facility is completely interpretive), and control over lexical analysis.

Thus, an existing assembler was used as the base language for an extendible compiler. This allowed maximum use of existing software.

2.2.2 Degree of machine independence and language level

The two concepts, language level and machine independence, although related, are not equivalent. The language level is best described in terms of the degree of clarity and conciseness possible. Machine independence is usually defined in terms of the degree of portability of a program written in the language, i.e., how easily a program may be transferred from one machine to another. A language may be very machine dependent and of a high level.

Since there were no plans for successors to the SAFEGUARD system, machine independence was not a major factor in the design of CENTRAN. The level of the language, however, was a factor. As was mentioned in the discussion of the environmental factors, at the time CENTRAN was being designed there was a perceived need for an intermediate-level language. At the same time, it was apparent that certain high-level language features would soon be needed. CENTRAN's extendible design made it feasible to satisfy both of these requirements.

2.2.3 Ease of use and ease of maintenance

A language may be constructed with consistency, uniformity, and good debugging features, all of which makes it easy to learn the language and to write programs. Languages of this sort are ALGOL 68 and SNOBOL 4.

Program maintenance is aided if the purpose of a program written in the language is easy to comprehend, even though the syntax and semantics are nonuniform. Languages of this sort are PL/1 and FORTRAN.

Are ease of use and ease of maintenance related? Programs may be easy to write but incomprehensible once written, e.g., programs written in PAL, QED, and APL. Programs may be difficult to write but easy to read once written and debugged (e.g., FORTRAN, PL/1, and COBOL). Programs may be difficult to write and difficult to maintain (e.g., machine language programs and IBM JCL).

Another aspect of ease of maintenance that should be considered in language design involves binding time: binding addresses to variables and programs, disc locations to files, and generated code to source statements. In general, the later binding occurs, the easier

programs are to maintain. "Patching" is usually easier, as is having independent compilation of subroutines and independent order of compilation. Late binding does, however, increase the cost in link, load, or run time. In CENTRAN, since the object module format was fixed, the language designer had no control over when binding was to occur.

2.2.4 Present technology or state of the art

A decision is made, unfortunately often only implicitly, as to whether the language is to advance the state of the art in language design and implementation or is to represent what present technology can accomplish.

Why design a language if it is not state of the art? Often, there is no need to invent a new language merely to fulfill user needs for a special-purpose language. It may be sufficient to select those features which are needed from existing languages. In a production environment, due to schedule constraints and caution on the part of management, state-of-the-art language may be considered undesirable. A state-of-the-art language and compiler represent more of a design investment and more of a risk.

CENTRAN was never sold as state of the art. However, CENTRAN still had to be implemented as an extendible compiler so that incremental implementation would be feasible. There was no time to do anything else.

Extendibility allowed the circumvention of general design issues by delaying their resolution, possibly indefinitely. If the language is not sufficiently machine independent, extend it to a machine-independent level and code only at that level. Completeness? Extend it as necessary. Efficiency? Start from the machine language; what could be more efficient?

Except for the extendibility features and treatment of machine registers, the extended CENTRAN language is not state of the art. Of course, the extendibility features of the base language, register allocation, and subroutine interface primitives may be considered state of the art, but the average user does not use these features.

2.3 Specific design issues

Specific design issues include: control structures, data structures, program-development features (e.g., tables of variables and attributes, listing format control), and extendibility features (e.g., programmer-defined subroutines, functions, macros, and data types) to be incorporated into the language.

The model chosen for the extended language for CENTRAN was PL/1. It is believed that this was the best decision, provided that a new language could not be designed from scratch. However, there are several reasons why ALGOL 68 (see Ref. 4) would be a better choice as the model. (It should be noted that the ALGOL 68 Report was not available when CENTRAN was being designed.) Perhaps the most important reason is that "expression languages" (in which most statements, as well as what are commonly thought of as expressions, return values and can occur anywhere expressions can occur) can allow the programmer to express himself in a degree of clarity not possible in other languages. Furthermore, an expression language is especially desirable for efficiency and clarity if the compiler does not do any common subexpression analysis, and the language gives the programmer access to hardware registers for the purpose of improving efficiency.

In particular, one of the results of modeling the extended language on ALGOL 68 would have been the choice of distinct representations for equality comparison and assignment. Then assignment could return a value, facilitating, for example, the use of register variables.

