International Journal of Computer and Information Sciences, Vol. 10, No. 4, 1981

An Effective Preprocessor for Structured FORTRAN: The HENTRAN System

Giovanni Guida[1]

Received November 1980; revised September 1981

In the paper the new HENTRAN preprocessor for structured FORTRAN is illustrated. The motivations and the goals of the project are first outlined. The extended FORTRAN language implemented through HENTRAN is then illustrated and its adequacy and flexibility for structured programming are discussed. The basic architecture of the HENTRAN translator is described and its main features concerning reliability, portability, and efficiency are discussed in comparison with other similar systems.

KEY WORDS: Preprocessors; extended FORTRAN; portability; reliability; programming methodologies.

1. INTRODUCTION AND MOTIVE

The topic of extending standard FORTRAN is not new in the literature on programming languages. A lot of preprocessors have been proposed in the past years aiming to improve some features of the FORTRAN language along the lines suggested by the studies on programming methodologies.(5,7,9,12)

This paper is involved with the design and implementation of HENTRAN, a FORTRAN precompiler for structured programming which has been recently developed at the Milan Polytechnic Artificial Intelligence Project.(8) The motivation and the interest in such a project mainly lie on the highlevel specifications of HENTRAN and on the valid results which have been obtained. The recent definition of the new standard FORTRAN 77(3,4) has not reduced the need for such a type of precompiler but, on the contrary, has still stressed the utility for a tool for structured programming. In fact, FORTRAN 77 provides several improvements to the language and its

[1] Milan Polytechnic Artificial Intelligence Project, Milan, Italy.

standard (character data type, arrays, expressions, control statements, input/output, parameter statement, subprograms, and several other features that enhance portability) but is very poor with respect to the set of available control statements (only *if-then-else* and an improved version of *do* have been introduced) and is quite rigid for what concerns coding.

The definition of the requirements for the HENTRAN system has been based on the critical analysis of the characteristics of several precompilers described in the literature.(5,7,9,12)

Two classes of requirements are considered: a first one concerning the language to be implemented and a second one concerning the technical characteristics of the translator.

Requirements About the Language:

- theoretical completeness in expressing algorithms(11) of the set of available control structures;
- flexibility for structured top-down design and coding of programs;(6,14,16)
- facility in documentation and code reading activities.

Requirements About the Translator:

- reliability;(14)
- portability;(15)
- efficiency (source code not exceeding 500 statements, translation time near to that one of a FORTRAN nonoptimizing compiler).

The relevance and novelty of these specifications can be evaluated in connection with the characteristics of other existing precompilers, which often provide only general and poorly organized improvements in the language without considering much the performance of the translator.

The design of the HENTRAN system has been based on the following general discipline:

- to adopt general, theoretically minded methods in designing parsing and translation algorithms;(1,5)
- to follow a sound programming methodology in the system development activity.(6,13,14,16)

The paper is organized in the following way: sec. 2 is devoted to presenting the HENTRAN language; sec. 3 illustrates the main technical features of the translator; in sec. 4 the most significant results concerning software quality and performance are discussed; in sec. 5 some conclusive remarks are reported.

2. THE HENTRAN LANGUAGE

HENTRAN provides an extension of standard FORTRAN IV;(2,3) FORTRAN statements are accepted by the HENTRAN precompiler, which is fully transparent to anything which is not recognized as an HENTRAN key word.(9) 2

Let us present in detail the new control statements available in the language. For each construct we shall define the HENTRAN syntax and the corresponding semantics through an informal flowchart description.

If-then-else

```
$IF  P
$THEN
  B1
[$ELSE
  B2  ]
$ENDIF
```

P denotes a FORTRAN conditional expression (without the outermost pair of parentheses) and *B*1 and *B*2 are general HENTRAN blocks. The square brackets denote that the enclosed form is optional. Fig. 1 illustrates the corresponding flow chart.

