TOOLPACK/1 RELEASE 2
INTRODUCTORY GUIDE
Second Edition - October 1986

W. R. Cowell        S. J. Hague        R. M. J. Iles

This guide is jointly issued by NAG and Argonne National Laboratory.
The NAG Publication Reference is NP1277.

March 1987

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.
A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.
DISCLAIMER

Toolpack/1 Release 2 is the second public release of the Toolpack Fortran software tools suite, the result of a collaborative project. Neither NAG nor other participants in the project can accept responsibility for the content or performance of Toolpack/1, and none of them will be liable in any way for the use of Toolpack/1 and the consequences of such use.

ISBN - 1-85206-32-8

The trademarks of the following organizations are used in this guide and in machine-readable Toolpack/1 documentation:

AT&T Bell Laboratories: Unix
Apollo Computer Inc: Apollo, Aegis
Digital Equipment Corporation: DEC, VAX, VMS, Rainbow, VT52, VT100, VT200
International Business Machines Incorporated: IBM
International Computers Limited: ICL, Perq, PNX.
The Numerical Algorithms Group Limited: NAG
Purpose:

This guide introduces Toolpack/1 Release 2, the second public release of Toolpack/1, an integrated suite of Fortran 77 software tools. The guide describes the background to the Toolpack project and explains the basic concepts and terms involved in the design and use of the tool suite. It also summarizes the facilities available in the second release and refers readers to other Toolpack/1 documentation for more detailed discussion. This guide should be regarded as essential preliminary reading for all prospective users of Toolpack/1.

Keywords: Fortran, Software tools, Programming environments, Programming aids, Software engineering, Integrated tool suites.
Preface

(a) Background

Release 2 of Toolpack/1 is the second edition of the Toolpack/1 Fortran 77 software tools suite. The first release of Toolpack/1 was the result of an international collaborative effort started in 1979. The original project aims were twofold:

1) To provide a suite of tools to assist the production, testing, maintenance, and transportation of medium-sized mathematical software projects written in standard-conforming Fortran 77.

2) To investigate the development of extensible programming support environments built around integrated tool suites.

The Toolpack project, which formally ended in 1984, was supported in the U.S.A. by the National Science Foundation and the Department of Energy, and in the U.K. by the Science and Engineering Research Council. The conceptual design of Toolpack as an integrated suite of tools and the construction of the prototype system based on that design were carried out by Professor L. J. Osterweil and colleagues at the University of Colorado [1]. Professor Osterweil and Dr. S. J. Hague served as technical co-chairmen of the project, which was coordinated by Dr. W. R. Cowell of Argonne National Laboratory. Toolpack/1 was assembled by R. M. J. Iles, I. Hounam, and M. J. Cohen of the NAG Central Office and prepared for release in conjunction with Argonne National Laboratory. Other organizations involved in the original Toolpack project were Bell Communications Research, Jet Propulsion Laboratory, the University of Arizona, and Purdue University.

(b) Relationship of Release 2 to Release 1

The first release of Toolpack/1, which has been distributed to over 400 sites, provided Fortran 77 programmers with a suite of consistent tools that could be used to manipulate programs conforming to the ANSI Fortran 77 standard, their test data, results, and documentation. The use of these tools to mechanize elements of the software life cycle can lead to a greater consistency and reliability in the resulting code. In addition to these tools, Toolpack/1 led to the provision of several other facilities:

- a portability base, TIE (Tool Interface to the Environment), designed to allow tool developers to design and produce portable tools within the Toolpack/1 framework;

- access functions to allow the tool developer to read, manipulate, and write intermediate objects for exchange with other Toolpack/1 tools;

- supplementary libraries of routines providing enhanced facilities for string, table, list, pattern, and screen handling; and

- an experimental object-oriented programming support environment for assessment and possible further development.

These additional facilities mean that Toolpack/1 also supports the programmer wishing to develop portable tools (including those for the manipulation of Fortran 77 programs) or experiment with programming environments.

Release 2 supersedes Release 1 of Toolpack/1. The definition of TIE (the portability base)
Toolpack/1 - Introductory Guide

has remained unchanged, but the tools and supplementary libraries have been modified, enhanced, and expanded. Existing implementations of TIE can support the Release 2 tools and supplementary libraries. The enhancements include

- all bug fixes and performance enhancements described in Release 1 update notices;
- new tools—in particular, a static semantic analyzer and a Fortran portability verifier;
- the ability of tools to process an enhanced dialect of Fortran 77; and
- improved and updated documentation.

