RADC-TR-77-148
Final Technical Report
May 1977

COMPILER ACCEPTANCE CRITERIA GUIDEBOOK

Proprietary Software Systems, Incorporated

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Douglas White

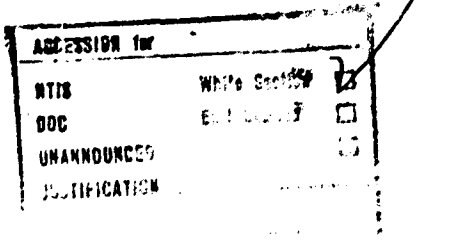
DOUGLAS WHITE
Project Engineer

APPROVED: Alan R Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office



A 2[illegible]

Do not return this copy. Retain or destroy.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-148	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): COMPILER ACCEPTANCE CRITERIA GUIDEBOOK	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report.
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): Joel/Fleiss, Guy/Phillips, Angel/Alvarez	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0229
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Proprietary Software Systems, Inc., 292 South LaCienega Blvd/Suite 218, Beverly Hills CA 90211	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55811217
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: May 1977
	13. NUMBER OF PAGES: 151
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 152 p.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

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)
Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Douglas White (ISIS)

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

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
PSS has composed a set of criteria that may be used to provide guidance in the procurement of compilers. These guidelines are presented in tutorial form to provide a useful tool to the individual who does not have experience in the specific area of compilers. The guidelines contain a check list of items that should be considered in the specification of a compiler and the acceptance and testing of that compiler. Examples and sample forms are provided to demonstrate the use of the guidelines described in the report.

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

HDC 529

I. Project Background

The acquisition of a compiler system often represents the most critical aspect in the development of a successful software system. Heretofore, virtually no guidelines were provided for buyers to facilitate the specification of a new compiler.

Software development involves many considerations. The selection of the appropriate language tools in the development process is not only critical, but will have major effects in terms of cost, development time, efficiency, portability, and ease of maintenance and modification.

Minimizing programming errors often depends a great deal on matching the complexity of the application with an appropriate programming language. Many variables should be considered when deciding on the features of a particular system.

There is a myriad of facilities and features that a compiler system might possess with respect to a modern computer system. Also there is no doubt that the specific requirements for compiler systems in different installations are widely divergent and based on the best utilization of available resources for particular applications.

II. Objective

The purpose of this research and development project was to define criteria and guidelines which could be used in compiler procurement specification and acceptance decision making processes. The resultant document, "Compiler Specification/Acceptance Handbook" is intended to be used as a guidebook in the specification of a compiler system and in the analysis of its acceptability.

The purpose of the document is to aid the compiler procurement agency. It is a guide to the specification of appropriate compiler design, implementation, testing and acceptance. No attempt was made to design or suggest a universal set of tests for all languages and compilers. In many cases, automated verification systems already exist for this purpose (JCVS - JOVIAL Compiler Validation System). It does suggest topics which should be considered in designing tests for a specific project and indicates usual results of importance. The weighting factors assigned are based on "sample" systems and compiler requirements and are by no means fixed. Each agency must assign its own weights and relative importance to each item. The sample weights are hypothetical guidelines to help each procurement agency to develop its own historical specification/ acceptance criteria.

III. The Handbook

The compiler specification criteria developed provide responsible individuals with a means of identifying the features and facilities a compiler system should contain to best utilize its available resources within the necessary budget and time constraints. A chart of major considerations was developed with sub-charts delineating each major chart item. It was the conclusion of this effort that the best method for insuring an acceptable compiler is an acceptable specification. Therefore, a great deal of emphasis was placed on compiler specification criteria.

Compiler acceptability, therefore, became a matter of reviewing the items specified and assigning appropriate weighting scores which would eventually determine when a compiler is acceptable. Acceptance matrices were developed which contain the major areas of consideration. These matrices are to be used to calculate an acceptability index. Sub-matrices were developed for each major item within the matrix. A predetermined sub-matrix item weight along with a user determined acceptability factor will be utilized to calculate a score. The score will then be used as an index into an acceptability chart. The weights or potential scores used are based on sample matrix applications to existing compilers.

In addition to the charts and matrices, examples were developed which should help the user to make specific choices according to the desired needs of a specific compiler system.

The form and content of the handbook are based on the conclusions reached during the research phase of the effort. This included:

- A wide range of differing 'authoritative' and 'expert' opinions and definitions exist with respect to:

  compiler components, processes and structure
  compiler 'features'
  compiler 'optimization' and 'optimizers'
  programming language definitional forms
  operating system functional characteristics
  compiler efficiency
  measures of efficiency

- A number of previous study efforts relating to program characteristics of programs coded in higher order languages support views which are diametrically opposed to each other.

- That in all previous efforts significant and accurate data has not been accumulated in a disciplined fashion within a 'production' environment to lend 'authoritative' credence to an absolute set of criteria or guideline for procuring a compiler.

- That subjective (empirically developed over time) 'weights' have to be established and assigned to the following type items in deciding what specifications and requirements are to be included in procuring a compiler:

    a. value of human resources in program development, debugging and maintenance phases.

    b. value of computer time used in the compilation process and that of the program execution time.

    c. value of linear time available to a project or installation (time required to implement a compiler).

    d. value of the total dollar cost of developing, debugging and maintaining a compiled with extensive features.

IV. Future Efforts--Updating the Handbook

It became apparent at a very early stage in the study that if the procuring agency stressed a particular item in its specification, then all acceptance criteria effected by that item would be weighted heavily.

For example, if precise language definition was stressed in the specification, then the accuracy and reliability of syntax/semantics analyzers (acceptance criteria) would be heavily weighted.

Therefore, the sample Compiler Acceptance Evaluation Matrix developed was based on a hypothetical "average" compiler. The values assigned represent composites of values derived from the analysis of several languages and their compilers.

The procuring agency should prepare a new set of potential scores as a function of language, project, user group and other constraints discussed in the handbook. For example, a file management system written in COBOL would probably not depend heavily on numeric accuracy (past 3 decimal places) or even resource utilization; but, it is likely that system interface would be extremely important.

It seems advisable that future compiler system contracts include, as part of the payment, an amount based on the

acceptability score. This additional financial remuneration could provide the incentive needed to turn acceptable and good compilers into excellent compiler systems.

EVALUATION

The procurement of compilers for computer programming languages is an important and difficult task. The quality of the resultant compiler and the aids it provides the programmer can determine costs or savings that are many times greater than the price of the compiler. Many government agencies that do not have expertise in the field of compilers have requirements to procure compilers including the specification and acceptance of these compilers. The goal of the Compiler Acceptance Criteria Guidebook effort was to develop a document that would provide assistance to government agencies in the procurement of compilers.

Proprietary Software Systems, Incorporated was selected to develop guidelines and to produce the guidebook. The guidebook that was produced contains information that will assist in the procurement of more cost effective compilers by insuring more complete specification of the compiler and more adequate criteria by which the compiler is accepted. Use of these guidelines should result in higher quality and more useful compilers with the end result of lower software costs.

Douglas White

DOUGLAS WHITE
Project Engineer

TABLE OF CONTENTS

INTRODUCTION

This document is intended to be used as a guidebook in the specification of a compiler system and in the analysis of its acceptability. It is divided into three major parts, compiler specification criteria, compiler acceptance scoring, and a sample acceptance scoring matrix.

Part 1 may be used as a glossary of compiler terminology It should be reviewed carefully by the reader unfamiliar with these terms before proceeding to parts 2 and 3.

The purpose of this document is to aid the compiler procurement agency. It is a guide to the specification of appropriate compiler design, implementation, testing and acceptance. No attempt is made to design or suggest a universal set of tests for all languages and compilers. In many cases, automated verification systems already exist for this purpose (JCVS - JOVIAL Compiler Validation System). It does suggest topics which should be considered in designing tests for a specific project and indicates usual results of importance. The weighting factors assigned in Part 3 are based on "sample" systems and compiler requirements and are by no means fixed. Each agency must assign its own weights and relative importance to each item. The sample weights are

hypothetical guidelines to help each procurement agency to develop its own historical specification/acceptance criteria.

Part 1, Compiler Specification Criteria, provides responsible individuals with a means of identifying the features and facilities a compiler system should contain to best utilize its available resources within the necessary budget and time constraints. A chart of major considerations is provided with sub-charts delineating each major chart item. It was the conclusion of this effort that the best method for insuring an acceptable compiler is an acceptable specification. Therefore, a great deal of emphasis is placed on Part I, Compiler Specification Criteria.

Part 2, Compiler Acceptability, reviews the items specified in Part 1 and assigns appropriate weighting scores which will eventually determine when a compiler is acceptable. The acceptance matrices are provided which contain the major areas of consideration. These matrices are used to calculate an acceptability index. Sub-matrices are provided for each major item within the matrix. A predetermined sub-matrix item weight along with a user determined acceptability factor is utilized to calculate a score. The score is then used as an index into an acceptability chart. The weights

or potential scores used in Parts 2 and 3 are based on sample matrix applications to existing compilers.

In addition to the charts and matrices, examples are provided which help the user to make specific choices according to the desired needs of a specific compiler system.

Part 3, the sample Acceptance Scoring Matrix is a summarization of the topics discussed in Part 2. In this matrix, the acceptance items are listed with the corresponding weighting factors appropriately broken down into sub-matrix form.

It should be noted that the specification and eventual acceptability of a compiler system is a complex process. This guideline establishes an orderly approach in determining the specification and acceptability criteria for a compiler system. It thereby helps to insure a thorough and objective decision-making process when selecting a compiler.

Part 1 Compiler Specification

The acquisition of a compiler system often represents the most critical aspect in the development of a successful software system. Heretofore, virtually no guidelines were provided for buyers to facilitate the specification of a new compiler.

Software development involves many considerations. The selection of the appropriate language tools in the development process is not only critical, but will have major effects in terms of cost, development time, efficiency, portability, and ease of maintenance and modification.

Minimizing programming errors often depends a great deal on matching the complexity of the application with an appropriate programming language. Many variables should be considered when deciding on the features of a particular system.

The charts on the following page present the major areas of concern when specifying a compiler. Each item is then further delineated in subsequent sub-charts. All items in the sub-charts are discussed as to their relative merit in the acquisition of a compiler system.

There is a myriad of facilities and features that a compiler system might possess with respect to a modern computer system. It is assumed that the specific requirements for compiler systems in different installations are widely divergent and based on the best utilization of available resources for particular applications.

The major areas of consideration in the specification of a higher level language compiler include:

Compiler Costs
Language Definition
Compiler Options
Extensibility
Transferability
Computer Environment
System Interface
User Profile
Documentation
Schedule and Installation
Acceptance Tests
Maintenance and Support

COMPILER SPECIFICATION - MAJOR AREAS OF CONCERN

The charts on the following pages further delineate each of the above major areas of concern.

1.1 Compiler Costs

Of primary concern in the acquisition of a compiler is the "actual" dollar cost. Since the cost associated with a compiler system involves the original investments as well as other significant on-going expenditures, the topic of "actual" cost becomes increasingly complex.

Usually, the compiler procuring agency is only concerned with the initial "out-of-pocket" outlay. The following chart and subsequent explanations may help the compiler specification writer to develop insights into "actual" compiler costs. Actual costs include:

Purchase Price
Cost of Compilation
Cost of Execution
Level of Expertise
Resource Utilization
Resource Acquisition
Maintenance Costs
Enhancement Costs-Options
Training Costs
Re-targeting Costs
Re-hosting Costs

COMPILER SPECIFICATION - COMPILER COSTS

- Purchase Price

A major consideration in the acquisition of any item is the purchase price. Although the purchase price is almost always not part of a compiler specification, it is often specified indirectly (in terms of man-hours and job classifications). Virtually all items delineated in the specification charts have a direct bearing on the purchase price.

The compiler specification writer should investigate the relative costs and benefits of the features under consideration. This will assure the most effective usage of the available resources. The relative merit of a special or "extra" feature must be analyzed in relation to its added cost in order to maximize the utility of the compiler system.

It is very common in the development of a compiler system to have progress payments made as the contract proceeds. This is usually due to the large expenditure of labor and computer time often necessary to complete a compiler project. It is highly improbable that any software organization specializing in compiler implementation would be willing or able to work for long periods without financial reimbursement.

Since compiler system contracts are usually fixed price, as opposed to time and materials, it is advisable to identify milestones at which point partial funds might be made available to the developing agency. Progress payment provisions relating directly to partial product deliverables have to be carefully constructed to insure that the balance of funds flowing to the vendor is approximately equal to the value flowing back to the user. In this manner, project termination will not create major hardships for either party.

Specific items include:

- Basic compiler (software) purchase price
- Option prices
- Advances
- Milestones

- Cost of Compilation

A major cost that is usually ignored in the specification of a compiler is the cost of compiling a program. Individual compiler implementations of the same compiler specification can have vastly different costs associated with the translating of a program.