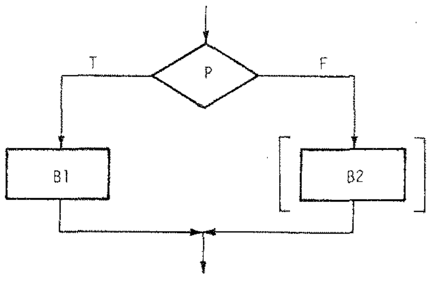

Fig. 1. If–then–else.

2 We have chosen as a design criterion to allow in HENTRAN all standard FORTRAN statements (including GO TO, arithmetic and logical IF, EQUIVALENCE, and COMMON) for three reasons: first, it is not wise to discard available resources which may be useful in some particular cases, also within a structured programming environment; second, dealing in HENTRAN with FORTRAN programs (or program parts) developed prior to the adoption of HENTRAN must not require a full revision of the already available code; third, we do not believe it appropriate to impose structured programming through a restriction on the set of available control mechanisms.

Select

$$
\begin{array}{l}
\text{\$SELECT} \begin{Bmatrix} \text{FIRST} \\ \text{ALL} \end{Bmatrix} \\
\quad \text{\$CASE } P1 \\
\quad\quad B1 \\
\quad\quad \vdots \\
\quad \text{\$CASE } PN \\
\quad\quad BN \\
\quad \begin{bmatrix} \text{\$OTHER} \\ \quad B \end{bmatrix} \\
\text{\$ENDSELECT}
\end{array}
$$

$P1,\ldots,PN$ denote conditional expressions and $B, B1,\ldots, BN$ HENTRAN blocks. The curly brackets denote that one of the enclosed forms may be used in the statement and that each form defines a different family of control structures (as default, FIRST is assumed). Figs. 2 and 3 show the flow charts corresponding to this family of structures in the case of the FIRST and ALL forms, respectively.

This structure reflects quite closely, in the FIRST form, the TEST construct proposed by Meissner(12) and, in the ALL form, it can be viewed as a generalization of the SELECT instruction which is available in the BLISS language.(17)

Perform

```
$PERFORM   ID
 ...
 ...
 ...
$PROC   ID
 B
$ENDPROC
```

ID denotes an identifier and *B* an HENTRAN block. Fig. 4 shows the flow chart representation of this control structure.

This construct is intended to facilitate the top-down construction of programs by stepwise refinement.(10,15) The statement $PERFORM allows to denote in a synthetic way, with a formal name, a whole block *B*, which

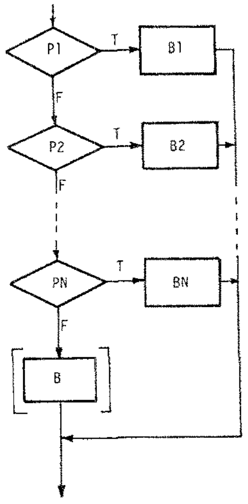

Fig. 2. Select–first.

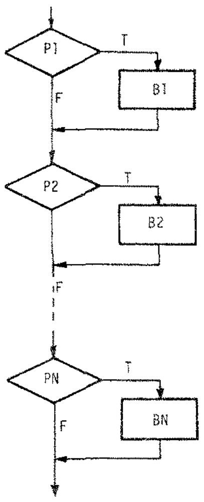

Fig. 3. Select–all.

will be expanded afterwards in the program by means of a $PROC structure. The parameters and the variables which appear in the block *B* are global to the whole program. This structure, which share some feature of the PERFORM statement available in COBOL, is therefore quite different from the FUNCTION and SUBROUTINE statements available in FORTRAN, and it is provided with the only aim of encouraging top-down coding.

While

```
$WHILE P
$DO
 B
$ENDWHILE
```

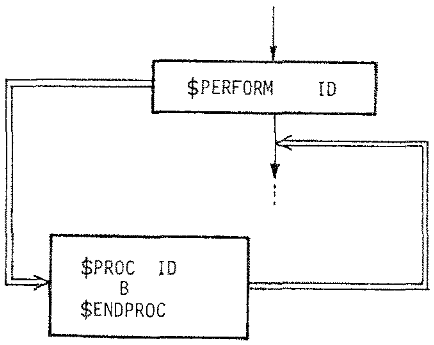

Fig. 4. Perform.