System macros (a set of utility macros used, for example, to interface with the operating system) were SNX-style and should have been CENTRAN-style. While implementation of a CENTRAN representation for all system macros was vetoed as not worth the effort, program bugs were induced by syntactical and semantic nonuniformities.

No thought was given in language design to program patching. Patching on the CLC was necessary, primarily due to the logistic problems involved in recompiling programs on the IBM machine and transporting them to the CLC. Little thought was given to data reduction because there were no requirements specified at the time. Requirements for patching and data reduction should have been considered. We pay the piper: for patching, one must patch in SNX or recompile; for data reduction, few symbolic data structures are allowed.

On the positive side, in addition to permitting the compiler to be built quickly, the extendibility mechanism confers additional advantages. The extended language was planned so that extensions could be made to semantics rather than syntax. Some documentation for the extension is free, since description for new syntax is not required. Some user education is free when new semantics can be associated with old syntax.

Extending a language is trivial if all extensions consist of new syntax not meant to interact with old syntax. That is how some language designers and users of extendible languages extend a language. The difficulty is to maintain uniformity, especially when the extension is

not orthogonal to the old language. The classic problem here is to add complex arithmetic to a language, extending the semantics of the existing arithmetic operators, rather than creating new ones. Reference 1 describes how this may be accomplished in CENTRAN.

2.4 Implementation issues

2.4.1 Compiler speed and degree of optimization

There always seems to be a trade-off between the speed of a compiler and the optimality of the code produced. In an academic environment, where there are many student jobs, there are many compilations and few executions. In that case, a fast compiler designed without regard to object code efficiency is acceptable. In a production environment, presumably little time is spent in compilation in comparison to the execution time for production programs. Here, highly optimized code is desired.

One way to circumvent making the trade-off is to write two compilers, but this introduces obvious problems, not the least of which is potential incompatible language implementations.

CENTRAN is a slow compiler. This is due primarily to its interpretive nature. While some performance improvements were made after the compiler was written, stability requirements outweighed compilation speed requirements, and extensive improvements have not been made. The lesson learned is that if a program works, it is not likely to be re-written just to improve its efficiency.

The design goal for CENTRAN was to optimize on the statement level only, producing the best code possible for statements such as " $a = b$ operation c ," where a , b , and c are simple variables. Sufficient manpower to produce a global optimizer was not available. Users would rather have more features in CENTRAN than have a globally optimized program. The local optimization design goal of CENTRAN was achieved, leaving global optimization to the user (aided by effective counseling).

Since the expression parser produced nonoptimal code, users were warned against using complex expressions if they had severe running time or space constraints. This was done also to protect the implementers against the possible wrath of users complaining about inefficient code. However, the lack of optimization of code produced by the expression parser was oversold, and programmers get much less leverage from CENTRAN than they could.

2.4.2 Compiler structure

After the questions of degree of optimization and speed of the compiler are resolved, there remains an issue that is the primary

concern of the designer: compiler structure. Related to the structure of the compiler is the question, "In what language should the compiler be written?"

Several alternatives were considered in the implementation of CENTRAN. First, as indicated earlier in the discussion of environmental constraints, it certainly was not feasible to create a language completely independent of SNX. There were no resources to implement a new output-module generator, interfaces to the operating system, and machine-operations listing. The compiler at least had to be assembler-ended; the output of CENTRAN had to be an input to the SNX assembler. The question then became that of the degree of interaction between the compiler and the assembler.

Why was CENTRAN not implemented as a preprocessor to or a co-routine with SNX? The answer is that it was not clear at the time how the interface could be achieved. It still is not clear that this can be done successfully. The assembler was not designed to interface externally with a language processor. Other problems to be considered include the possibility of duplicate symbol tables, duplicate language processing, the loss of the macro facility, and the introduction of nonuniformities.

A compiler-compiler was not used to implement CENTRAN because there was none available and creating one would have meant maintaining two languages.

The method of implementation of CENTRAN consists of a combination of recursive descent and precedence tables. The arithmetic, logical, and relational expression parsers are driven by precedence tables; everything else is recursive descent with a vengeance. All the statements generated by the compiler (even those generated by the table-driven parser in the expansion of a CENTRAN statement to machine code) are legal CENTRAN source statements. There is no "canonical" intermediate-level language inaccessible to the user of the extended language. Each machine operation is (textually) generated in only one place. All CENTRAN code generating statements are eventually expanded into a set of CENTRAN statements, each generating exactly one machine instruction.