The current contents of the available Release 2 services are detailed in the note TOOLPACK/1 (RELEASE 2) CONTENTS SUMMARY available from NAG.

(c) Public Access/Domain

The first release of Toolpack/1 was the result of a collaborative research project to study the design and production of programming support tools and so to stimulate interest in and the development of properly supported programming environments. Because of the nature of the project funding, the resulting software was in the public domain. NAG played a major role in developing and assembling Release 1, and agreed to provide a public distribution service, the aim of which was to enable Fortran 77 programmers to make sample use of the tools developed during and after the project. The Release 1 service terminated on September 1, 1986, and no further Release 1 tapes are available from NAG.

Continuing in the spirit of the public access/domain philosophy, NAG coordinated the assembly of Release 2 and has launched extended distribution and information services for this latest release. Considerable care has been taken in preparing Release 2 for general use; indeed, many Release 2 tools are in regular use within NAG in the development of software products. However, Toolpack/1 (Release 2) remains in the public domain; it is not a fully supported commercial product. In due course, enhanced programming support facilities may be available from NAG as part of a supported commercial service, but Toolpack/1 (Release 2) is not a NAG product.

(d) Acknowledgments

The second release of Toolpack/1 has benefited from the generosity of contributors in supplying additional tools and installed TIECODE versions derived from the first release. These contributors include (at the time of publication)

Tim Hopkins and Martin Broom, University of Kent, UK
G. Russo and P. Defert, European Southern Observatory, FRG
Ken Hillstrom, Argonne National Laboratory, USA
Peter Jewell, University of Bradford, UK
Mark Maybee, University of Colorado, USA
K. M. Lovell, University of Dundee, UK
A. A. Pollicini, CEC Joint Research Centre, Ispra, Italy
D. J. Martin, University of Glasgow, UK
R. Traynor, Coventry (Lanchester) Polytechnic, UK
Peter Endebrock and H. Ritter, University of Hannover, FRG

The authors gratefully acknowledge the comments and assistance provided during the preparation of this guide by their colleagues Malcolm Cohen, Brian Ford, Burton Garbow, Sven Hammarling, and Gail Pieper.
The authors and tool developers would additionally like to thank the staff of NAG Central Office, particularly Mike Hooper, for their patience, tolerance, and understanding in using the tools during their development.

W. R. Cowell
S. J. Hague
R. M. J. Iles

September 1986
CONTENTS

1 General Introduction ........................................................................................................... 1
  1.1 What is Toolpack? .......................................................................................................... 1
  1.2 Purpose of This Guide ................................................................................................. 1
  1.3 The Rest of This Guide ............................................................................................... 1
  1.4 An Introduction to Software Tools ................................................................................ 2
    1.4.1 Mechanization Through the Use of Tools ............................................................... 2
    1.4.2 Basic Assumptions .................................................................................................. 3
2 General Design and Operating Principles ........................................................................ 10
  2.1 Design Overview .......................................................................................................... 10
    2.1.1 The Integrated Tool Suite .................................................................................... 11
    2.1.2 The User/Toolpack Interface ............................................................................... 12
    2.1.3 The Tool/System Interface .................................................................................. 13
  2.2 Design Detail ................................................................................................................ 14
    2.2.1 Virtual Machine Files .......................................................................................... 15
    2.2.2 Examples of Tool Invocation Techniques ............................................................ 16
    2.3 Toolpack/1 Fortran Language .................................................................................... 19
3 Tools Available in Toolpack/1 ........................................................................................ 21
  3.1 General Tools .............................................................................................................. 21
  3.2 Documentation Generation Tools .............................................................................. 21
  3.3 Fortran 77-Oriented Tools ......................................................................................... 22
  3.4 Facilities for Tool Construction .................................................................................. 23
4 Using the Tools .................................................................................................................. 25
  4.1 Examples of Tool Usage .............................................................................................. 25
    4.1.1 Polishing a Fortran 77 Source File ....................................................................... 25
    4.1.2 Complex Processing of a Fortran 77 Source File .................................................. 25
    4.1.3 Handling Documentation ..................................................................................... 26
  4.2 Tool Invocation Sequences ......................................................................................... 27
5 Before You Start .............................................................................................................. 29
  5.1 Nature of the Support for Toolpack/1 ......................................................................... 29
  5.2 Prospects for Future Development .......................................................................... 29
  5.3 Other Documents to Read ......................................................................................... 29
  5.4 What to Do Next ........................................................................................................ 29
  5.5 Contact Addresses ..................................................................................................... 30
6 Local Site Information ..................................................................................................... 31
  6.1 Example 1: Toolpack on the Apple, NAG Central Office ........................................ 31
  6.2 Example 2: Toolpack on the ICL Perq, NAG Central Office .................................... 31
  6.3 Example 3: Toolpack on the OUCS/VAX2 ............................................................... 31
Glossary ............................................................................................................................... 33
References ........................................................................................................................... 40