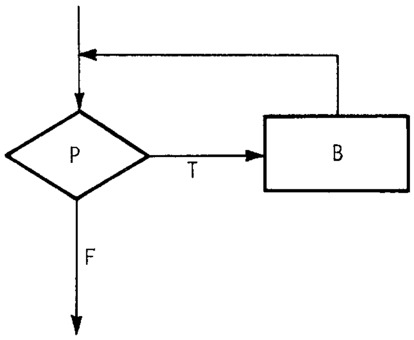

Fig. 5. While–do.

P denotes a conditional expression and B an HENTRAN block. The flow chart of this structure, which is very familiar in all ALGOL-like languages, is shown in Fig. 5.

A little modification of this structure allows to obtain the following *repeat-until* construct, which doesn't increase the theoretical power of the set of the available control structures,(11) but which is often useful in programming.

Repeat-until

```
$REPEAT
 B
$UNTIL P
$ENDREPEAT
```

Fig. 6 shows the corresponding flow chart.

Cycle

```
$CYCLE  ID, U = N1, N2, N3
 B
$ENDCYCLE  ID
```

ID is an identifier, I an integer variable, $N1$, $N2$, $N3$ are integers (with $N3 \neq 0$), and B is an HENTRAN block. B may contain any number of occurrences of the statement \$BREAK ID; when a \$BREAK ID is executed, it results in an exit from the cycle (i.e., an unconditioned jump to the first statement following \$ENDCYCLE ID). Fig. 7 illustrates the semantics of this structure.

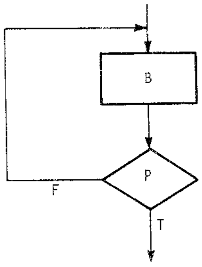

Fig. 6. Repeat–until.

Cycles can obviously be nested, thus supplying an high-level construct, original with HENTRAN, which proved very useful and flexible in the programming practice.

Loop

$LOOP *ID*
B
$ENDLOOP *ID*

ID is an identifier and *B* a general HENTRAN block. *B* may contain any number of occurrences of the statement $EXIT *ID*; when a statement $EXIT *ID* is executed, it causes an unconditioned jump to the first statement following $ENDLOOP *ID*. Fig. 8 shows the meaning of this construct.

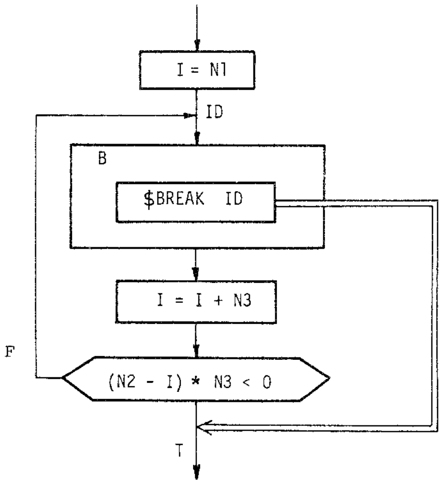

Fig. 7. Cycle.

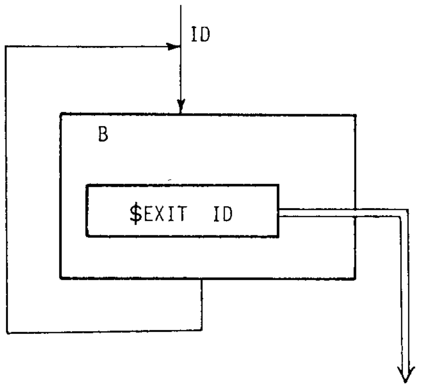

Fig. 8. Loop.

Loops can be nested (each loop must, of course, be denoted by a reserved identifier), thus allowing to implement both the classes of the RE_n and Ω_n structures.(11) Several constructs have been proposed in the past for these families; we recall DO-FOREVER in XPL, LEAVE in the BLISS language, and the LOOP-UNTIL structure proposed by Zahn.(17) Our loop structure is, in our mind, more simple and flexible in practical programming, still ensuring the theoretical features of the RE_n class.