Part of any compiler specification should contain a maximum cost per specified test program. The test

programs should represent a reasonable set of tests in terms of expected features used. Of extreme importance is the compiler's operation during large compilations. Often a compiler will perform well for small modules but will run slowly when reaching certain "intrinsic limits." Too often compiler buyers have neglected this area and have received a compiler that met all specifications, yet was of little practical value.

Specific items include:

- small program costs per statement
- large program costs
- medium or average costs
- overhead costs

- Cost of Execution

The code generated by a compiler and the tools provided to improve the efficiency of the code can be vital in the overall effectiveness of the compiler systems. A compiler system is merely a tool to be used in the development of software.

The intended application of programs generated by a compiler coupled with the expected life of the compiled program (number of times to be used) can often make

this aspect the most critical of all cost items. Depending on the particular application, the erudite writer of compiler specifications should include some minimum acceptable ratio of compiler-code-generated execution cost versus the execution cost of an identical machine language program written by an above average programmer. It should be noted that the term execution cost implies not only execution time, but also computer resources utilized by the compiled program (memory, disk, I/O access, etc.). This may be handled in a small, medium, large program fashion as for compilation costs. Specific items of concern are:

- CPU time per statement or program
- core usage
- I/O access time
- wait or dead time
- disk storage
- tape drive mounts

In large computer systems there will often be a machine unit cost of execution formula which may be used for computing total cost. This may be quite complex and is left to the specification agency discretion. A prime example is the OS 370 HASP system.

- Level of Expertise

  Another often neglected cost item is the "quality" of the compiler system as a major determining factor in the level of expertise required by users. The relative time it takes to get a job done is usually directly related to the level of the language (assembly-higher level-special purpose) and its features.

  If a junior level programmer can accomplish the productivity of a senior level programmer because of the compiler system's language and features, then the dollars saved become a major cost factor. The quantitative analysis and relative merit of new features is very important to the future costs associated with the compiler's usage.

- Resource Utilization

  It is not at all uncommon for the vendor to require the usage of resources supplied by the procuring agency. At a minimum these resources will probably include the participation of staff in both the design concept and during the compiler acceptance approval cycle.

  It is quite common for the buying organization to supply machine time for the development, checkout, and installation

of the compiler system. In addition, other resources are often provided in terms of office space, telephones, secretarial services, keypunching, etc. Thus, it is necessary to stipulate what types of resources will be supplied by the procuring agency.

If the maximum level of resources is exceeded, then the specification should state the consequences (charges). This will insure that excessive resource utilization will be minimized by the vending organization.

On the other hand, very stiff penalties may result in inadequate testing, or non-optimized code generation. Resource expenditures should be as liberal as possible to insure effective, on-time deliveries. Low resource costs usually may be obtained by use of slow turnaround, low priority computer usage and subsequent schedule delays. Therefore, resource availability is also of primary concern.

- Resource Acquisition

  In some cases it may be necessary for the procuring agency to purchase additional hardware to facilitate the development or use of a compiler system.

In general, the procuring agency must determine the resources to be provided in advance of contract negotiations or even RFP preparation. Appropriate responses should include a statement of resources required in addition to those listed in the RFP. Hardware items most likely to be critical in compiler utilization are:

- Core storage (fast)
- User interface devices (printer, CRT, card readers)
- Intermediate I/O (modems, multiplexors)
- Secondary storage devices (disk)

Proper allowances must be made for these items to insure a compiler system which will be available to the maximum number of users over a full range of applications.

- Maintenance Costs

The compiler specification should contain a provision for vendor maintenance after acceptance. The maintenance period should extend for the useful life of the product. A heavily used compiler will become reliable within a short period of time.

In this case, a front-loaded maintenance budget is recommended. In general, the vendor should be required to supply the first year's maintenance as part of the

compiler development costs. A typical maintenance budget is:

Years After Acceptance	Percentage of Development Cost
0-1	0 %
1-2	20%
2-Life of the Product	10%

This is subject to the choice of vendor (reputation and location), complexity of the language/project, time schedule for implementation, and previous experience.

- Enhancement Costs-Options

  The compiler RFQ should explicitly require fixed price estimates for enhancements and additions to the basic system. Items to consider include:

  - Debug packages
  - Optimization routines
  - Language extensions

- Training Costs

  Initial procuring agency personnel training and education costs should be included in the vendor proposals at a very low cost. Training includes proper documentation of the language, compiler usage and program listings. Additional specifications for later training of other vendors/users

of the compiler must be included. Too often, an agency becomes dependent on a particular vendor for subsequent acquisitions. Usually, this is because the agency-owned compiler cannot be altered/maintained by other vendors. The original vendor must be required to accept responsibility for training in subsequent installations as well.

- Re-targeting Costs

  The compiler specification must include provisions for additional target computer implementation. The cost proposals should include vendor costs estimates for the full implementation and costs for training of other vendor personnel. The specification should require an absolute minimum cost for re-targeting and concommitant advanced technology. The code generator or target dependent portion of the compiler should be one of the least expensive to duplicate.

- Re-hosting Costs

  Re-hosting is the process of moving the compiler system to a new computer without changing the result or output (target programs). The vendor should be required to prepare cost quotations for every major host system to be used by the procuring agency and major civilian companies.

The re-hosting costs should include estimates for original and new vendor implementations.

1.2 Language Definition

The most important part of any compiler system is its ability to properly translate each specified language form to obtain the eventual execution of the compiled program. The specification of the features of a compiler language should be as clear as possible in order to avoid any misinterpretations.

Several mathematical languages have been developed that facilitate the specification of compiler features. Relatively accepted, mathematically oriented languages, such as Backus Nauer, already exist for this purpose. In addition, it is also advisable to have a brief English explanation of the various language components. The clarity of syntax and semantic representation is of utmost importance to the compiler implementor as well as the eventual user.

In order to eliminate any ambiguities or misunderstandings the definition of a new computer languages' format must be defined as clearly and concisely as possible. Meta-linguistic syntactic definitions have been developed in a complete, efficient, and concise form. In fact, the success of these definitions can be judged from the

diversity of language descriptions that have been developed from a meta-linguistic notation.

The language in which the compiler is defined is termed a meta language. The meta language must be uniquely distinguishable from the language being described. The following chart lists the items of a compiler language which should be defined in both a meta language as well as in a prosaic manner.

If the language to be specified already exists in other environments, then it is highly likely that a formal meta-linguistic mathematical specification is already in existence. Popular languages such as FORTRAN, COPOL, BASIC, ALGOL, PL/1, and JOVIAL have all been defined via vigorous notations. Aspects of most languages can be categorized and described as:

Syntax Characteristics
Declarative Statements
Control Statements
Subroutine Statements
Processing Statements
Allocation Statements
Input/Output Statements
Formating Statements
Compiler Directives

- Syntax Characteristics

  The specification of a compiler would be meaningless without a formal description of the language to be translated. The syntax rules for the following items should be included in the specification:

  - Character Set
  - Symbol Construction
  - Keywords
  - Statement structure
  - Comments
  - Statement Termination
  - Variables
  - Logical Operators
  - Relational Operators
  - Arithmetic Operators
  - Expressions
  - Literals
  - Functions
  - Constants

- Declarative Statements

  The declarative statements specify the variables that will be used in writing a program. Often, the declaration statements include information that describes

attributes of the variables.

- Control Statements

  Control statements control the program flow. Transfer statements, conditional statements, switch statements, iteration statements, and decision table statements are examples of control statements.

- Subroutine Statements

  Subroutine statements provide a means of modularizing a program. Usually these statements include facilities for declaring a subroutine, referencing a subroutine, and exiting a subroutine. In addition, subroutines often have the capability of passing arguments.

- Assignment Statements

  Assignment statements provide a means of assigning values to variables.

- Allocation Statements

  Allocation statements control the placement of variables in memory. In addition, allocation statements declare the dimension of variables. In some languages the declaration statements perform some of the functions of the allocation statements.

- Input Output Statements

  Input output statements provide a means of getting information into and out of a computer. The capability to describe the devices is included in some languages.

- Formating Statements

  Formating statements describe the format of data to be read into or out of a computer.

1.3 Compiler Options

There are a myriad of capabilities and facilities that a compiler may possess or influence with respect to a modern computer system. This guidebook assumes that compiler requirements among different installations are widely divergent. This is especially true when selections are based on a maximum utilization of available resources.

The following chart and subsequent explanations define numerous capabilities and facilities which improve the utility of a compiler system. Assignments are made for expected dollar benefit versus anticipated costs of each compiler option. This eases the choice of items to include in the specification.

Compiler Types
Subsets/Special Versions
Optimizations
Measurement Tools
Debug Aids
Test Aids
Documentation Aids
Cross Reference/Dictionaries
Source Libraries
Compools
Text Maintenance
Directives
Detailed Diagnostics
Reformatters
Standard Verification
Source Listings
Partial Compilations

COMPILER SPECIFICATION - COMPILER OPTIONS

- Compiler Types

  The purpose for which a compiler will be used and the intended applications for which programs will be developed all have an impact on compiler type.

  - BATCH

    BATCH compiler requires all input files to be supplied before compilation. This is the most popular implementation type for compilers. It provides for the most efficient means of minimizing the cost per compilation.

  - CONVERSATIONAL/INCREMENTAL

    The CONVERSATIONAL compiler communicates with the user throughout the translation process. The user can continually make correction/modifications as the compiler translates/executes the users' program. A conversational compiler is usually very effective for the development of a program where compilation cost is not very important.

  - INTERPRETIVE

    An INTERPRETIVE compiler performs the execution of a program directly rather than preparing an object version of the program to be executed later. An interpretive compiler usually generates less efficient object code

but minimizes compilation time and re-targeting tasks.

- Subsets/Special Versions

  The primary motivations for subsetting are cost and time. Subsets permit smaller compilers which can be developed more economically and/or in less time. It is important that a subset be a "proper" subset to insure upward compatibility.

  Often a compiler of an already existing language is specified as a proper subset except for several "special" features. These 'subset versions" special features may be more costly (in terms of compatibility and transferability) than the benefits they provide.

- Optimization

  Over the last two decades numerous techniques have been developed to improve the efficienty of code generated by compilers. Usually these optimization features are divided into local and global optimization. The cost associated with developing optimization facilities for a compiler can usually be divided into two areas: additional development costs and more costly compilation.

Normally, if sophisticated optimization facilities are to be developed, it is advisable that they be conditional and selective. For example, it is not usually necessary to optimize application programs during their development cycle. In addition, numerous studies have shown that small portions of a program usually consume the majority of a program's execution time. Hence, optimization of selected areas usually proves the most effective in terms of cost/benefit ratio.

There are often several alternatives to expensive optimization facilities. These include measurement facilities which pinpoint areas to be optimized and facilitate interfacing to assembly language programs. Each optimization facility should be weighed in terms of cost/benefit ratio.

Local optimization features are usually less expensive than global optimization facilities. The following is a list of potential local optimization facilities:

- Reordering of the evaluation of an expression
- Efficient use of registers and memory
- Common expression elimination
- Redundant statement elimination
- Dead variable elimination

- Factoring (eg. A*B+A*C to A*(B+C))
- Constant evaluation
- Retargeting of jumps (eg. GOTO L2 becomes GOTO L3 if L2 contains a GOTO L3)
- Elimination of redundant stores
- Efficient utilization of target machine instructions

The following is a list of global optimization facilities:

- Flow analysis - this is usually a very costly optimization technique. Its benefits involve the re-ordering of statements and the elimination of unnecessary statements. Many of these benefits can also be attained through proper programming techniques by the user.
- Elimination of assigning a variable to an expression that is never used.
- Base register versus active register usage.
- Code redistribution - moving code segments to a path that eliminates redundant execution.

- Measurement Tools

Measurement tools are an invaluable asset in improving the performance of a compiler system. There are three basic areas of potential improvement that measurement systems can be used in a compiler system. These are:

- Measurements to improve the language itself.
- Measurements to improve the performance of the compiler's operation.
- Measurements to improve the performance of programs that have been compiled.

Usually measurement facilities can be built into a compiler at a fraction of the cost of optimization features. It is recommended that these facilities be both optional and selective.

The gathering of statistics, such as features used; combinations of statements; and frequent user errors provide valuable insights to language designers in terms of improving the utility of the language.

Statistics regions help pinpoint areas to be optimized. These statistics can point out vital areas of the compiler/application program that should be optimized.

- Debug Aids

One of the major purposes of a higher level language is to simplify the cost of developing and maintaining programs. Debug aids are indispensable tools in the checkout of new

programs/modifications or in the location and eventual correction of problems. The following is a list of debug aids:

- Corrections

  The ability to modify object programs without having to recompile saves both dollars and time.

- Snapshots/Dump

  The displaying of variables in a format that is compatible with the language is very beneficial.

- Item/Location Modification

  The facility of being able to see a variable's modification at a particular location (routine, statement number) is extremely helpful in the checkout of a program.

- Traces

  The display of sequences of program instructions (especially symbolically coupled with item modification) is useful for efficient program checkout.