Appendix A Tutorial Examples .......................................................................................... 42
Appendix B Quick Reference to Tool Information .......................................................... 65
Appendix C Quick Reference to Regular Expressions .................................................... 80
Appendix D Toolpack/1 Release 2.1 Documentation ....................................................... 81
1. General Introduction

1.1. What Is Toolpack?

Toolpack is both the name of a project and of a suite of software tools for Fortran programmers. A brief description to the Toolpack project is given in the preface to this guide. In the rest of this guide, we shall use the name “Toolpack” to refer to a suite of Fortran software tools built using Toolpack conventions. When we are specifically referring to the released suite of Toolpack tools, the name Toolpack/1 will be used.

1.2. Purpose of This Guide

The guide explains the basic concepts and terms involved in the design and use of the tool suite. It also overviews the facilities available in the second release and refers readers to other Toolpack documentation for more detailed discussion. This guide has been written to act as an introduction to Toolpack/1 with essentially four categories of readers in mind:

1. prospective users of Toolpack,
2. prospective installers,
3. prospective tool writers, and
4. others interested in the design of Toolpack, e.g., software engineers.

Certain sections in this guide may not be relevant to all categories of readers and so are appropriately identified. One aim in writing the guide has been to convey adequate but not superfluous information to the first-time Toolpack user who has a straightforward requirement. It is likely that the same user will subsequently attempt more complex tasks using Toolpack and may then benefit from studying this guide again, including those sections omitted on first reading.

As mentioned in the preface, “Fortran” in the context of the second release of Toolpack means the Fortran 77 programming language, as defined in [3], extended as defined in [2]. The initial release covered a much smaller subset of extensions, as standard conformance-checking was not then available.

1.3. The Rest of This Guide

Section 1.4 gives an introduction to the use of programming aids, and may be omitted by the more experienced Fortran programmer already familiar with concepts such as symbol tables, semantic analysis, and token streams.

Section 2 discusses the general design and operational principles that were devised in developing Toolpack. Section 2.1 is intended to give sufficient information about the main concepts so that a user with a straightforward application, e.g., “format the source code of my program,” can proceed to Section 3 and beyond. In Section 2.2, readers with more complex applications or with an interest in software techniques will find a more detailed discussion of the design of Toolpack. Section 3 gives a general functional description of the facilities provided in the second release of Toolpack; the individual specifications of particular components of the tool suite are given in separate documents (which are distributed in machine-readable form but may also be available in printed form at some sites). Extended examples of the use of some of the facilities described in Section 3 are given in Section 4 and the sample session in Appendix A. The fifth section is entitled “Before You Start” and discusses several important issues:

* the nature of the support provided for Toolpack in field use,
* the prospects for future developments of Toolpack, and
* other documents to which the reader should now refer (noting, in particular, the need to consult local documentation).

Contact addresses are given at the end of Section 5.
Section 6 contains information pertinent to your own local site of Toolpack that should be provided by the Toolpack Installer. The Glossary of Terms which follows will be of particular use since it explains the meaning of various expressions used in a Toolpack context, such as the term Toolpack Installer, used above.

Appendix A contains tutorial examples of the use of tools via the Toolpack/1 command executor ISTCE. Appendices B and C give quick reference information for tools and regular expressions. Appendix D lists additional Toolpack documentation.

1.4. An Introduction to Software Tools

The rest of Section 1 is devoted to an explanation of the nature and role of software tools. It is intended primarily to be read by the less experienced Fortran programmer who is unfamiliar with terms such as software tools and programming aids. Other readers may prefer to proceed to Section 2.

Toolpack is a suite of software tools designed to support the Fortran programmer. In this context, a “software tool” is a utility program that may be used to assist in the various phases of constructing, analyzing, testing, adapting, or maintaining a body of Fortran software. Alternative terms used to refer to such programs are programming aids and mechanical aids. We will henceforth usually refer to these programs simply as “tools.” Typically, the input to such a tool is the user’s Fortran software. That software may be part of a program, a complete program, or indeed a suite of programs. The tool reads the software as input data, processes it, and produces output that may have one or both of the following forms:

1. A report that gives an analysis of the input program, e.g., a summary of the types of statements used. A tool that produces this form of report without requiring the user program to be executed is called a static analyzer.