Before concluding this section, let us recall that the HENTRAN language provides, moreover, some aesthetic improvements (free form input, comment space, etc.) and some options available at precompile time through the special command $HEN (selective translation, tracing, page heading, etc.).

3. THE TRANSLATOR

The HENTRAN preprocessor has been designed as a modular one-pass translator. Fig. 9 shows the general organization and the software architecture of the system. It is written in a portable subset of FORTRAN IV(2,15) and is organized as a hierarchical set of modules (coded as independent subroutines) each one devoted to a particular function. The decomposition of the system into modules has been performed following both the top-down refinement technique(16) and the methodology of modular programming.(13,14)

Fig. 10 illustrates the main steps of the translation activity. This has been designed as a one-pass no-backtrack translation.

The lexical analysis is performed by a finite state recognizer and the syntactic analysis by a deterministic bottom-up perser.(1) Translation and

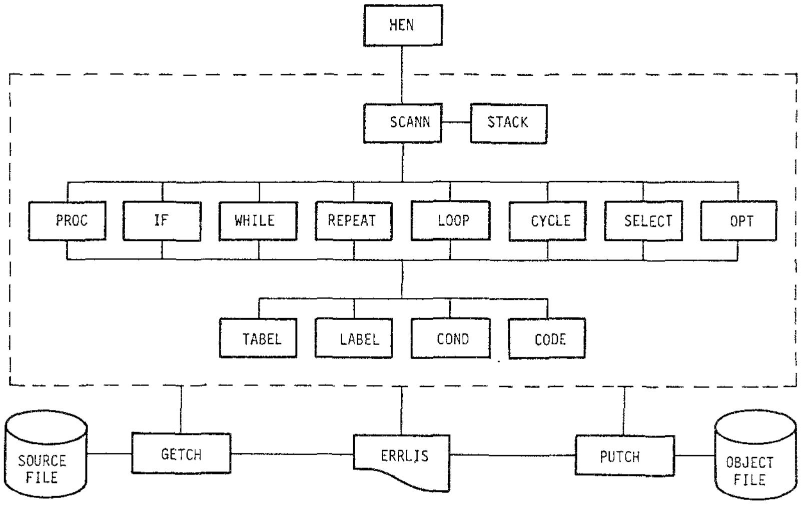

Fig. 9. Software architecture.

code generation are based on a syntax-directed translation schema: the appropriate semantic routines for code generation are activated whenever the productions to which they are bound are involved in a syntactic reduction.

Let us illustrate now in some detail the single steps of parsing and translation processes. We refer to Figs. 9 and 10. HEN is the main program and is devoted to create the appropriate environment for the translation activity. It provides for the type and dimension declarations, and for the initial assignement of the global variables; moreover, it receives from the user the information needed for starting a correct run: input unit, source file, object file, options. After these preliminary operations, HEN calls the subroutine SCANN which manages the whole parsing and translating activity. Its main task is to recognize the HENTRAN key words in the source block supplied by the routine GETCH. This scanning activity is based on the classical model of finite state recognizers,(1) which appears as the most efficient method for lexical analysis.(5) Whenever a key word is recognized, the scanner activates first the subroutine STACK for updating the central stack and the auxiliary stack which are devoted, respectively, to the check of the appropriate nesting of the constructs and of the correct sequencing of the key words. Afterwards, it calls the appropriate routine proper of the construct which has been recognized. The subroutines PROC, IF, WHILE, REPEAT, LOOP, CYCLE, SELECT, OPT are each one devoted to a particular syntactic construct and call, for the different activities required, other specific routines of lower level.