- Conditionality

  The ability to invoke the above features conditionally is a very powerful program development methodology.

- Test Aids

  The generation of test programs to verify the "correctness" of a set of programs is a major consideration when developing a system. In the last decade tools have been developed which aid the generation of test programs.

  Intrinsic to these tools is a flow analysis of the program to be tested. Since compilers must make a like analysis during global optimization, several compiler systems now include the preparation of test programs as an integral part of the system.

- Documentation Aids

  The facility for effectively interjecting program comments into the source program is of utmost importance. It is, however, an integral part of the language definition and not an option. The compiler may be used to enhance this feature by specially formating printing of comments, automatically indenting loop structures, and summarizing classes of variables and statements used. Many of these aids are traditionally supplied via the cross-reference facility.

- Cross Reference/Dictionaries

  Most compilers will output an alphabetized symbol table (dictionary) at the conclusion of each module. The dictionary provides an easy-to-use reference to the type and value associated with each user declared variable.

  A cross reference provides a means of identifying where each variable is referenced in a particular module. This facility is a very important asset in the development and maintenance of programs. In languages such as JOVIAL the scope of each variable is also of utmost importance and can be further emphasized here.

- Source Libraries

  Often, an application will contain a set of programs with identical source statements (usually declarations). It is both more economical and less error prone to be able to reference a set of source from a single statement rather than including the same source in numerous modules.

- Compools

  Compools provide a means of gathering common variable definitions into a single file. Compools usually differ

from source libraries in that they have been pretranslated and exist in a more readily accessible form. Usually a language that facilitates compools has specific rules regarding undefined variables in a program and their accessability from the compool.

- Text Maintenance

  Text maintenance provides an efficient means of maintaining and modifying a source program. Nearly all major computer systems already contain text editors and source maintenance programs. If neither of these are available, the compiler system should contain a facility for updating source files.

- Directives

  Directives provide a means for the user to guide the compiler in performing a particular compilation. The utility of the directives depends on the application to be performed and the function of the directive. The following is a brief list of some of the directives which might be included:

  - Macro - A set of directives which permits the user to define new facilities in terms of existing facilities. This facility can make the language

more flexible to future needs (but may cause future compatibility problems).

- Conditional - A set of directives that determines which statements are to be compiled. This allows a source program to adapt to different environments based on predefined conditions.
- Optimization - A set of directives allowing the user to specify which areas to optimize and the ratio of speed versus space considerations.
- Mode - A set of directives which influences the type of code to be generated (eg. whether or not a subroutine is to be re-entrant).

- Detailed Diagnostics

The level of information provided by a compiler for error messages is extremely important. The ability of the compiler to pinpoint the reason for the error is vital to the efficient development of software.

It is unadvisable (unless other constraints prevail) to have the compiler lump numerous error messages into a single category. It is also useful to have a diagnostic summary at the conclusion of each module.

- Reformatters

  A reformatter rearranges the display of a source program. This is useful if previous or present programs are being developed that do not use any standardization in terms of source arrangement.

- Standard Verifications

  Often a department will develop a set of "good programming standards." Included in these standards might be structured programming concepts, variable identifier rules, etc. It is beneficial to management if the compiler system contains facilities for monitoring and reporting any exceptions to the standards.

- Source Listings

  A compiler will usually list the source program it has translated. It is often helpful if the user can optionally obtain an assembly listing expansion of his source. If an assembly listing is provided, then it is extremely beneficial if there is some direct correlation between user defined variables and the variable names generated by the compiler. This makes the assembly code considerably more readable. In addition, such things as

location, object and sequence numbers should be included to aid the programmer in developing/maintaining his or her program. It is also usually beneficial if the assembly listing is interspersed with the compiler source listing. Titles, dates of compilation, programmers name, and similar items should also be provided in the listing.

- Partial Compilations

A partial compilation facility allows the programmer to control the behavior of the compiler. Often a compiler is implemented as multiple passes through a source file (or an intermediate language translation of the source). A developer of a new program may only desire a listing of his source and any syntax errors that exist. It is often possible to accomplish this task with a fraction of the time/cost normally required for a complete compilation.

1.4 Extensibility

A compiler is extensible if it can easily be modified to include new features. In general, a compiler that is implemented using "good programming practices," (structured programming, top-down design, small modules, etc.) is usually readily modifiable for future requirements.

If a new language is being implemented, or an existing language is expected to be modified, it is advisable that the specification include an "open-end" design. The following chart depicts items which are essential to a compiler system's extensibility:

Modularization
Open-End Design
Top-down Design/Structured Programming
Minimal User Restrictions/Compiler Limitations
Parameterization

COMPILER SPECIFICATION - EXTENSIBILITY

- Modularization

The idea of breaking a programming system into small modules not only benefits program checkout and maintenance, but it also makes the compiler easier to modify for future

requirements. A complex compiler system requires numerous functions. It is quite conceivable to find in excess of 150 distinct modules for a well structured compiler system. It is certainly advisable that the specification contain a statement that the compiler consist of multiple modules and that no module should be more than some finite number of statements.

- Open-End Design

When designing a compiler system, data structures are developed which are used to retain such information as variable type, variable dimensions, name scope, error directories, cross references, and literals. In addition, internal translations/parsing of source often build intermediate language forms with character strings replaced by numeric representations.

It is crucial that the compiler designer "leave room" for future entities if extensibility is to be meaningful. For example, assume 3 bits are allotted throughout the intermediate language to designate type of operation. Further assume the following eight entries already exist:

Operator	Binary Representation	Meaning
+	000	Add
-	001	Subtract
*	010	Multiply
/	011	Divide
&	100	Logical And
|	101	Logical Or
¬	110	Logical Not
%	111	Logical Exclusive Or

Now assume that the usage of these three bits is scattered through numerous modules. If it later becomes desirable to add several new operators (eg. exponentiation, relationals, etc.), it might be extremely difficult. This would especially be true if the three bit item was contained in a structure which had no additional or unused space.

- Top-Down Design/Structured Program

  A top-down design usually allows for the simpler understanding of the basic flow of the system. Coupled with structured programming, the basic logic of the system is not only more maintainable but also easier to modify for future requirements.

- Minimal User Restrictions/Compiler Limitations

  Often compilers are implemented with fixed limits in terms of items such as the number of symbols, number of records per source statement, and number of parentheses in an expression. Usually these limitations are a result of poor design techniques and/or inadequate use of the proper data structures.

  Stacks, circular queues, variable length arrays, pointers, and generally well designed data structures will minimize the number of compiler limitations and, hence, user restrictions. Minimization of restrictions is very crucial to compiler extensibility.

- Parameterization

  Parameterization is a method whereby a program may easily be adjusted to accept new program, hardware or system attributes or restraints. It is extremely important that values which define arbitrary limitations be parameterized.

  The compiler implementation language should definitely include a facility for parameterization. The compiler specification should require the usage of this feature. See the following section on transferability for a more detailed discussion of parameterization.

1.5 Transferability

Transferability is a measure of the ease of moving a computer program from one computing environment to another. Transferability with respect to a compiler system can be divided into two major areas: the movement of the compiler to a new host computer, and the generation of object code for a different target computer.

These two areas represent substantially different problem areas, although there is considerable overlap involved with both. If it is intended for the compiler to operate on multiple hosts or more than one target, then the specification should contain a future option requiring the desired task.

A usual judgement of the desireability of any transfer effort or technique is its cost effectiveness. Specifically, the figure of merit is the cost per transferred unit. The transfer is complete when the end-user is as satisfied with the compilers' operation as he was before the transfer.

The following chart and subsequent explanations help depict areas of consideration involving transferability.

Compiler Type and Structure
Implementation Language
Modularity
Parameterization
Code Constraints
Emulators/Simulators

COMPILER SPECIFICATION - TRANSFERABILITY

- Compiler Type and Structure

  A variety of basic compiler structures exists. The type of structure has a tremendous bearing as to future adaptation to new targets.

  Numerous efforts have been initiated to enhance compiler transferability, especially in the area of automatic compiler generation systems. Such concepts as meta-compilers, syntax table driven compilers, and macro backend processors have had only minor successes in the adaptation of a compiler to a new target machine.

  Those that are successful, in terms of inexpensive transfer cost, have almost always proven too costly in other areas, (i.e. poor code generation, extremely long compilation time).

A less popular method involves compilers which generate interpretive code. With interpretive compilers all that is usually required is the implementation of a new set of run time routines. The only major drawback is that programs will execute considerably slower. They will, however, compile faster.

As previously mentioned, it is strongly advisable to modularize the implementation of a compiler system. In addition, a compiler's modules can usually be segmented into the following parts:

- Program Control -- Controls the overall flow of the compiler and calls the proper routines based on the statements being processed.
- Parser--Translates the higher level source into an intermediate language.
- Expression Evaluator--Polishes expressions so that they are more conducive to code generation.
- Optimizer--Scrutinizes the intermediate language to minimize number of required operations.
- Code Generator--Translates the intermediate source to the appropriate instructions of the target machine.

If a compiler is well structured, the only portion which should have to be re-implemented when transferring to a new target is the code generator. If a future target is a major consideration, then the specification should require well structured programs.

- Implementation Language

  Probably the singularly most important factor in determining the degree of transferability to a new host is the implementation language. The following is a brief discussion of several implementation languages.

  - Assembly Language

    This is by far the least transferable (to a new host) of any implementation language. Its major advantage is that the compiler will operate at maximum efficiency (assuming good programming techniques).

  - Macro Language

    Assuming the new computer(s) has a macro processor that facilitates the implementation of the macros used to implement the compiler, then this approach provides a very economical method of moving to a new host. The disadvantages involve the compiler's own translation cost and the fact that macro assemblies are usually very slow.

- Systems Programming Language

  Often compilers are implemented in a special higher level language. Usually these languages are subsets of ALGOL like languages. The cost of transferring to a new host is then usually the cost of rewriting the code generator of the systems programming language.

- Popular Higher Lever Language

  Since nearly all major computer manufacturers support FORTRAN, COBOL, and possibly ALGOL and PL-1, the use of these languages provides major advantages if a new host is of primary importance. In addition, these languages provide other benefits such as ease of transfer of programmers already knowledgeable in the implementation language. The major disadvantage is the inefficiency of the compiler's operation, both in terms of speed and resource utilization.

- Meta Compiler

  There now exist numerous compilers which accept a meta-linguistic definition of a language and produce a compiler system. The major problem that exists in this approach is that almost always the compiler is very poor in resource utilization (space and speed) as well as poor in terms of quality of code generation.

- Modularity

  Modularity is a formal way of dividing a program into a number of subunits. Each subunit should have a well defined function and relationship to the rest of the program. There are well known advantages in modularity: of primary importance is that those program functions which will need the most programmer attention upon transfer can often be isolated and functionally identified as distinct modules. The modules, if organized and documented properly, can be worked on with little reference to, or interference with, the rest of the program.

  A essential aspect of these modules is their isolation from the rest of the program. Each section should be a sequence of programming language statements having a well marked start and end. The function of the module and its interface with the rest of the program should be simple and well documented. A module performing several closely related functions may have several entry points. However, one entry point is usually better practice.

  A compiler system where transferability is a major consideration should specify that the modules which are machine dependent be isolated to as few modules

as possible. The specification should also require appropriate documentation that describes which modules need to be changed.

- Parameterization

  Parameterization is a method by which a source program may self-adjust to a program, hardware, or system modification. Some numeric or symbolic items in the program may require alteration if the same function is to be performed in a new environment. These values may be written into the code explicitly or they may be parameterized. In the latter case, symbolic variables are set to the applicable values in an initialization phase of the assembly/compilation. The values chosen may be directly initialized by programmer coding or they may be set up or calculated by the language processor.

  If the value is written in by the programmer, it should be in a well marked statement, with all such statements collected in one place. The documentation for each value should be explicit in how and when a new value is to be determined for the parameter. If the value is to be computed, the computation must be checked out for a range of values.

Although a transferred program usually performs an unaltered function on the new machine, parameterization may be used even more extensively when the program function is altered during transfer.

Parameterization requirements depend on the details of the programs involved. Each actual value in the source code must be considered for parameterization as it is being written. If the value and the representation of that value are independent of any reasonable change in machine applications, scope, or language, or if the programming language does not permit a symbolic value in that context, then parameterization does not apply.

In general, over-parameterization is rarely a problem, especially in the assembly or compilation process. Thus, the cost of parameterization is nontrivial only if the logic of the program or usage of the language must be perverted. The value of parameterization, on the other hand, can be considerable.

- Code Constraints

  Program transferability can be greatly enhanced by avoiding code that is difficult to transfer. Finding and using alternative code, however, does involve

additional programming costs which must be measured against the benefits obtained.

For higher level languages, the primary code constraints of concern involve the avoidance of language features which are apt to be missing or altered, or which will give different results when compiled on the transfer target.

The language features most likely to be missing or different in other compilers are the language extensions and the expensive or little-used standard features. Unless a language was specifically extended to suit an application, use of extensions can be avoided with only a limited loss of efficiency. In those cases where the standard language is inadequate, the unusual usages should be isolated in modules for recode.