2. A modified version of the input program. In this case, the processing tool is a transformer. An example of such a tool is a formatter which improves the appearance of the user’s program. The Toolpack/1 formatter is called “polish.”

In some cases the input may not be a program as such, but could be an associated or derived body of information, e.g., test data, documentation, or a report based on information generated by a previously applied tool. Tools that assist directly in the preparation of documentation are usually called documentation generation aids. These and other tools serving utility functions all have an important role to play and so, even if they do not process the user program directly, they are still regarded as programming aids.

Further examples of software tools of interest to the Fortran programmer include

- a text editor, with Fortran 77-oriented features, which is useful for creating a new Fortran program and then modifying it;

- a transforming tool that standardizes the declarative part of a Fortran program; and

- an instrumentor that modifies the program by inserting monitoring and other control statements. The instrumented program is then compiled and executed, and data are gathered that are used to generate reports. Execution of an instrumented program is an example of dynamic analysis.

1.4.1. Mechanization Through the Use of Tools

Examples of each of the tools mentioned in the previous section can be found in the contents of the second release of Toolpack. The list of facilities in that release, given in Section 3, is by no means exhaustive: there are many more tools that could be of considerable use. These could be written to conform to Toolpack/1 construction guidelines using the facilities outlined in Section 3.4.

Readers unfamiliar with the use of software tools may not be convinced that these efforts to provide enhanced support for Fortran programmers are either necessary or worthwhile. After all, they might argue that the Fortran language has survived for over 30 years without the availability of such
tools in widespread use. They might also wonder why terms such as “software tool” and “programming aid” have become fashionable only recently, and are still not part of some Fortran teaching courses. More fundamentally, they might ask why the major vendors of Fortran compilation systems have apparently paid little heed to the desirability of providing various program analyzers and transformers.

To attempt to give full answers to the above questions would be inappropriate in this introductory guide, since numerous technical and other issues are involved. However, the upsurge in interest in Fortran software tools can be ascribed mainly to the experiences of numerical software producers during the 1970’s. In their efforts to build soundly constructed, well-tested, portable products, these producers came to appreciate the potential value of using software tools and the associated benefits of mechanizing parts of their particular software production cycles (see [4] and [5] for instance).

The main potential benefits claimed for mechanization are

- Reliability: if a thoroughly tested tool operates on a valid user program, then the outcome should also be predictably trustworthy.

- Economy: if a tool is to be applied repeatedly, then the costs of its development and usage may be significantly less than those of performing the task by hand, given a sufficient scale of operation.

- Consistency: related to the notion of reliability, there is the prospect that programs processed by a tool will be more uniform in appearance and construction than if they were changed by hand. This makes programs easier to understand and to maintain.

- Flexibility: if a program is to be transformed by a tool whose effects can be readily altered, then such a tool gives scope for producing alternative forms (which may be useful for experimental purposes or for the management of variant forms of the program to be used in different contexts).

There are therefore several strong arguments for considering the use of software tools. Having outlined these advantages, however, we must mention possible drawbacks, particularly those that the less experienced Fortran programmer might encounter. Perhaps the two most important cautionary notes to make are

- Overreliance on the outcome: the user may assume that it is unnecessary to make any kind of visual or computational check on a transformed program before, for instance, sending copies of that program to colleagues elsewhere.

- Underestimation of required resources: the exact computational costs of using software tools will vary considerably (and in the case of Toolpack/1 will depend critically on the efficiency of the implementation of the underlying portability base). Users should not underestimate the storage and processor requirements of using software tools.

However, despite these reservations, the drive towards greater mechanization of the programming process deserves strong support on at least the following counts:

- Toolpack/1 will be of practical benefit to many users.

- Toolpack will give a foretaste of the kind of comprehensive, integrated tool suite that ought to be at the programmer’s disposal.

Users should be encouraged to expect that facilities equivalent or indeed superior to those in Toolpack should be available, as a matter of course, as part of their host Fortran compilation system.

1.4.2. Basic Assumptions

To use Toolpack tools, one need not have an intimate understanding of their internal operations, although for the interested reader some details are given in Section 2. Both in that section and in the
specifications of the individual tools, we assume that the reader is familiar with certain basic assumptions about the way that the tools operate and the several forms of program representation. This section discusses those assumptions.