TABEL is devoted to the management of the symbol tables containing

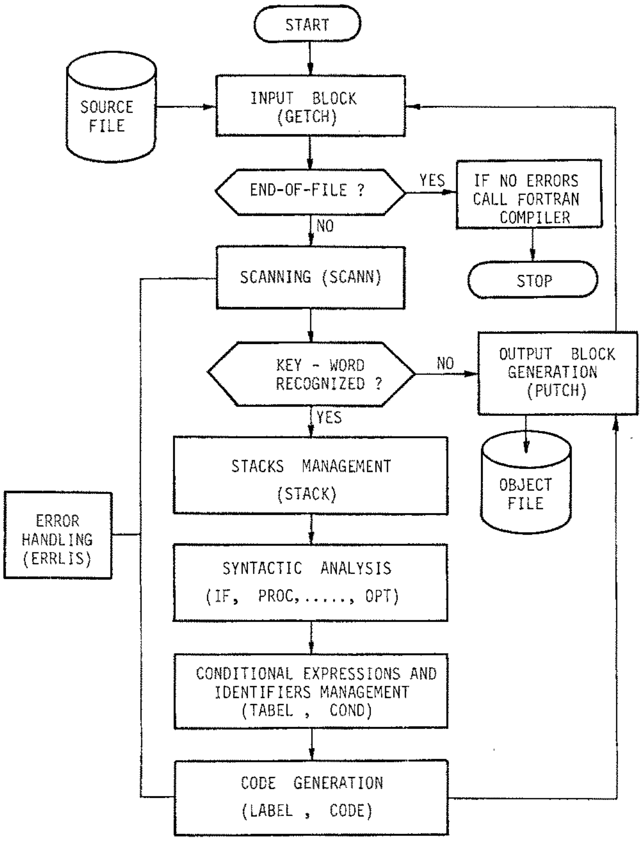
START
SOURCE FILE
INPUT BLOCK (GETCH)
END-OF-FILE ?
YES
IF NO ERRORS CALL FORTRAN COMPILER
STOP
NO
SCANNING (SCANN)
KEY - WORD RECOGNIZED ?
NO
OUTPUT BLOCK GENERATION (PUTCH)
OBJECT FILE
YES
STACKS MANAGEMENT (STACK)
ERROR HANDLING (ERRLIS)
SYNTACTIC ANALYSIS (IF, PROC,....., OPT)
CONDITIONAL EXPRESSIONS AND IDENTIFIERS MANAGEMENT (TABEL , COND)
CODE GENERATION (LABEL , CODE)

Fig. 10. Translation process.

the $PROC and $LOOP identifiers, and utilizes a linear access method, which is very efficient for tables with a small number of elements.[5] COND controls the correctness of the conditional expressions. LABEL generates the new labels required by the code generator. The subroutine CODE consitutes the kernel of code generation. Its activity is based on the technique of the semantic routines, which is a very suitable one for the development of a one-pass translator. The subroutines GETCH and PUTCH are devoted to read the source code from the input file, and to write the object code on the output files (code and listings). The subroutine ERRLIS is called by all the routines involved in the translation process whenever an error occurs, and provides for the appropriate error handling.

The HENTRAN precompiler provides 23 diagnostic messages (warnings and fatal errors). The lexical errors are generally recoverable by the system and do not affect a correct code generation; on the other hand, syntactic errors denoting an incorrect use on the constructs may cause code generation to be suspended. In such a case, if any stack underflow or overflow takes place, the stack is restored and the analysis of the source program goes further correctly, without being affected by the errors previously occurred.(1)

The object code generated by the precompiler is standard FORTRAN IV ANSI(2,6) and is quite easy to read and understand.

4. QUALITY AND PERFORMANCE EVALUATION

Fig. 11 illustrates the main steps of the development methodology which has been followed in designing and implementing the HENTRAN precompiler (note that the whole project has required an effort equivalent to