Aside from subsets and extensions, two things in general make code difficult to transfer - techniques that bind to a particular representation of data structures and instructions, and those that bind to a particular sequence of operations. A binding technique, then, is any usage that takes advantage of, or is cognizant of its own implementation. When any portion of any instruction is used as an operand, or when any address

is used which is not completely symbolic, a binding technique is being used. Program code may be cognizant of operation sequences because they define a function which must be reproduced when the program is transferred.

Program code which modifies itself has proven to be bad practice for reasons of maintenance difficulty, error proneness, and non-re-entrant properties. Recent machines and languages have taken steps to render self-modifying code both unnecessary (e.g. execute instructions) and difficult or impossible (e.g. write protect). Higher level language code does not directly allow dependencies upon instruction representation.

Generally, unless the machine is deficient in indexing and other dynamic address functions, representation-dependent code may be easily avoided. Furthermore, resource conservation realized through use of this type of code usually amounts to only an insignificant percentage.

For any particular machine, external data representations correspond to predictable internal storage bit patterns. It is seldom the case that those representations will be equivalent over a variety of different machines. If the external representation used does not correspond

to the function of the item, the internal representation will become inappropriate upon transfer. An example of this type of code is character identification by comparison with small decimal integers.

Similarly, packing several data items in a word will bind the code with respect to word size. A data item may be split by a word boundary upon transfer. For example, 3 items of 6 bits each do not repack well into a 16 bit word.

Operation sequence and data representation constraints can also be combined in ways which may seem of great value but which definitely hinder transferability.

A program may be dependent upon aspects of system configuration in ways that make it infeasible to transfer the program to a system in which those aspects differ significantly.

Those system attributes which become severe constraints are usually either system resources which are heavily used or devices which provide facilities not qualitatively present on the new system.

Primary storage is likely to be the constraint most often felt. If a compiler is written for a machine

with a large amount of core storage, it may be infeasible to move to a machine with less storage unless the program was initially written with this in mind. Similarly, compilers often do not adjust well to an overlay/nonoverlay transition since basic algorithms may differ (overlay processing often requires a file pass for each phase, where this would be unnecessary with adequate primary storage). On the other hand, large primary storage may have been crucial to efficient operation of the program, and it would be wasteful not to use it.

One approach to this problem, then, is to consider (and document) primary storage size as a system constraint precluding code and perhaps design for transfer to a machine with less storage.

Similar considerations apply to random versus sequential access secondary storage devices.

In the area of input/output, efforts must be directed toward avoidance of explicit references to physical devices which can permanently bind a program to a particular configuration. Programmers should design and code with logical entities such as the system input device (rather than the card reader) or the

object output file (rather than the tape punch). Physical devices and their associated characteristics (such as record length) must be parameterized and isolated from the main body of the program as much as possible.

- Emulator/Simulator

  The development of an emulator or simulator that operates on one host and emulates another is another means of transfering a compiler to a new host. This method is almost always an unlikely alternative considering the major disadvantage of excessive resource utilization on the new host.

1.6 Environment

The operating environment or computer system must be well defined before compiler development begins. Future systems are discussed in Section 1.5. The initial environmental specification should be directed toward two operational levels - minimum and capacity. The minimal configuration for processing "average" programs one at a time may be a milestone in development and payment. The capacity consideration is the largest or most (multi-programming) compilation(s) which are expected in the application program developmental cycle.

This information affects several areas:

- Hardware acquisition and selection
- Compiler implementation
- Compiler transfer
- User base and profile
- Choice of operating system
- Programming capacity-size of source programs

A procuring agency which over estimates available core storage, for example, may severly limit the size of source programs and the number of users who may run simultaneously.

Further areas of consideration are covered in Section 1.1, Resource Acquisition, and in Section 1.7, System Interface.

1.7 System Interface

The ease with which a user can access a compiler file, link compiler object modules, and re-run compilations is of paramount importance in the specification of a compiler system. No tool, no matter how good, is really useful unless it can be satisfactorily applied within the system.

Although some aspects of this problem seem self evident, they should not be left to the imagination of the vendor. The specification should clearly state the desired system interfaces. The following chart and subsequent descriptions will provide representative specifications for system interface.

Operating System Interface
Job Control Language Requirements
Interface with System Text Editor
Object Language
Interface with Other Languages
Subroutine Linkages
Resource Usage

COMPILER SPECIFICATION - SYSTEM INTERFACE

- Operating System Interface

  A compiler interfaces with the operating system for the following major tasks:

  - To be given control to execute
  - To attain its share of resources
  - To receive user input files
  - To transmit user output files.

  It is advisable that a compiler's specification includes definitions of operating system tasks to be used and means by which it is to perform the above tasks. It is also advantageous for the interface to the operating system to be centrally located in the minimal number of modules. For example, it is quite possible to have four modules for each of the four functions above. Arguments to the modules can be used for determining the necessary tasks to be performed, as shown below:

  - Subroutine Name: Control

    Function: Entry point of compiler. Also includes exit to operating system and any necessary interface for attaining or presetting system information (e.g. obtaining date of compilation, etc.).

- Subroutine Name Resources:

  Function: Obtain internal storage requirements for compiler.

  Inputs: Amount of space required.

- Subroutine Receive/Transmit

  Function: Obtain/Output a record from/to a user file.

  Inputs: File name or number, buffer to receive/transmit record, size of record, mode (ASCII, EBCDIC, Binary, etc.), Operation (Normal, rewind, end of file, etc.)

In order to insure the desired operating system interface, the compiler specification should contain detailed descriptions of the above interfaces.

- JCL Requirements

Nearly all modern operating systems require a control language specification (job control) prior to job submission. The "JCL" describes the programs to be executed and the environment for execution. It is crucial that minimal job control and knowledge of operating system idiosyncracies be required for a compilation.

A user should not have to become a JCL expert to have his program translated. Usually, operating systems will allow the invocation of JCL from procedure libraries, thereby allowing a user the capability of minimizing his JCL requirements. It is reasonable to include in the specification that JCL for all typical compilations/executions be provided to simplify the user's interface with the compiler.

- Interface with the System Text Editor

  If the compiler is part of a system which contains a text editor, then the compiler source input files should be in a format that can be used by the text editors. Often compilers will encode the source, thereby rendering any system text editors useless.

- Object Language

  A language processor translates a source program to a form that is readily loadable into a computer. This form is called an object language. An existing system will usually contain loaders and linkage editors that process the object language. The compiler specification should stipulate that the object language be compatible with the existing system's object language.

- Interface with Other Languages

  If it is deemed desirable to implement a system using more than one language, it should be made part of the specification. Compatibility among data structures, object languages, and file structures will enable the use of additional system language processors. This is especially important with respect to existing assemblers.

- Subroutine Linkages

  Usually a system will contain a standard procedure for calling a subroutine. This will include both the activation of the subroutine and the passing of arguments. If such a convention exists, it is strongly advisable that the specification include this as a requirement. If no convention exists (or if the standard convention is deemed not worthy of consideration), then the compiler specification should require a consistent method be used throughout the compiler.

- Resource Usage

  Resource usage can be considered in two basic environments, compile time and execution time. There is often a correlation between expensive resource

utilization during compilation of a source program and economical execution of the source program, or vice versa. Modern software technology coupled with usually available system software facilities (e.g. overlay loaders, selective loading of object modules) enables compiler systems to minimize their resource requirements.

Usually there is no major benefit in having the entire compiler core resident during compilation. Source compression technology, with efficient usage of data and file structures, allows large compiler systems to operate efficiently without usurping vast resources from the system. If left unspecified, the procuring agency may find their compiler using enormous amounts of memory and resources. Thereby the compiler system becomes virtually a stand alone operation.

1.8 User Profile

The number and types of users of a compiler can have large effects on its efficiency and performance rating. Compilers designed to meet a wide range of source program sizes, for example, may not produce the most efficient code or may require extensive link-loader processing. A one-pass incremental system may compile quickly but execute slowly. A multi-pass, optimized, compiler may compile slowly and execute rapidly, etc.

Similarly, a compiler which is highly optimized and capable of handling large source files is not efficient for a tutorial situation requiring fast response and many users.

The procuring agency should identify:

- the prospective set of users
- programs
- experience levels
- percentage of tutorial or "fun" compilations (as in a university)
- life expectancy or "production" percentages
- real-time requirements
- number of simultaneous users
- desired response times

The user profile specified should provide the basis for acceptance tests. Those tests which most closely resemble the user profile should compile the most efficiently.

1.9 Documentation

Documentation is vital to the successful use and maintenance of any compiler system. A compiler system without adequate documentation is worthless. A compiler system's documentation can be divided into two parts, internal (maintenance) and external (user-oriented).

Any compiler specification should include requirements for sufficient documentation in both areas. The following chart and subsequent descriptions will help provide several insights into the types of documentation that can be provided.

User Manual
Job Setup
Primer
Training Material
Compiler Limitations/Idiosyncracies
Diagnostic Description
Syntactical Language Description
User Flow Charter
Compiler Source Listings
Flow Charts/Prosaic Description/ Variable Descriptions
System Generation
System Interface

COMPILER SPECIFICATION - DOCUMENTATION

- User Manual

  Every compiler specification should require a user manual. The users' manual should describe the usage of all compiler features as well as examples illustrating their use.

- Job Setup Description

  Often this can appear as part of a user manual (usually an appendix). This describes how to compile and/or execute programs. It should clearly specify each step and the entire range of job submittals.

- Primer

  If the language is new, or if it contains different concepts, or if it is to be used by relatively inexperienced programmers, it is beneficial to have a primer specifically describing these attributes.

- Training Material

  In addition to primers, it is beneficial to develop visual training aids and sample decks or terminal inputs for a new compiler system.

- Compiler Limitation/Idiosyncracies

  This item often appears as an appendix. Although essential, it is too often missing from the available compiler documentation or is scattered throughout a variety of manuals. It is vital that all compiler limitations/idiosyncracies be contained in an easily accessible document.

- Diagnostic Description

  All error messages should be contained in one contiguous document (or a user manual appendix). The diagnostics should include details of how and why each error occurs and, if appropriate, what action is to be taken.

- Syntactical Language Description

  This should be part of the compiler specification and also be available to users. As mentioned previously, a meta-linguistic notation (g. Backus Nauer Format) along with a prosaic descri ion should be used to specify all possible language constructs.

- User Flow Charter

  The inclusion of a program which will automatically

generate flow charts for application programs may be beneficial to users. The costs of this item must be weighed against the potential benefits. Usually automated flow charts of higher level languages provide little more than a slightly better pictorial arrangement of the program flow.

- Compiler Source Listings

  It is essential that the compiler implementors provide well-commented, well-structured source listings if the procuring agency expects any type of reasonable maintenance or understanding of the programs. All good programming habits, as enumerated by recent articles (topics including structured programming, top-down design, modularization, etc.) are beneficial to the individuals responsible for maintaining or modifying the compiler.

- Flow Charts/Prosaic Description/Variable Description

  The value of technical documentation is extremely difficult to measure. Brief synopsis of the various compiler elements along with flow charts of complex algorithms are usually helpful to the maintenance programmer. It is essential that a technical document

be provided if the compiler source programs do not contain sufficient comments describing every variable.

- System Generation

  It is essential that a document be provided that illustrates the methodology for symbolic modifications to the compiler and then made to be operational as part of a new system. The specifications should always include this as part of the required technical documentation.

- System Interface

  It is also important that technical documentation exists as to what system facilities are used, how they are used, and where they are invoked. Often a new system facility will replace an existing facility. Unless documentation is provided, the maintenance/modification to support such changes becomes extremely complex and costly.

1.10 Schedule

The compiler specification RFP and vendor supplied proposals should both contain an implementation schedule with provisions for milestone payments. The actual schedule time requirements vary tremendously with language, compiler efficiency, implementation strategy and hardware constraints. Large scale compilers such as PL/1 may require more than one linear year to initial completion. Less efficient implementations of FORTRAN and BASIC may require only a few man months. The actual schedule will be a judgement based on choice of vendor, language and options. Typical payment/acceptance milestones in development can be identified, however.

- Compiler design - detailed written descriptions of compiler modules, intermediate languages, object languages, parsing and translation algorithms to be used. This is the basis for compiler documentation and should be done first.
- Parsing and Translation - a demonstration of successful generation of intermediate language from a test source program.
- Code generation - demonstration of programs to convert intermediate languages to relocatable or loadable object forms.

- Initial Configuration - a first stage compiler capable of processing simple language forms to un-optimized link or loadable object.
- Optimizer - A demonstration of compiled code which has been optimized and executed successfully.
- Self-compilation - if applicable--complete compilation and optimization of the compiler programs and subsequent use of the compiled object to run additional tests.
- Performance Tests - compiler performance acceptance tests.
- Documentation Package
- Training
- Agency Trial Period - before final payments, the user organizations should make exhausted use of the compiler and critique its performance.
- Compiler Modification - as a result of user tests, enhancements and alterations may be required for maximum use and efficiency.
- Maintenance Period - final developmental payment should be made before the beginning of the maintenance period. Maintenance may be defined as correction of bugs not found in previous testing. Changes to the specifications and enhancements are not maintenance items.