Taking first a general case, assume that a software tool, T, operates on a user program in source form, which we will denote by P. The tool produces an output program in source form, to be denoted by P'.

1.4.2.1. Lexical Analysis and Token Streams

To carry out its operations efficiently and reliably, the tool T must first “recognize” the program P. This recognition involves at least a lexical analysis of the program, which means that the individual language tokens (sometimes called atoms or lexemes) are delimited and identified. For instance, the Fortran statement

\[ X = 1 \]

is recognized during lexical analysis as consisting of four tokens:

- identifier, with the value \( x \),
- the assignment operator (EQUALS),
- constant, with the value \( 1 \), and
- end of the statement.

Note that some tokens (i.e., those that represent defined language keywords) are complete in themselves, while others have a “value” consisting of the string representation of the token. For example, the token type NAME may, in Fortran, have the value of any legal identifier (e.g., FRED, X03AAF, I). The value of such a token is termed a “saved string” or “saved value.” Another form of token that receives special treatment is the comment statement. Depending on the lexical analyzer used, each comment line may be discarded, left embedded in the output stream, or placed in an associated, but separate, comment stream. As we will see later, the Toolpack/1 lexical analyzer produces a comment stream. Comment markers are placed in the token stream so that the comments can be reinserted in the correct order at a later stage of processing.

If the original program was valid in terms of the Fortran 77 language standard, the outcome of the lexical analysis phase is a stream, or sequence, of tokens. This stream consists of all the linguistically significant atomic fragments in the original program P, but much or all of the space characters in the source program P have been discarded. Thus the fragments

\[ X = 1 \]

and

\[ X \quad = \quad 1 \]

will result in the same token stream after lexical analysis, unless the lexical analyzer being used specifically retains intertoken spacing information. The discarding of spacing details means that if the token stream is restored to source form again, the restored program will be computationally identical to the original, but not necessarily identical in terms of its appearance.

A tool that performs lexical analysis is a lexer and is sometimes referred to as a scanner since the analysis involved is a preliminary scan of the program.

1.4.2.2. Syntax Analysis and Parse Trees

In terms of computing resources, lexical analysis need not be an expensive operation, and the resulting token stream can be stored compactly. Lexing Fortran is more costly than lexing most other
languages because of the definition of the language. (Consider DO10I=; is this a DO loop or an assignment?) If the software tool T can carry out its processing operation without further preliminary analysis, then its overall computational requirements, in terms of storage and processor time, will be correspondingly modest. (Examples of such tools in Toolpack are the source code formatter and the Fortran program comparison tool.) However, lexical analysis guarantees only that the individual elements of the program are valid; it does not ensure that the code conforms generally to the grammatical (syntactical) rules of the appropriate language definition (Fortran 77 in the case of Toolpack). To ensure that the program is grammatically correct, a further checking operation after lexical analysis is carried out. This operation is known as syntax analysis or parsing. To appreciate the difference between lexical and syntactical analysis, consider the following portion of a Fortran program:

\begin{verbatim}
    X = 1
    Y =
    Z = 1
\end{verbatim}

The line beginning \texttt{Y=} will be recognized in lexical analysis as consisting of three valid atoms or tokens:

- the identifier, with the name \texttt{Y},
- the assignment operator \texttt{EQUALS}, and
- the "end of statement" token.

However, it would be rejected by the syntax analyzer because it is an incomplete statement: the Fortran 77 language definition contains a syntax rule that says, in effect, that the right-hand side of an assignment statement must be a valid expression such as \texttt{X+1}.

The syntax analyzer (or parser) takes the token stream as input, checks that it is grammatically valid according to the appropriate language rules, and produces a parse tree as output. The parse tree reflects the correct syntactical structure of the program, and each statement of the program corresponds to a distinct subtree in the parse tree. For example, the assignment statement

\begin{verbatim}
    X = 1
\end{verbatim}

becomes this token stream after lexical analysis:

\begin{verbatim}
    <identifier X>, <assignment operator (EQUALS)>, <constant 1>, <e-o-s>
\end{verbatim}

where e-o-s is the special "end of statement" token.