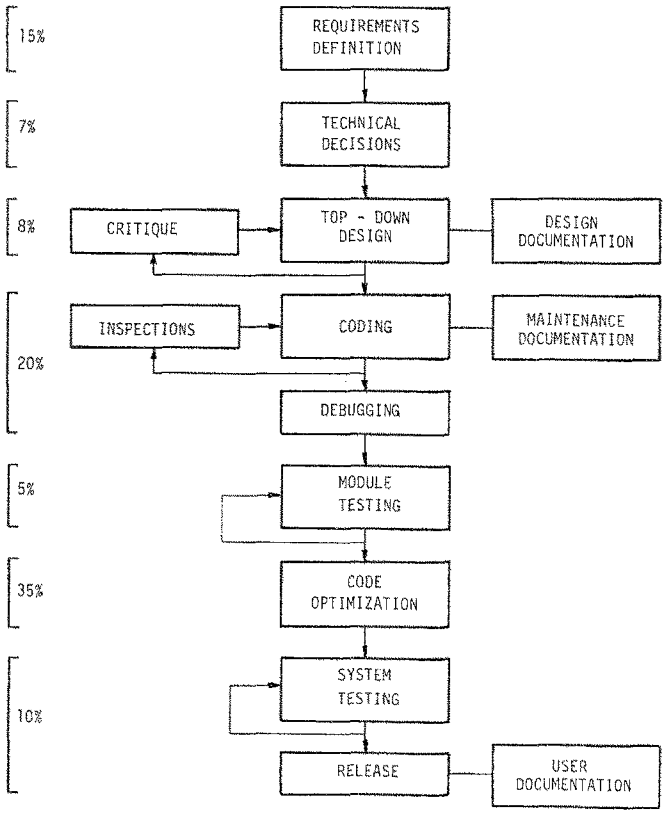

Fig. 11. Development methodology.

one person × 4 months). The present version of the system is the result of a deep optimization activity (about 35% of the whole development cycle) centered on the three parameters of memory occupation, translation time, and portability.

The precompiler consists of a main program and of 17 subroutines of 340 FORTRAN lines in all, without comments and program documentation (about 450 lines). Thanks to its very small dimension, the translator may run on several minicomputers without the need of applying segmentation and overlay techniques.

The static profile of the program:[10]

24%	CALL
20%	assignement statements
16%	logical IF
10%	GO TO
7%	RETURN
4,5%	DO, COMMON, END, SUBROUTINE, INTEGER
3%	FORMAT, arithmetic IF, WRITE, DATA, LOGICAL
1,5%	CONTINUE
0,5%	READ

shows the highly modular structure of the system and the well balanced use of FORTRAN.[3]

The translation efficiency of the precompiler has been evaluated by comparing the average HENTRAN translation time with the average FORTRAN compilation time with different compilers, on different computer systems, and with sets of programs of different size. The experimental results obtained are reported in Fig. 12.

The run time profile of the precompiler[9,10] shows that 80% of the translation time is spent in input/output operations and is due to less than 10% of the source code (GETCH and PUTCH routines). This feature suggests that by substituting (part of) the standard input-output routines with dedicated assembler programs would greatly increase efficiency, deeply damaging, however, the portability of the system. The experimental work done has shown that by utilizing macroassembler routines on a DEC PDP-11/34 the total translation time may be reduced up to 60%.

The portability of the system is ensured both by the use of a restricted portable subset of FORTRAN IV ANSI.[2,5,15] in the coding activity, and by the particular technic adopted for encapsulating the nonportable segments of the program (four segments, less than 10% of the source code). The

[3] Please note that the sum of the percentages is 116, 5 > 100, since the logical IF statement can embed other statements.

average number of source program lines	FOR	F4P	FTN4	HEN
100	8s	20s	10s	5s
600	35s	85s	37s	22s
1000	48s	125s	65s	32s

FOR	- FORTRAN compiler on DEC PDP-11/34
F4P	- extended FORTRAN compiler on DEC PDP-11/34
FTN4	- FORTRAN COMPILER on HP 1000
HEN	- HENTRAN precompiler on DEC PDP-11/34 or HP 1000

Fig. 12. Translation efficiency.

portability experience done (on DEC PDP-11/34, HP 2100, HP 1000, UNIVAC 1100) has shown that a programmer with a good knowledge of the host system can generate the precompiler in only a few hours work.

Let us conclude this section by outlining that the reliability of the HENTRAN system, which proved very high, is primarily due to the sound development methodology followed in design and implementation (see Fig. 12).(13) The following error profile (errors detected and corrected in each phase of the development)

10%	design: basic conceptual errors concerning algorithms and program structure
40%	code inspections : trivial coding errors
47%	module testing : undesired side effects
3%	system testing : subroutine link
1 error	field testing and experimental use (six months): uncorrect management of a particular type of stack overflow

ensures, moreover, a good degree of confidence in the system correctness (the total number of errors to which the above profile is referred is 31).