1.11 Acceptance Tests

A vital part of any compiler specification is the acceptance test. Usually the compiler implementor will generate a set of test cases during the development phase and the procuring agency will nod their tacit approval as acceptance tests. Far too often these tests are later found to be incomplete and subsequent usages of the compiler cause the discovery of numerous problems.

Several existing languages have a set of tools that aid in the validation of a compiler. The quality and comprehensiveness of the acceptance tests are extremely important in the initial acceptance of a compiler.

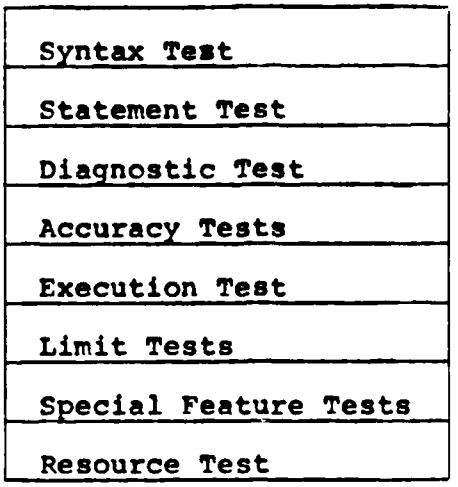
With the increasing complexity of the new compiler systems, it is necessary that acceptance tests proceed beyond a mere superficial state of initial debugging. The desire to "just get it working" has prevailed in so many compiler development efforts, that the initially delivered product has often proven to be a future pandora's box.

The compiler specification should include statements specifying the use of automatic program testing tools, if available. If not, then the following chart lists some of the tests which should be provided. It is usually

advisable that the acceptance tests be developed by the procuring agency shortly after the design phase of the compiler.

A great need exists for structured methods in the testing of a compiler system. It is impossible to test all possible paths in a complex compiler system. For example, assume a compiler system has 2^{54} different paths (a very small number with respect to a fairly complex compiler system). Using a computer with an internal speed of 1 microsecond, it might seem reasonable to verify all possible paths, while in actuality, approximately 1000 years of computer time would be required.

The following chart lists those items which should be included as part of the acceptance tests. The development of test cases for each of these items should be made part of any compiler specification in order to assure a reasonable degree of reliability for a new compiler system.



COMPILER SPECIFICATION--ACCEPTANCE TESTS

- Syntax Test

  A set of test modules that verifies the compiler's grammar and handles all legal syntax structures.

- Statement Test

  A set of test cases that verifies the proper translation for each possible statement, and their use in conjunction with other statements.

- Diagnostic

  A set of test cases which generates every known compiler error.. If an error can be created for a variety of circumstances, then each possible cause should be made a part of the test.

- Accuracy Test

  A set of test cases to verify the accuracy of compiler specified mathematical operations, functions, etc. This includes tests that convert from one numeric mode to another (eg. integer to floating to integer, etc.).

- Execution Test

  A set of test cases that verify that the compiler system translates all possible statements to executable code.

- Limit Test

  It is essential that a set of test cases be developed which check all compiler limits. For example, a test case that includes numerous identifiers, extremely long symbol names, maximum parentheses, maximum loop nesting, maximum continuation, etc. Often these test cases are ignored and prove to be future bottlenecks.

- Special Features Test

  A set of test cases to validate the proper operation of all special features should be required and so stated in the specification.

- Resource Usage

  A "typical application program" should be developed to measure resource usage. This includes both compile and execution time resources. Often this test case will have to include special hooks into the compiler so that statistics (time, space, I/O accesses) may be properly recorded. This test case can be used to verify that the maximum specification requirements (resource usage) are not exceeded.

1.12 Maintenance and Support

The acquisition of a compiler system should include plans for its continual maintenance and support. Normally a vendor will maintain a software product at no cost to the procuring agency for a jointly specified time frame after product acceptance. As mentioned previously, even if the acceptance test cases are implemented with extreme care, it is impossible to verify all paths of logic within a complex compiler system. Therefore, the specification should include plans for product maintenance throughout the expected life of the compiler system.

Numerous programmers have used an existing "field tested" compiler only to encounter problems after five to ten years of compiler usage. The complexity of a compiler system, and its critical importance as a software tool warrants future maintenance requirements be established at the time of specification.

As new problems are encountered in a compiler, it is advisable to modify the acceptance test to include these problems. This will tend to insure error free future versions.

The utility of a compiler system is determined, in part, by the ease with which it may be used, corrected, modified, and updated. The following descriptions of the items in the chart below enumerate the requirements for effective maintenance of a compiler system.

Technical Documentation
Debug Tools
Skilled Programmers
Warranty Contract

COMPILER SPECIFICATION - MAINTENANCE

- Technical Documentation

It is vital to any maintenance effort that efficient technical documentation exist. Since the source program for the compiler is the only guaranteed actual program logic, a well structured source is the best assurance of future maintenance.

Small modules, meaningful variable names, and sufficient commentary to explain algorithms and routine functions are a prerequisite to a maintainable system.

- Debug Tools

  When problems arise, it is almost imperative that debug aids exist to help isolate and eventually correct the problem.

- Skilled Programmers

  Usually the implementors of compilers are well versed in the technical requirement of developing language processing systems. Often junior level people are hired to maintain a system. The specification writer should consider the level of maintenance required when deciding on or accepting a set of individuals to maintain the compiler.

- Warranty Contract

  It is almost always advisable to have the company personnel that implemented the compiler be responsible for its warranty. If the warranty is to exist over an extended period of years, as if often the case, the specification should attempt to have more than one individual cognizant of the internal workings of the compiler system. In addition, the agency which has the maintenance responsibility should be held responsible for the transfer of knowledge if the individuals are transferred elsewhere.

The specification should state the response time for the correction of problems. In addition, a standard method of reporting problems should be adopted. Periodic status reports should be made available to users whenever new problems arise or are corrected.

Part 2 COMPILER ACCEPTANCE

The development of an objective decision-making process when accepting a compiler is extremely important in insuring a successful compiler system. Part 1 provided a means of identifying the features and facilities available for a compiler system and for their reasonable specification to the vending organization. The specification charts developed in Part 1 also provide a meaningful basis for development of acceptance criteria. In reality, the weighting factors or scores introduced in this section are only meaningful in terms of the importance or desireability of items specified in Part 1.

The approach taken in this guideline is to change numerous qualitative evaluations into quantitative scores with weights provided for each category. A summation of the evaluation provides a compiler score. The score is then used as an index into an acceptability chart. Since this guidebook was not developed with any particular language or user group in mind, the relative weighting scores assigned are based on a hypothetical compiler system.

The representative acceptability chart below is based on a perfect score of 1000 points.

COMPILER ACCEPTANCE CHART

Score	Evaluation
950-1000	Excellent
850-949	Satisfactory
750-849	Temporarily Acceptable
0-749	Completely Unacceptable

It seems advisable that future compiler system contracts include, as part of the payment, an amount based on the acceptability score. This additional financial remuneration could provide the incentive needed to turn acceptable and good compilers into excellent compiler systems.

In the following sample compiler acceptance matrix, the acceptance of a compiler is divided into ten parts. Each part is given an empirically derived total point potential.

The values specified below should be modified to the particular requirements of the compiler system being developed. The matrix shows each of the major categories, potential and eventual actual scores. If certain categories are not applicable to the compiler systems acceptability, then the total point potential of the remaining categories should be modified accordingly.

Compiler Category	Potential Score	Acceptable Score	Actual Score
Accuracy and Reliability	400	360	
Resource Utilization	100	80	
User Interface	50	40	
Documentation	100	80	
System Interfaces	50	40	
Options	70	60	
Extensibility	30	25	
Transferability	100	80	
Schedule and Installation	75	65	
User Profile Adherence	25	20	
TOTAL	1000	850	

COMPILER ACCEPTANCE MATRIX

Topics or performance items which do not meet the acceptable score must be re-worked and evaluated until the minimum acceptable score is attained for that item.

The following pages further delineate each of the major categories when accepting a compiler system. Each major category is broken down into a sub-matrix.

2.1 Accuracy and Reliability

The speed of compilation, minimization of system resources and superlative user options are of little value unless the compiler is able to generate valid object code for higher level source statements. The attributes of accuracy and reliability are best judged by acceptance test program runs that are compared with expected results.

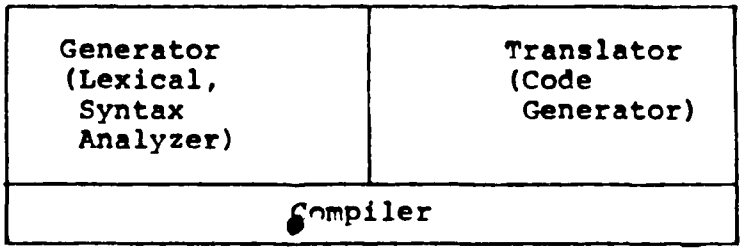
There are several attributes of a more general nature which can be checked for in a compiler that affect the overall system's accuracy and reliability. The following matrix depicts the items used to measure a compiler's accuracy and reliability. This matrix is organized by the features and structure of languages and compilers.

Category	Potential Score	Acceptable Score	Actual Score
Compilers Internal Structure	100	90	
Syntax and Semantics Analyzers	25	23	
Assignment Statements	75	70	
Declarative and Allocation Statements	50	45	
Control Statements	25	23	
Subroutine Statements	25	23	
Input/Output Statements	25	23	
Limits	25	20	
Documentation/Diagnostics	50	43	
TOTAL	400	360	

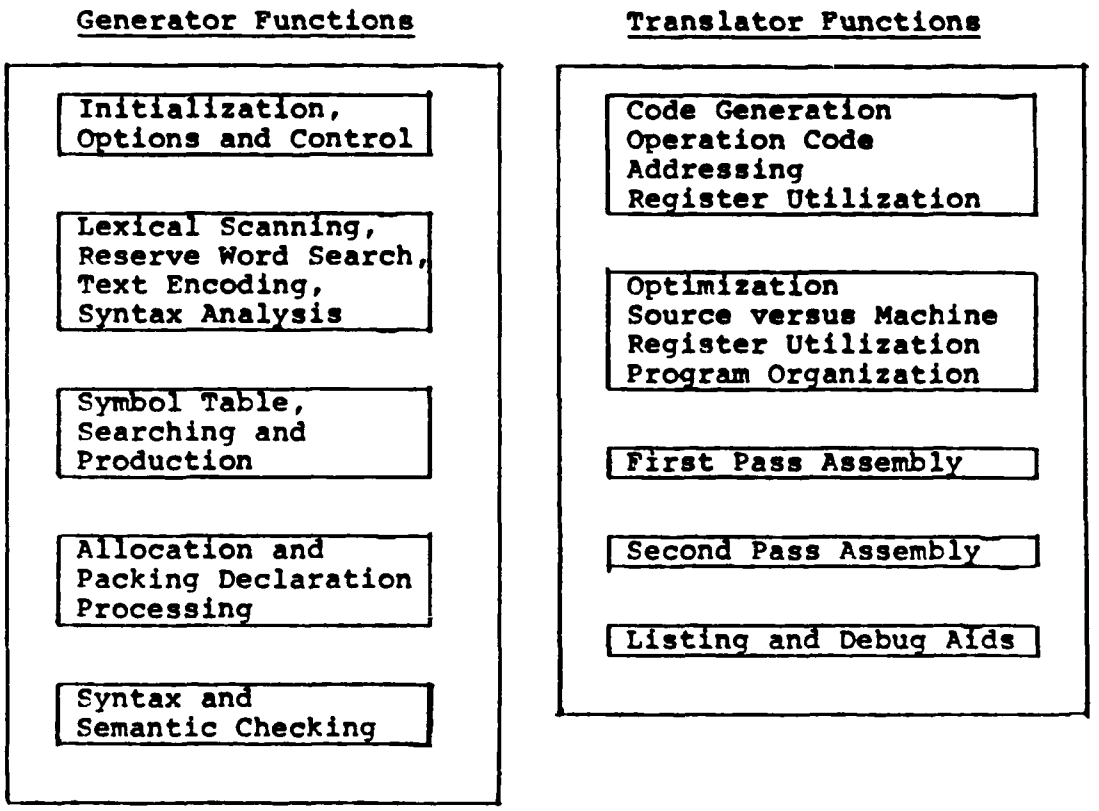
COMPILER ACCEPTANCE - ACCURACY AND RELIABILITY

- Compiler Internal Structure

A compiler can be portrayed as comprising two major processing functions.



These compiler elements likewise reflect the five basic factors which effect compiler reliability:



These compiler functions directly relate to hardware, operating system facilities, language and the implementation techniques employed.