After syntax analysis, the stream corresponds to this parse subtree:

\begin{verbatim}
    ASSIGNMENT (EQUALS)
    +--------------------------+
    |                          |
    | IDENTIFIER CONSTANT     |
    | (X) (1)                 |
\end{verbatim}

This subtree forms part of the overall tree that corresponds to the whole program. Each position in the tree is called a node. The top position, which spans all the program, is called the root, or program node. Nodes at the lowest level in the tree are called leaves and correspond to tokens in the token stream.

While performing its analysis, the parser encounters program elements with names or values, e.g., the identifier named \texttt{X}, or the constant \texttt{1}. It stores this information in a "symbol table," for ease of reference in later processing. In fact, the symbol table may exist after lexical analysis (depending on the particular lexical tool used); in that case, the parser adds more information to an existing table (e.g.,
it will distinguish between the arithmetic constant with value 10, and the label 10, whereas the lexer may not).

1.4.2.3. Semantic Analysis and Attribute Information

Syntax analysis checks that the program conforms to the formal grammatical rules of the language, but those rules may not cover all aspects of the use of the language, particularly with regard to consistency within the program and, most crucially, the significance of each instruction, i.e., the “semantics,” or meaning, of the program.

For instance, the statement

GO TO 10

is syntactically correct, but if there is no statement with label 10, the program unit is invalid. Another example could be these two type declaration statements:

```
REAL M
INTEGER M
```

Both are correctly formed in terms of the grammar of Fortran but, taken together, contravene the language standard that requires a variable to be declared only once.

The task of checking for such inconsistencies and contraventions is called semantic analysis. In the process of performing these checks, more information is gathered about various aspects of the program, and this additional data is usually stored in an attribute table associated with the symbol table produced by the syntax analyzer. For instance, in an attributed symbol table, the entry for identifier M might contain this information (in coded form):

- **M** — data type: integer; explicitly declared; used as a dummy argument; used as array bound;

if M occurred in the following parts of a program:

```
SUBROUTINE ANY (...A,M,...)
INTEGER M
REAL A(M)
```

Semantic analysis can, in fact, be a much more involved and comprehensive exercise than the preceding example suggests. It might have a particular emphasis, e.g., data flow analysis (in which the way data is passed between variables is examined) or checking for coding constructs known to give portability problems.

The output from a full semantic analysis may be voluminous, and much of the information may not be directly relevant to the input requirements of a subsequent processing tool. This implies that storage and processing resources could be unnecessarily expended while producing unwanted information. Therefore, the strategy adopted in Toolpack/1 is to incorporate into the parser a limited degree of semantic analysis, producing a modest amount of information likely to be of use to most other tools. If any of those tools has additional analysis requirements, these are fulfilled either by the tool itself or by a separate semantic analyzer. Thus the Toolpack/1 parser produces a symbol table augmented by a basic set of attribute information where “basic” in this context consists of such things as

- type of a program unit (program, subroutine, function, block data);
- whether a variable is explicitly typed, its type, and ways in which it is used;
- whether a variable is unsubscripted or an array, and whether it appears in a common block;
uses made of labels; and

the names and types of external references and whether they are declared as external.

These attributes can then be added to by the semantic analyzer. An additional feature of the Toolpack/1 semantic analyzer is that it will check conformance to the Fortran 77 standard [3], providing warnings of the use of extensions and invalid constructs.

1.4.2.4. Summary of the Recognition Process

To summarize the discussion so far, we observe that the process of "recognizing" the program, P, involves lexical, syntactical, and semantic analysis and produces two distinct "subsource" forms of program representation (the semantic analyzer merely enhances the attribute information derived by the parser):

\[\begin{align*}
\text{source program, } P \\
\text{scan} \\
\text{token stream + saved strings} \\
\text{parse} \\
\text{parse tree + attributed symbol table} \\
\text{enhance attributes} \\
\text{parse tree, additional attributes + standard conformance violation warnings}
\end{align*}\]

Earlier, we mentioned two processing tools that take as input the token stream produced by the lexer and output a modified token stream. However, some of the Toolpack tools operate on the parse tree form of the program. Use of these tools therefore involves additional storage and processing overheads. The reasons for accepting these overheads are twofold:

1. the additional checking involved in parsing the program gives a better guarantee that the processing tool will perform its task correctly, and
2. the attribute information produced as an important byproduct of the analysis is used to guide the operation of the tool.

1.4.2.5. The Overall Transformation

To complete our description of a source-to-source transforming tool, T, we must now consider how the eventual source output form is produced. If T operates on the token stream, then it must
incorporate a utility which converts the modified token stream back to source form. This utility is in fact the polish tool mentioned earlier. It converts Fortran keywords from an internal coding form for tokens to character strings and inserts spaces and newline characters where appropriate. This operation is sometimes called unscanning since it is the opposite of the initial lexical analysis phase, known as scanning.