The HENTRAN system has been tested first (module testing) through an appropriate set of test data which have been prepared by hand directly by the author, and afterwards (system testing) through a sample of application programs mainly concerning sorting and searching problems (about 20 programs of 50–350 FORTRAN lines each). Field testing and experimental use have been performed by several classes of users in different fields, such as scientific applications, nonnumerical programming (specialized text editors, user interfaces, utilities, etc.), high school teaching on programming.

5. CONCLUSION

In the paper the HENTRAN system has been presented; a particular attention has been devoted to the methodological aspects concerning the design and the implementation and to the evaluation of the results obtained.

A short comparison with other FORTRAN precompilers mentioned in the literature shows the good performance which has been obtained with the HENTRAN system. If we consider, for example, the data reported in (5) for RATFOR,(1) optimized RATFOR, and MOUSE4,(5) and we compute the ratio precompile time/FORTRAN compile time we get the following results (for programs of about 600 lines):

RATFOR:	9.47
optimized RATFOR:	5.54
MOUSE4:	1.40
HENTRAN:	0.25 ÷ 0.62 (depending on the type of FORTRAN compiler used),

which are a valid index of the programming efficiency. This is primarily due to the appropriate use of general, theoretically minded methods for compiler construction, and to the constant application of a sound programming methodology.

ACKNOWLEDGMENTS

I am indebted to F. Spada and E. Viganó for their basic contribution in system design and implementation. Moreover, I am grateful to the students and colleagues which have kindly accepted to work with HENTRAN during the advanced stages of the system testing, and which have supplied several comments and suggestion.

REFERENCES

1. A. V. Aho and J. D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, Mass., 1977).
2. *American National Standard Programming Language Fortran* (Am. Nat. Standard Institute, New York, 1966).
3. *American National Standard Programming Language Fortran* (Am. Nat. Standard Institute, New York, 1978).

4. W. Brainerd (Ed.), "Fortran 77," *Comm. ACM* **21**(10):806–820 (1978).
5. D. Comer, "MOUSE4: An Improved Implementation of the RATFOR PREPROCESSOR," *Software-Practice and Experience* **8**:35–40 (1978).
6. E. W. Dijkstra, *A Discipline of Programming* (Prentice Hall, New York, 1976).
7. For-Word, "Fortran Development Newsletter," *SIGPLAN Notices* **4**:11 (1976).
8. G. Guida, F. Spada, and E. Viganò, *HENTRAN-Manuale d'Uso* (Istituto di Elettrotecnica ed Elettronica del Politecnico di Milano, Milano, Italy, 1979).
9. B. W. Kernighan, "RATFOR—A Preprocessor for Rational Fortran," *Software-Practice and Experience* **5**:395–406 (1975).
10. D. E. Knuth, "An Empirical Study of FORTRAN Programs," *Software–Practice and Experience* **1**:105–133 (1971).
11. H. F. Ledgard and M. Marcotty, "A Genealogy of Control Structures," *Comm. ACM* **18**:629–639 (1975).
12. L. P. Meissner, "Proposed Control Structures for Extended FORTRAN," *SIGPLAN Notices*, **11**:16–21 (1976).
13. G. J. Myers, *Software Reliability: Principles and Practice*, (John Wiley, New York, 1976).
14. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM.* **15**(12):1053–1058 (1972).
15. M. A. Sabin, "Portability-Some Experiences with FORTRAN," *Software-Practice and Experience* **6**:393–396 (1976).
16. N. Wirth, "On the Composition of Well-structured Programs," *ACM Computing Surveys* **6**:247–259 (1974).
17. W. A. Wulf, et al., *BLISS Reference Manual*, (Carnegie-Mellon University, Pittsburgh, 1971).
18. C. T. Zahn, "A Control Statement for Natural Top-down Structured Programming," *Lecture Notes in Computer Science* **19**:170–180 (Springer Verlag, New York, 1974).

Printed in Belgium