Although this representation is somewhat simplified or general in nature, it does provide insight into the inherent opportunities for modularization. Modularization is the single most important factor in predicting reliability and ease of maintenance.

A compiler implemented using structured programming techniques is also a candidate for high reliability scores.

In general, then, if a particular set of acceptance test cases are successfully executed and the compiler is both modular and structurally programmed, there is a high probability that similar programs will be valid.

Structured programming and modularization and both somewhat subjective concepts to measure. Because of this, a visual inspection of listings and internal structure flow charts may be necessary. Some items of interest in such an inspection include:

- parameterization
- meaningful source commenting
- open-endedness
- size of sub-programs and dependence on complex scope relationships
- simplicity of linkages and arguments passed between modules
- indirect addressing and deeply nested pointer schemes
- degree of isolation of target dependent modules
- re-entrancy

- recursion
- host dependent coding

- **Syntax and Semantics Analyzers**

  Basic components of any compiler are the syntax and semantics analyzers, sometimes called the parsing algorithm. The SOFTECH report and other recognized studies have indicated that the choice of parsing algorithms is not as crucial to reliability as is its implementation technique. Items to include in tests of this area are:

  - scanning - number of blanks, multiple operators, deeply nested syntax
  - spelling - use of similar but not equal names such as abbreviations, extensions, reversed letters
  - searching - speed of search as a function of label size, number of labels
  - recovery procedures - skipping erroneous constructs, pinpointing errors
  - addressing errors - out of range or undefined labels flagged before "assembly"
  - mode definitions - default variable classifications
  - uninitialized variables

- scaling and conversion anomalies - infinite and 0 results predictable at compile time
- bit packing
- nesting levels
- constant conversion - artificial or host induced magnitude and accuracy constraints

- Assignment Statements

  Assignment statements are given a high relative weighting in determining accuracy and reliability. Although the definition of assignment infers simply computing and storing a value, almost all of the compilers' functions are tested:

  - parsing
  - evaluation of expressions
  - searching
  - data conversion
  - function linkage
  - variable initialization
  - fetch and store
  - optimization
  - parallel, serial, array addressing
  - code generation
  - cross-reference and user aids

In particular, in languages allowing multiple assignment, a simple test may detect complex errors. For example:

I = signed item

J = array of floating point items

J(I),I = (I+1)/J(I)

may be defined as:

$(I_0+1)/J(I_0) \rightarrow I_1$

$(I_0+1)/J(I_0) \rightarrow J(I_0)$

or as:

$(I_0+1)/J(I_0) \rightarrow I_1$

$(I_1+1)/J(I_1) \rightarrow J(I_1)$

or other possibilities dependent on language definition, and compiler analization techniques, and data conversion algorithms.

Sample assignment statements can and should be easily hand checked for expected results.

- Declarative Statements

Declarative statements exercise parsing, searching, allocation, initialization, subroutine linkage, addressing structure and routines. Items which can be best checked include:

- declarative statement syntax
- mode, value, or array structure assumptions

- levels of subscripting-indexing
- parallel structure
- packing
- data conversion
- interfaces to assembly/higher languages
- visability of variable placements
- overlay and equivalence
- continuation statements
- statement sizes
- addressing range and virtual locations

- Subroutine Statements

  Subroutine or subprogram statements are particularly important for checking scope, modularity and addressing. Subprograms should be compiled separately and also in-line to verify language rules.

  - Addressing techniques may become cumbersome or inefficient in separately compiled structures.
  - Identically named variables may be erroneously addressed or set in nested subroutines.
  - High usage core, such as directly addressed pages in base register machines, may be easily over-loaded as the number of routines and/or arguments increases.

- Linkages or register contents should be maintained.
- Re-entrances or recursion may not exist.
- Artificial calling linkages or nesting limits may exist.
- Registers may be unused or unavailable causing inefficient code.

- Control Statements

  These statements control program flow and sequence of execution. Tests in this area should include:

  - loop control and exiting
  - subroutine calls and returns
  - subroutine control arguments
  - computed transfer ranges, defaults, error detection and data conversion
  - user warnings
  - parallel and embedded scope
  - BEGIN-END structures
  - conditionals: IF, THEN, ELSE, WHILE
  - addressing range and technique
  - instruction set and condition code usage

- Input/Output

  There are several major areas of reliability which are of concern when verifying input/output statement compilations.

These include:

- Timing dependencies - computations may be improperly made if instruction sequence execution proceeds during input/output. Similarly, execution times may increase if wait states occur unnecessarily. Data may be lost or filled during hardware malfunctions.
- Machine independence - the modularization of I/O routines, pointers and run-time package linkages should be verified.
- Artificial coding restraints - source language usage should not be tied to record sizes or block control flags unless specified in the language.
- Parity and data checks should be verified to insure integrity of the source program computations.
- Format statements and similar input/output structures should be tested for data conversion, packing and error checking capabilities.

- Limits

Internal compiler structure limitations can be tested via several of the statement types previously listed. Of particular and common interest are ranges of values for:

- compiled program array dimensions and execution time index values which are out of range - default conditions, error recovery, compile time checks, maximum values
- label sizes
- character constant sizes
- record sizes for I/O
- array dimensionality maximums and the relationship of dimensionality to object code efficiency
- FOR-DO loop nesting limitations
- FOR-DO loop scope limitations
- nesting of array indices
- nesting of subprogram function calls
- nesting of IF THEN ELSE structures
- number of subprograms
- number of continuation lines and characters in a statement
- maximum program (loadable)
- maximum number of items, tables, arrays, symbols
- dynamic table re-allocation (when a limit is reached)
- operator/operand sequences
- nested parentheses
- COMMON or COMPOOL sizes

- Documentation and Diagnostics

  Accuracy and reliability are also important in system and user documentation. The results of previous language constructs and compiler implementation tests should have been predicted in the vendor documentation.

  This is particularly true of limit tests, data conversion anomalies and input/output characteristics. If vendor documentation does not contain a high percentage of correctly predicted limits and anomalies, then it is probable that the vendor has not performed adequate reliability and accuracy tests.

  Similarly, re-hosting or re-targeting documents should be randomly checked for accuracy. If a routine is documented as having a particular function, it should be checked by visual inspection of the code and by trace and dump techniques.

  All error messages should be purposely generated by test and compared to user documentation. Again, if the documents do not reflect the actual results, some feeling for compiler reliability may be surmised.

  The following list is a representative sample of conditions which should be included or checked in acceptance tests.

The language and computer system chosen are paramount to specification of specific accuracy tests. The vendor should prepare these tests using at least the following:

- existing compiler or hand-calculated benchmark results
- misspelled names
- duplicate names
- mode definitions
- uninitialized variables
- absent terminators
- bad nests of all types
- improper variable modes (floating point loop control)
- recursion
- improper data conversions
- GO's and SWITCH's on bad values
- constant redefinition:

```
CALL SUB(5)
   :
SUB(X)
   :
X=X+1
   :
```

- values of FOR loop variables outside the loop
- magnitude of loop variables (use large values, negative values, etc.)

- bad patches
- erroneous packing specifications
- improper semantics
- multiple, undefined operator sequencies: X=Y++1
- user un-optimized code
- compiler "private" symbols and characters
- floating point underflow/overflow
- array overflow/underflow
- undefined variables
- misplaced parameters (TABLES)
- incorrect data types: FLOAT (FLOAT(
- assembly language subprograms

2.2 Resource Utilization

Efficiency of the compilation process and of compiler produced object code can be viewed in terms of use of available resources. These resources might include:

Category	Potential Score	Acceptable Score	Actual Score
CPU processors (time)	40	30	
Core storage	10	8	
Instruction set	20	18	
Peripheral devices	5	4	
Registers & Addressing	10	8	
Programmer Productivity	10	8	
Numeric Accuracy	5	4	
TOTAL	100	80	

COMPILER RESOURCE UTILIZATION

Acceptance tests prepared for the purpose of judging compiler resource utilization must somehow be compared to known standards. The comparison of one compiler to another is often meaningful only if both show the same pool of resources. Otherwise, it is the configuration of hardware that is being judged and not necessarily the compiler software.

Similarly, both compilers would be required to solve the same compilation problems - parsing, evaluation, optimization, and code generation. If such a comparison compiler is available - same language, host and target, the new compiler may be compared exactly on a test by test basis.

Each compiler may excel or falter in a particular test case situation. Therefore, a compiler based comparison will, at best, determine the "efficiency" of the new compiler only in terms of the type of test. Compiler A may be efficient for small subprograms while Compiler B may do better in large program situations.

It appears that the only reliable benchmark for resource utilization comparisons is an assembly language implementation of the same application program.

The area most likely to be deficient in benchmark comparisons is instruction set utilization. For example, the statement ALPH=ALPH + 1 often produces

```
        LOAD  ALPH
        ADD   #ONE
        STORE ALPH
#ONE    DATA  1
```

This utilizes four memory locations and three instruction "cycles". If the target computer contained an increment or memory-add instruction, the code could be reduced to:

```
        INCREMENT  ALPH
or      LOAD       1
        MADD       ALPH
```

At this point, it should be stressed that computer design can have a great deal of influence in compiler speeds and core storage requirements. Certain aspects of computer architecture are especially important to language processor design and performance. In particular:

- Direct Addressing Capability

  The computer should have the ability to readily address and access information and data consistent with the system core sizing requirement. For example, if the system core requirements were 4096 words, 12 bits of a computer word would be necessary to effect direct addressing access; if 32,767 words were required, then 15 bits would be necessary, etc.

- Multiple Purpose Register Facility

  The computer should have a set of multiple registers which can be utilized and manipulated in a variety

of different fashions. Several functional characteristic of these registers are that they should function as:

a. Accumulators

b. Index registers

c. Masking registers

d. Control registers

- Partial Word Accessing and Utilization Facility
  The computer should have the ability to fetch, store, test, position and manipulate variable length contiguous portions of a computer word. Associated with this capability must be the ability to treat the data as signed or unsigned variable length information when involved with the particular operations.
- Conditional Testing and Branching Capability
  The computer should have the full capability of supporting comparative relational conditions such as equal, not equal, less than, less than or equal, greater than, and greater than or equal. This capability must ideally provide for register-register, register-memory and memory-memory comparative facility.

- **Arithmetic Computational Capability**

  The computer should have the capability to support the full computational capabilities of addition, subtraction, division, multiplication and negation. This facility should provide for full and partial word operations.

- **Shift Operation Capabilities**

  The computer must have the capability to support both algebraic and logical shifts in two directions on individual multiple purpose registers and between multiple purpose registers.

- **Logical Operation Capabilities**

  The computer should have the capability to support the logical operations: AND, OR, NOT, exclusive OR, EQUIVALENCE, etc.

- **Special Instruction Capabilities**

  The computer should have a group of special purpose instructions designed to accommodate frequently occurring programming situations in the particular application system. These include the following types of commands:

  a. Execute
  b. Store zero
  c. Transfer on register or memory zero or non-zero.

d. Transfer on register or memory negative

e. Halt

f. No operation

g. Test, set, reset, test & set, test & reset

h. Increment/decrement

If the above features are not avoidable, efficiency ratings must be adjusted to reflect computer inadequacies.

Once a set of assembly language benchmarks has been prepared, the following statistics may be gathered:

- compilation time = compilation computer time/ assembly computer time
- programmer productivity = source program preparation cost/assembly program preparation cost
- object storage = compiled program size/assembly program size
- object time = execution time-compiled/execution time/assembled
- numeric accuracy = precision of compiled program results/precision of assembled program results
- compilation storage = total core storage for compilation/assembly
- overhead time = minimum cpu time necessary for void or "nothing" programs-compilation/assembly

- overhead storage = minimum size of object programs
- capacity = maximum source program size-higher level/ assembly level
- peripheral = devices used (tape, disc, drum) and number of accesses

Many of the statistics listed can best be obtained using built-in performance measurement facilities. A comprehensive system for compilation/execution measurement statistics gathering should be included in every compiler procured. Their cost is minimal and their value in programmer aid and compiler "tuning" is considerable.

These statistics gathering facilities can be and should be capable of producing size figures by statement type for:

- declaratives
- assignments
- data conversions
- loop controls
- structured controls (IF,GOTO, etc.)
- input/output
- modularity constructs (subroutine linkages)

The figures of merit in these instances include:

- compilation time

- compilation space
- register utilization in the object
- machine features used - indexing, indirect addressing, pointers, overlays, instruction set, hardware stacks
- execution time
- execution space
- I/O time
- I/O wait states
- disk/drum/tape accesses
- record sizes
- data region size
- bit size - packing (full word?)
- number of compilations/assemblies to working programs

The resource utilization sub-matrix scores are computed from the statistics gathered and compared to empirical (desired or experienced) values. The actual assignment of scores depends upon the statistics available and the weight given to each attribute.