III. LANGUAGE USAGE

3.1 Who is using the language?

CENTRAN is the official language for the SAFEGUARD project. Except for programs which had been written in assembly language before the availability of CENTRAN (parts of the CLC operating system), all SAFEGUARD programming is done in CENTRAN. Programmers may not use machine language without management approval. No cases are

known where it was necessary to “drop down” into machine language. In a large sample, no programmers had machine language interspersed.

3.2 How are the extendibility features being used?

As might be expected, most extensions are made in terms of macros used to generate CENTRAN syntax. Some programmers, however, have extended the language in data structures, where it is weakest.

IV. CONCLUSION

4.1 The designer-implementer-educator-user relationship

From our experience in the development of the system, we can draw several conclusions that might be helpful to others. We as designers along with the implementers, educators, and users should not be disjoint groups. We should be involved as an implementer to keep in touch with reality. We should also be involved as an educator (if a feature is difficult to explain, maybe there is something wrong with it), and a user (uniformity in extension is best achieved by knowing how language is being used). The implementer should act as both educator and program counselor to get feedback on bugs being “programmed around” and to establish priorities for fixing them.

Several things about the implementer-user relationship should have been learned earlier in CENTRAN development. First, the release cycle should be rigidly controlled as soon as possible, no matter how short the cycle. It does not pay to give fixes to bugs informally. Next, old versions of the compiler should not be kept around and certainly not maintained. The maintainers are blamed for bugs that no longer exist, and much time is spent rediscovering causes for problems long since resolved.

Notices of new releases must go to everyone, not just supervision. Users often underestimate the impact on schedules of changes due to improvements to the compiler, even though the improvements were requested.

Insofar as the designer-implementer-educator-user relationship is concerned, we, as designers, should have contributed more to the structure and content of the CENTRAN courses. Frequent symposia (e.g., “Advanced Topics in CENTRAN Programming”) should have been held, with compulsory attendance.

4.2 Lessons learned

Most of what has been learned in the design and implementation of CENTRAN has been covered in previous sections. Some of the more critical aspects are worth reiterating.

CENTRAN should have been an expression language. This would not only have aided the production of more efficient, clearer, and more concise code, but would have provided a greater degree of uniformity to the language.

We should have given more thought to data types required for data reduction. Maintenance of CENTRAN programs (especially patching) should have been given greater priority in the design of CENTRAN.

Variability in the backgrounds and experiences of programmers should have been anticipated. Not enough consideration in the design of the language was given to the characteristics of the user population, and not enough emphasis was placed on continuing education.

Several of the CENTRAN design approaches were advantageous. CENTRAN was implemented by a small group of programmers. This approach avoided communication and other problems typically encountered in a large group of programmers.

The register allocation mechanism, subroutine interface primitives (Ref. 5), and extensibility mechanism designs worked well, as exhibited by CENTRAN's short development time. The ability to have partial word variables has been found useful. The structured programming features have been used extensively. The ability to program at several levels in one language made the language suitable for systems and applications programming. Finally, and most important, the design of the extended language is sufficient for the implementation of SAFEGUARD software. The SAFEGUARD programs have been successfully implemented in CENTRAN. Several studies of the suitability of CENTRAN for SAFEGUARD have been made outside of Bell Laboratories, and all have arrived at positive conclusions.

REFERENCES

1. B. N. Dickman, "ETC—An Extendible Macro-Based Compiler," Proc. AFIPS SJCC, 38 (1971), pp. 529–538.
2. M. E. Barton, "The Macro Assembler, SWAP—A General Purpose Interpretive Processor," Proc. AFIPS FJCC, 37 (1970), pp. 1–8.
3. G. M. Weinberg, *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971.
4. B. J. Mailloux, J. E. L. Peck, and A. Van Wijngaarden, "Report on the Algorithmic Language Algol 68," *Kibernetika*, 1, 1970.
5. B. N. Dickman, "Subroutine Interface Primitives for ETC," ACM, SIGPLAN Notices, 7, No. 12 (December 1972).