If the tool T operates on the tree form, then before unscanning, the revised tree must be "flattened" into a token stream. This essentially involves traversing the leaves of the parse tree but also must take into account any modifications made to the symbol/attribute tables. The flattening may take place as part of the tree transforming operation within T, or as a separate phase.

The operation of our hypothetical tool, T, therefore could involve one of several transition paths between P and P', as illustrated below:

1.4.2.6. Tool Fragmentation, Combination, and Integration

In the previous discussion, we referred to a processing tool, T, as if it were a monolithic utility, incorporating several distinct phases of operation. Although we presented the hypothetical tool, T, as a source-to-source transformer, for much of the discussion we could have been referring to a Fortran compiler. Such compilers are usually also monolithic because they appear to users as large "black boxes" that transform their source code programs into a lower level language form, suitable for subsequent execution. Though the details may vary, these compilers incorporate the same kind of components for lexical, syntactical and semantic analysis as described earlier, but, instead of flattening and unscanning a program tree, they enter a code generation phase in which the low-level instructions (variously called relocatable binary, semi-compiled object modules) are produced. In general, the internal components and data structures of the compiler are not accessible to the user; a source program is presented and, if it is found syntactically and semantically valid, a compiled form of the program is returned.

The approach adopted by Toolpack is a distinct step away from the "black box" philosophy that has been prevalent among most compiler writers. The Toolpack strategy has been to fragment tools wherever practical. Fragmentation involves identifying operations performed similarly in more than one major tool and then developing a separate tool to perform that operation, placing the output in a file. These "intermediate" files can then be accessed by a number of different tools rather than each having to create the required information independently.
The overall functionality of T is achieved in Toolpack by combining a sequence of specific-function tools. Although certain common processes (e.g., polishing) are provided as monolithic capabilities, these simply make use of the general fragments already available within the tool suite. Thus polishing may be performed by the discrete invocation of scan and polish fragments (in which case the token stream remains for further processing) or by invoking a monolithic polish function (in which case no intermediate token stream is output).

The sequence of tools required may also be given a name and presented to the user as if it were a "black box" command, e.g.,

"Polish" might mean: scan ---> polish

"Apt" might mean: scan ---> parse ---> apt ---> polish

where apt is the name of an (actual) tool that converts the precision of a program in tree form, flattening to token form in so doing. If, in addition to applying apt, we wish first to use tool x with similar input/output requirements to those of apt, the tool sequence becomes

scan ---> parse ---> x ---> parse ---> apt ---> polish

or

scan ---> parse ---> apt ---> parse ---> x ---> polish

That is, it is not necessary to restore the program to source form and rescan it before applying the second tool. It is true that the program must be reparsed between the use of x and apt, but that has the benefit of validating the syntactical correctness of the program transformed by x, and ensuring the associated symbolic/attribute information is updated. This example is illustrated in greater detail in the design overview of Section 2.1.

One of the benefits of fragmentation is, therefore, that unnecessary operations may be avoided. For instance, if it was required to generate further information from the preceding example, this could be done as follows:

\[\text{scan } \rightarrow \text{parse } \rightarrow \text{apt } \rightarrow \text{polish} \]

\[\text{\hspace{1cm} |}\]

\[\text{\hspace{1cm} |}\]

\[\text{\hspace{1cm} +--> \text{parse } \rightarrow \text{revised symbol table}}\]

\[\text{\hspace{1cm} |}\]

\[\text{\hspace{1cm} +--> \text{callgraph}}\]

\[\text{\hspace{1cm} printed symbol table}\]

\[\text{\hspace{1cm} common block usage information}\]

Another advantage is that it facilitates the inclusion of additional tools (so long as they conform to Toolpack conventions), thus making the suite of tools extensible. It also permits considerable flexibility in the way in which the tools are presented to users. The tools are "integrated" in the sense that they can be combined in an appropriate sequence to satisfy a user's request for a specified result. The ways in which the tool sequence can be constructed and then invoked are described in the discussion of the user/Toolpack interface in Section 2.1.2.