RESOURCE UTILIZATION STATISTICS

CATEGORY	INPUT ATTRIBUTES
CPU Usage	Compilation time Object time Overhead time I/O wait time
Core Storage	Compilation storage Object storage Overhead storage Capacity Bit size - packing Data region size
Instruction Set	Instruction frequency counts Special instruction usage (increment, memory add, etc.)
Peripheral Devices	I/O times I/O accesses by type (disk, drum, tape, etc.) Record sizes
Registers	Register usage frequency counts Indexing Pointers Stacks

CATEGORY (continued)	INPUT ATTRIBUTES
Addressing	Direct Indirect Pointers Indexed Stacks Sub-words
Programmer Productivity	Number of compilations/ assemblies per program, Statistics available on language usage, Statistics available on statement speed and bottlenecks, Time to prepare programs.
Numeric Accuracy	Precision of program results. Number of data conversions. Use of special registers (floating, double, etc.).

2.3 User Interface

The acceptability of the compiler - user interface is not as quantitative in nature as accuracy or resource utilization. However, there are several important aspects of the user interface which can be identified. These include:

- Partial compilation facilities
  - optimization levels
  - guided optimization
  - grammar/syntax checking
  - object suppression
  - diagnostic override
- Assembly level facilities
  - assembly level listing
  - user access to symbols/tables
  - easy subroutine linkages
- Diagnostics
  - explicit messages rather than numerical codes
  - error pinpointing - to source line symbol, operator, punctuation, etc.
  - previous error recovery - eliminate domino-effect errors
  - error messages within the listing, not at the end

- error levels - warning, bad, disastrous, etc.
- error recovery - assume obvious or non-fatal conditions and continue compilation
- spelling - check for "close" spellings and proceed with compilation

A representative weighting scheme is:

Category	Potential Score	Acceptable Score	Actual Score
Diagnostics	35	29	
Partial Compilation	10	8	
Assembly Level Facilities	5	3	
TOTAL	50	40	

USER INTERFACE

2.4 Documentation

A compiler may be functionally correct and translate code properly without being completely documented. However, future maintenance, extensibility, re-targeting and user interfaces will suffer accordingly. In addition, a poorly documented compiler is highly likely to be an unreliable one.

There are three classes of documentation of concern to the procuring agency: compiler implementation, user's guide, and operator's guide.

In the implementation documentation, several items will seem unimportant upon initial receipt of the compiler. Nonetheless, they will become more apparently valuable as time progresses.

Implementation Documentation

- Compiler limitations and idiosyncracies
- Compiler source listings
- Flow charts, prosaic descriptions of every compiler module and variable
- System interfaces
- Re-targeting guidelines with a sample actual re-target effort

- Extensibility guidelines with a sample extension
- Compiler organization, allocation, tables and packed variables
- Functional breakdown (parser, expressions evaluator, optimizer, etc.)
- Each "pass" functions
- Debug aids and measurement tools
- Re-generation procedures
- Causes (internal) of error messages
- Input/output, records, blocking, etc.
- Language of implementation
- Algorithm description - polishing, parsing, etc.
- Database definition
- Parameter parsing and subroutine linkage
- Variable addressing scheme
- Lexical scanning, text decoding, searching
- Special modes such as loop-save, conditional
- Syntax analysis
- Compiler generated symbols and naming conventions
- Access to symbols - symbol table structure
- Proposed enhancements and costs

This documentation must be supplemented by a comprehensive guide to expected modification difficulties and accurate

in-line comments in the compiler code. These comments should be checked for accuracy against other documentation and test results.

The second category, user guidelines, should be equally exhaustive and complete. However, unlike poor implementation documents, errors or omissions in user guides will become rapidly and glaringly apparent.

User Guidelines

- Diagnostics - description, cause, correction procedures
- Language description - a comprehensive user oriented description of the language implemented with numerous examples, cross-referencing and job setup procedures. This document should contain two approaches: beginner's tutorial and experienced users' reference.
- Capacities, limits, minimum requirements
- Compiler idiosyncracies
- Transferable programming guidelines
- Dialectic or subset differences, items not in adherence to a standard
- Visual training aids
- User flow charter
- Assembly/other language interfaces

The operator's guide is also the type of documentation that will be immediately verifiable. The procuring agency should insist on a "no-vendor" installation by agency personnel. Most difficulties and omissions in the operator's guide will be rapidly apparent.

Operator's Guide

- Devices necessary to system residence (disk packs, system tapes, tec.)
- Peripherals used during execution - mounting procedures, labeling conventions, etc.
- System (compiler) installation procedures - step by step
- Re-generation procedures
- Disk file maintenance - scratching and compressing schedules and limits
- Core requirements
- Multi-task versus stand-alone operation
- Operator messages and responses
- Compiler options (operations oriented)
- Hardware/console switches

Category	Potential Score	Acceptable Score	Actual Score
Compiler Implementation	50	40	
User's Guide	40	33	
Operator's Guide	10	7	
TOTAL	100	80	

DOCUMENTATION

Further explanation of the items listed in this section may be found in Section 1.9, Documentation Specification.

2.5 System Interfaces

User access to computer files, linking of object modules and conditional compiler use and outputs are just a few of the system interfaces each user will experience. The following items should be provided by the vendor and adequately described. Procuring agency acceptance of these items should not be assumed.

- Job control requirements
- Operating system usage
- Text editor and similar utility interfaces
- Object language interfaces with other system routines and languages
- Device independence/dependence
- System subroutines
- Resource requirements
- Compiler debug facilities/requirements
- Load time system overhead-core,devices, etc.
- Batch, remote, interactive and on-time requirements
- Execution time and I/O wait states expected as a function of system load
- Minimal configuration
- Re-entrancy
- Multi-processing

For additional descriptions please refer to Section 1.7, System Interface.

Category	Potential Score	Acceptable Score	Actual Score
Job control	20	15	
Resource Requirements	5	4	
System Routines	5	4	
Devices - I/O	10	8	
Utility Interfaces	10	9	
TOTAL	50	40	

SYSTEM INTERFACE

2.6 Options

Section 1.3 discusses compiler options which may be specified in the procurement of a compiler system. Option requirements and needs are divergent among installations, personnel, and applications and are usually dictated by available resources.

Of all the optional features listed in Section 1.3, the most variable and expensive will be optimization.

The following chart represents a general concensus of the relative importance of the most popular optimization techniques and can serve as criteria evaluating compiler optimization acceptance.

COMPILER OPTIMIZATION HIEPARCHY

Technique	Application Areas			Optimized Gain	Increased Programmer Productivity	Overall Payoff
	Overall Program	Computational Statements	Decision Statements			
Common Expressions Moving non-dependent variables out of loops	Low	Medium	Low	Speed	Yes	Medium
Common Expression Evaluation X=(I+1)/10+(I+1)*2 X=Y(I+1)+Z(I+1)	Low	Medium	Low	Speed	Yes	Medium
Function Evaluation SQPT(5)	Low	Low	Low	Speed	Yes	Low
Constant Evaluation X=5+7	Low	Low	Low	Speed/ Space	Yes	Low
Expression Evaluation X=Y-5+(A-B)*2 -A-B =Y-5+A-3B	High	High	Low	Speed/ Space	Yes	High
Logical IF's IF X GOTO 5 GOTO 6 (Eliminate one expression)	Low	Low	High	Speed	Yes	Medium
Logical IF Inversion IF X GOTO 5 GOTO 6 5 CONTINUE	Low	Low	Low	Speed/ Space	Yes	Low

COMPILER OPTIMIZATION HIERARCHY

Technique	Application Areas			Optimized Gain	Increased Programmer Productivity	Overall Payoff
	Overall Program	Computational Statements	Decision Statements			
DO or FOR Loops						
a. Collapsing Loops	Medium	High	Low	Speed	--	Medium
b. Addressing of Subscripts	High	High	High	Speed/ Space	--	High
c. Redundant Subscript Computations X(I+1)= Y(I+1)=	High	High	High	Speed	--	High
d. Converting Limits	Low	Low	Low	Speed	--	Low
Expression Re-ordering Y=-X+5 =5-X	Low	Low	Low	Speed/ Space	--	Low
In-line Code for Selected Operations X=Y**2	High	High	Medium	Speed	--	Medium
Complex Variable Expression Evaluation	Low	Medium	Low	Speed	--	Low
Utilizing Special Extended Hardware Instructions	Medium	Medium	High	Speed	--	High
Index Register Allocation	High	High	High	Speed	--	High
Memory of Registers Used	Low	Low	Medium	Speed/ Space	--	High
Accumulator Allocation	Medium	High	Medium	Speed	--	High

Category	Potential Score	Acceptable Score	Actual Score
User Debug Facilities	12	10	
Optimization of Code	13	11	
Measurement Tools	17	15	
Test Aids	10	9	
Documentation Aids	5	4	
Cross/Reference/ Dictionary	3	2	
Text/Source Processing	5	4	
Partial Compilation	2	2	
Compilation Directives	3	3	
TOTAL	70	60	

OPTIONS

2.7 Extensibility

A compiler is extensible if it can be easily modified to include new features. Perhaps the best way to test for the acceptability of this feature is to insist upon a demonstration. In other words, after the compiler has been implemented and delivered for acceptance testing, a new feature, statement or primitive should be added to the language. A detailed written description of all steps taken to implement the new feature should be kept. This would include:

- source updates
- source additions (new code)
- re-compilations
- re-linking
- self-test
- errors, cause and corrections
- man-hours
- estimated relative difficulty

This workbook and other deliverable documentation should give an indication of the degree (score) to which the following extensibility attributes have been used:

- documentation
- modularity - number of programs changed and

percentage of code changed in each module

- open-end design - array structures, tables, intermediate languages that required re-definition
- structured programming - degree of difficulty in locating appropriate code and success in changing it
- compiler restrictions - syntax, usage of the new features
- parameterization - array entries, pointers, tables, parsing algorithms

See Section 1.4, Extensibility, for additional descriptions.

Category	Potential Score	Acceptable Score	Actual Score
Modularity	9	8	
Documentation	6	5	
Open-ended	4	3	
Parameterization	5	4	
Structured Programming	4	3	
Compiler Restriction	2	2	
TOTAL	30	25	

EXTENSIBILITY

2.8 Transferability

Transferability of a compiler can refer to changing the host or compiler resident computer or to changing the target or object machine. Perhaps the most likely and therefore critical of the two is re-targeting. Most installations which procure a rather expensive compiler expect to utilize the host machine for an indefinite period of time. New applications and target computers can be required at relatively short notice, however.

The design aspects which most directly effect transferability of host or target are the same as those listed for extensibility:

- modularity
- documentation and source comments
- parameterization
- structured programming
- open-ended design

In particular, all host and target machine dependent attributes must have been parameterized and/or isolated to specifically identified modules. The documentation must clearly reflect all machine dependent constants, algorithms and programs.

The suggested test for extensibility is also the best way to judge transferability - change the conditions. It is considerably more expensive to re-host or re-target in most instances, however. For this reason, the extensibility test can be used as an approximation of re-hosting difficulties. If that test showed high reliability of comments, modularity and parameterization, then the re-hosting task has probably been minimized. Of course, re-hosting difficulties are also a function of implementation language - technique. Self-compiled or interpretative compilers will be most easily transferred. Assembly level coded compilers are virtually non-transferable. Since re-targeting of the compiler is highly probable, transferability of object is of most immediate importance. Modularity is again the keyword.

In modern compiler technology, it is possible and relatively easy to isolate all target machine dependent code generation into one set of independent modules.

Optimization modules, if present, should be mathematical processes divorced from machine parameters.

The code generator portion of the compiler should be approximately 30% of the total depending on language and target.

Again, the best way to judge re-targetability is to procure another target and insist on use of agency or different vendor personnel in the effort. If this is possible, then the previously assigned scores for modularity, documentation and parameterization can be used again.

Category	Potential Score	Acceptable Score	Actual Score
Modularity	40	32	
Documentation	20	15	
Parameterization	10	8	
Implementation Language	30	25	
TOTAL	100	80	

TRANSFERABILITY INDICATORS

Additional discussion of transferability may be found in Part 1.

2.9 Schedule and Installation

Assigning a score for schedule and installation performance is a highly subjective task. The schedule had been determined in the specification and contract negotiation phases. If it is assumed that the schedule was a realistic one, then failure to meet delivery dates and milestones will probably result in payment delays. In most cases, delay of payment is usually sufficient penalty to the vendor. A score is somewhat valuable, however, in helping to determine efficiency and reliability of the compiler. Proper milestones can help to insure checkout and acceptance of each phase of the compiler and thus improve the chances that the final product will be reliable.

Schedule milestones for a typical compiler project were discussed in Section 1.10. For acceptance scoring purposes, the following major categories can be used.

Category	Potential Score	Acceptable Score	Actual Score
Milestones On-Time	5	4	
Documents Complete at Each Milestone	15	13	
Milestone Coding Complete	10	9	
Errors at Each Milestone	10	9	
Maintenance	35	30	
TOTAL	75	65	

SCHEDULE AND INSTALLATION

Errors at each milestone refers to the number and severity of errors encountered - usual, fewer than usual, abnormally large number, etc.

Maintenance arrangement refers to the ease and rapidity in which error reports are handled. Does the proper person(s) respond or is the task shuffled from senior to junior personnel? Is the time of response acceptable, quicker than expected, poor? Does there seem to be a lot of head-shaking and throwing-up-of-hands or are errors handled quietly and professionally?

Milestone coding and documentation should be complete. Was the code/document submitted just to meet the deadline and incomplete or inaccurate? Does each milestone delivery "stand alone" and reflect modular design techniques?

Training will probably be given the highest weighting in this category. Some items to consider in scoring the training function include:

- Adequacy of personnel used - were they the implementors and users, or professional teachers?
- Range of subject material - did it include the full range from design concepts to maintenance and enhancements?

- Depth of subject material - was each program module, source language type, and system generation procedure covered completely?
- Time frame - is training on-going, repeatable, hurried, etc.
- Agency personnel - are the trainees confident in their use, maintenance, and modification ability? Do they "understand" the compiler or are they just going through the motions, etc.?
- Options - were all options explained and demonstrated?
- Error detection and reporting - were procedures developed and explained for determining when bugs exist and for reporting them to the vendor?
- Hardware and operations - have the proper personnel been instructed on use, initial program load, error recovery, and similar operational problems?

2.10 User Profile Adherence

In Section 1.8, several aspects of the User Profile or cross-section of user requirements were listed. At this point in the acceptance testing of the compiler, the procuring agency must determine if the delivered product truly adheres to the desired goals of the end-users.

Several subtle changes may have been introduced which could have the effect of eliminating classes of users' problems. For example, were fixed-point variables changed to floating point for "efficiency" by the vendor? Are fewer users able to compile simultaneously while maximum source program size has increased/decreased? The suggested weighting scheme is based on a broad range of users and should be changed according to the particular agency's most basic requirements.

Category	Potential Score	Acceptable Score	Actual Score
Compiler type - batch, etc.	8	6	
Response Time	5	4	
Number of Users	7	6	
Relative Efficiency (Compile/Execute)	5	4	
TOTAL	25	20	

USER PROFILE ADHERENCE

Part 3 Compiler Acceptance Evaluation Matrix

The Compiler Acceptance Evaluation Matrix is a summarization of the sub-matrices listed in Part 2.

Please note that the specification matrices in Part 1 were not included in the Acceptance Evaluation Matrix. Although it might seem desireable to place specification and acceptance items in a side-by-side or hierarchial fashion, this is really not meaningful.

Part 1 discussed specification decisions, documents, costs and similar items to be prepared or studied by the procuring agency. Thus, a rating or performance score cannot be assigned to these activities and no scores were assigned to specification items in Part 1.

It is reasonable, however, to use the discussions in Part 1 as a basis for preparing potential minimum acceptable, and actual criteria evaluation scores for the items listed in Part 2. If the procuring agency stressed a particular item in its specification, then all acceptance criteria effected by that item would be weighted heavily.

For example, if precise language definition was stressed in the specification, then the accuracy and reliability of syntax/semantics analyzers (acceptance criteria) would be heavily weighted.

The sample Compiler Acceptance Evaluation Matrix provided is based on a hypothetical "average" compiler. The values assigned, therefore, represent composites of values derived from the analysis of several languages and their compilers.

The procuring agency should prepare a new set of potential scores as a function of language, project, user group and other constraints discussed in Part 2. For example, a file management system written in COBOL would probably not depend heavily on numeric accuracy (past 3 decimal places) or even resource utilization; but, it is likely that system interface would be extremely important.

In the Compiler Acceptance Evaluation Matrix, the following titles are used:

POTENTIAL SCORE:	The weight or importance of the criteria in relation to a total score of 1000.

MINIMUM ACCEPTABLE:	The minimum POTENTIAL SCORE which would constitute an acceptable performance.
ACTUAL SCORE:	The value or percentage of POTENTIAL SCOPE which was actually observed in evaluating or testing the compiler criteria.
PASS/FAIL:	A check indicates that the actual score was less than acceptable: a failure. In this case, re-work and re-evaluation would be necessary before total acceptability can be attained.
COMMENTS:	Procuring agency notes or comments regarding the test used, score attained, value of the criteria, etc.

COMPILER ACCEPTANCE

EVALUATION SAMPLE MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Accuracy and Reliability					
• Internal Structure	100	90	94		
• Syntax/Semantics Analyzer	25	23	24		
• Assignment Statements	75	70	72		
• Declaration/Allocation	50	45	45		
• Control Statements	25	23	24		
• Subroutine Statements	25	23	24		
• Input/Output Statements	25	23	25		
• Limits	25	20	25		
• Documentation	50	43	35	✓	See Note 1
Subtotal	400	360	368		
Resource Utilization					
• CPU Processors (Time)	40	30	39		
• Core Storage	10	8	0	✓	See Note 2
• Instruction Set	20	18	10	✓	
• Peripheral Devices	5	4	5		
• Registers/Addressing	10	8	6		
• Programmer Productivity	10	8	8		
• Numeric Accuracy	5	4	5		
Subtotal	100	80	73	✓	

COMPILER ACCEPTANCE

EVALUATION SAMPLE MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
User Interface					
• Diagnostics	35	29	30		
• Partial Compilation	10	8	10		
• Assembly Facilities	5	3	4		
Subtotal	50	40	44		See Note 3
Documentation					
• Compiler Implementation	50	40	30	✓	See Note 4
• User's Guide	40	33	35		
• Operator's Guide	10	7	8		
Subtotal	100	80	73	✓	
System Interfaces					
• Job Control Language	20	15	16		
• Resource Requirements	5	4	4		
• System Routines	5	4	4		
• Devices - I/O	10	8	8		
• Utility Interfaces	10	9	10		
Subtotal	50	40	42		See Note 5

COMPILER ACCEPTANCE

EVALUATION SAMPLE MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Options					
• User Debug Facilities	12	10	12		
• Compiler Optimization	13	11	11		
• Measurement Tools	17	15	17		
• Test Aids	10	9	10		
• Documentation Aids	5	4	4		
• Cross/Reference	3	2	3		
• Text/Source Processing	5	4	5		
• Partial Compilation	2	2	2		
• Compilation Directives	3	3	3		
Subtotal	70	60	67		See Note 6
Extensibility					
• Modularity	9	8	8		
• Documentation	6	5	2	✓	
• Open-Ended	4	3	4		
• Parameterization	5	4	4		
• Structured Programming	4	3	4		
• Compiler Restrictions	2	2	1		
Subtotal	30	25	23	✓	See Note 7
Transferability					
• Modularity	40	32	35		
• Documentation	20	15	5	✓	See Note 8
• Parameterization	10	8	8		
• Implementation Language	30	25	30		
Subtotal	100	80	78	✓	

COMPILER ACCEPTANCE

EVALUATION SAMPLE MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Schedule and Installation					
• Milestone on Time	5	4	4		
• Documentation Complete Completeness	15	13	13		
• Milestone Coding Completeness	10	9	9		
• Errors at Each Milestone	10	9	10		
• Maintenance/Training	35	30	31		
Subtotal	75	65	67		
User Profile Adherence					
• Compiler Type	8	6	8		
• Number of Users	7	6	7		
• Response Time	5	4	5		
• Relative Efficiencies (Compile/Execute)	5	4	2	✓	See Note 9
Subtotal	25	20	22		
TOTAL	1000	850	857		

Notes from COMPILER ACCEPTANCE EVALUATION SAMPLE MATRIX

Note 1: Documentation incomplete or erroneous. Must be redone. Note: Actual total exceeds minimum yet compiler is not satisfactory due to documentation.

Note 2: Compiled Program would not fit in core. Basic benchmark not satisfactory. Code optimization and more efficient code generator necessary.

Note 3: Satisfactory.

Note 4: Errors in documentation.

Note 5: Satisfactory.

Note 6: Very Good.

Note 7: Errors in documentation.

Note 8: Incomplete, erroneous documentation.

Note 9: Would not compile basic benchmark adequately.

COMPILER ACCEPTANCE

EVALUATION MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTANCE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Accuracy and Reliability					
• Internal Structure					
• Syntax/Semantics Analyzer					
• Assignment Statements					
• Declaration/Allocation					
• Control Statements					
• Subroutine Statements					
• Input/Output Statements					
• Limits					
• Documentation					
Subtotal					
Resource Utilization					
• CPU Processors (Time)					
• Core Starage					
• Instruction Set					
• Peripheral Devices					
• Registers/Addressing					
• Programmer Productivity					
• Numeric Accuracy					
Subtotal					

COMPILER ACCEPTANCE

EVALUATION MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTANCE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
User Interface					
• Diagnostics					
• Partial Compilation					
• Assembly Facilities					
Subtotal					
Documentation					
• Compiler Implementation					
• User's Guide					
• Operator's Guide					
Subtotal					
System Interfaces					
• Job Control Language					
• Resource Requirements					
• System Routines					
• Devices - I/O					
• Utility Interfaces					
Subtotal					

COMPILER ACCEPTANCE

EVALUATION MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Options					
• User Debug Facilities					
• Compiler Optimization					
• Measurement Tools					
• Test Aids					
• Documentation Aids					
• Cross/Reference					
• Text/Source Processing					
• Partial Compilation					
• Compilation Directives					
Subtotal					
Extensibility					
• Modularity					
• Documentation					
• Open-Ended					
• Parameterization					
• Structured Programming					
• Compiler Restrictions					
Subtotal					
Transferability					
• Modularity					
• Documentation					
• Parameterization					
• Implementation Language					
Subtotal					

COMPILER ACCEPTANCE

EVALUATION MATRIX

COMPILER CRITERIA	POTENTIAL SCORE	MINIMUM ACCEPTABLE	ACTUAL SCORE	PASS/ FAIL	COMMENTS
Schedule and Installation					
• Milestone on Time					
• Documentation Complete Completeness					
• Milestone Coding Completeness					
• Errors at Each Milestone					
• Maintenance/Training					
Subtotal					
User Profile Adherence					
• Compiler Type					
• Number of Users					
• Response Time					
• Relative Efficiencies (Compile/Execute)					
Subtotal					
TOTAL					

APPENDIX A REFERENCES

Aho, Alfred V., Ullman, Jeffrey D., The Theory of Parsing Translation and Computing, Volume II: Compiling, Prentice-Hall, 1973.

Brandon, Dick H., Segelstein, Sidney Esq., Data Processing Contracts - Structure, Contents, and Negotiation, Van Nostrand Reinhold Company, 1976.

Hays, David G., Introduction to Computational Linguistics, Americal Elsevier Publishing Co., Inc., 1967.

Higman, Bryan, A Comparative Study of Programming Languages, Americal Elsevier Publiching Co., 1967.

International Computer Systems, Inc., A Proposal For "A Statistics Gathering Package for the JOVIAL Language," RADC RFP No. F30602-73-C-0062, 1973.

Knuth, Donald E., The Art of Computer Programming, Second Edition, Volume 1, Addison-Westley Publishing Co., 1968.

Lee, John A. W., The Anatomy of a Compiler, Van Nostrand Reinhold Company, 1967.

Minsky, Marvin (Editor), Semantic Information Processing, The MIT Press, 1968.

Neighbors, Michael A., "Assuring Software Reliability," "Computer Decisions," December, 1976.

Proprietary Software Systems, Inc. (Fleiss, Joel; Phillips, Guy), "A Statistics Gathering Package for the JOVIAL Language," RADC Report No. RADC-TP-73-381, January, 1974. AD#775380/9GI.

Proprietary Software Systems, Inc., A Proposal For "Criteria for Evaluating the Performance of Compilers," RADC RFP No. TR-74-254, October, 1974.

Proprietary Software Systems, Inc., "Programming for Transferability," RADC Report No. TR-72-234, September, 1972. AD#750897.

RADC document, "Notes on the Future Considerations Regarding Compiler Specifications."

Sammet, Jean E., Programming Languages: History and Fundamentals, Prentice-Hall, Inc., 1969.

Shirely, Richard, "Performance Evaluation Computer," September/October, 1972.

Shirley, Lynn D., "Compiler Specification Guidance," July 31, 1973.

Softech Inc., (Bloom, Burton H., Clark, Mac H., Coe, Robert K., Feldman, Clare G.), "Criteria for Evaluating the Performance of Compilers," RADC-TR-74-259, October, 1974. AD#A002322.

System Development Corp., "A Guide to Computer System Measurement."

Walsh, Dorothy, A Guide for Software Documentation, Advanced Computer Techniques Corp., 1969.

Acknowledgement

Proprietary Software Systems would like to acknowledge the contributions of the following individuals to the preparation of this report.

Marilyn Azaria
James Leggett
Lois Orgo
Douglas White

In addition, Proprietary Software Systems acknowledges the sponsorship and excellent support of the Air Force System Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441.

METRIC SYSTEM

BASIC UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

*U.S. GOVERNMENT PRINTING OFFICE: 1977-714-038/233