The price paid for this fragmentation is the need to generate and use intermediate files. If no further processing of intermediate files is required, then this is a costly and unnecessary step which can be avoided by combining the fragmented capabilities into single monolithic tools. Several such monolithic tools are provided in Release 2 of Toolpack/1. These versions always run more quickly than their fragmented equivalents, but prevent additional processing steps from being added to intermediate program forms.
2. General Design and Operating Principles

This section gives readers background information about the design and operation of Toolpack/1. Section 2.1 provides an overview that should enable users with straightforward applications to proceed to Section 2.3. Toolpack installers, tool writers, and users who have a deeper interest in the internal aspects of the system should also read Section 2.2.

2.1. Design Overview

Toolpack/1 consists of three categories of software:

1. an integrated suite of tools.
2. the user/Toolpack interface, and
3. the tool/system interface.

As discussed in Section 1, Toolpack/1 can be described as an integrated, extensible, and portable suite of tools intended to support Fortran 77 programmer. The tool suite has an associated set of files (tilestore): the tools process objects such as program units held in these files and place the results of their processing in other files. The term "Toolpack system" refers to the combination of the tool suite and the Toolpack filestore. We may also refer to this as an "environment" in that a developer may be able to undertake a succession of tasks using Toolpack/1 facilities only. However, Toolpack/1 does not provide a complete, or "closed," programming environment because major facilities (e.g., a Fortran 77 compiler) are not part of the distributed tool suite. On some host systems there will be a distinct separation between Toolpack tools and major host system facilities such as the compiler. For these "hostile" hosts, a user needing access to a compiler or system text editor may have to leave the Toolpack/1 environment. On other systems, however, this separation can be effectively hidden by mounting the tool suite so that the user can invoke host system facilities from within the Toolpack/1 environment (see Section 3). When mounted on these "friendly" host systems, Toolpack/1 can therefore offer a comprehensive and convenient "closed" support environment for the Fortran 77 programmer.

The user/Toolpack interface interprets user commands and invokes the appropriate sequence of tools. Each tool in the sequence expects to find its input in files created by the user or by other tools; in its turn, the tool writes its output to files. The tools call routines in the tool/system interface to perform input/output with the file store, including access to special files created by other tools. The tool/system interface also provides interaction between tools and the user/Toolpack interface as well as various "low-level" functions required by tools, such as character translation, bit manipulation, string processing, and time-stamping.

The user’s view of Toolpack is presented by the user/Toolpack interface and the commands associated with particular tools. The user must know what results may be obtained and what commands must be issued to produce the results, but he may use the tools effectively without knowing the details of their operation: moreover, the tool/system interface is invisible to him.

The three categories of Toolpack software are semi-independent in the sense that if any one is altered in certain prescribed ways to match the needs and resources of a particular site, the implied alterations of the other two components are well understood and easily made. The Toolpack installer is responsible for assembling the system from the available components and providing the necessary high-level documentation to users. The principal alterations permitted in each software category and their implications for users are as follows:

1. The tool suite may be tailored to the needs of users: it may be enlarged to contain additional tools and may omit tools of no interest at a particular site. The set of available tools and their interdependencies are known to the user/Toolpack interface: hence, tailoring of the tool suite implies that the user/Toolpack interface must be kept current.

2. The user/Toolpack interface may be selected from among several now available but a site with the required programming resources and a strong interest in preserving a familiar command format may choose to develop its own interface. The choice of user/Toolpack interface reflects the level of sophistication in tool invocation and file management available to the user, as well as the cost of that sophistication in terms of systems resources.
The definitions associated with the tool/system interface are the same for every host system, but the software that implements these definitions may be in (reasonably) portable Fortran or may be tailored to a particular host, and need not be in Fortran. Although the tool/system interface is invisible to users, its implementation strongly affects its efficiency in terms of execution time and the file space required for Toolpack. These definitions are described in [6].

Basic features of the software in each category are sketched out in the following three sections.

2.1.1. The Integrated Tool Suite

A summary of the major tools in the current suite may be found in Section 3: a full list of tools, including their arguments, may be found in Appendix B. The tools are written in Fortran 77. As mentioned above, each tool interacts with its environment through calls to routines in the tool/system interface. Whereas the tool writer must be aware of internal Toolpack/l design conventions, none of these need concern the user. The important fact for the user is that the source code for the tools is identical—the strongest form of portability—in every host environment. Of course, different sites may select quite different user/Toolpack interfaces, so the user cannot expect to issue the same commands at a different site: but he can be sure that any differences are not in the tools themselves.

The following example illustrates the integration of the tool suite: a Fortran program in real single precision is to be transformed to double precision, the resulting program is to have its declarative part rebuilt, and then the entire program is to be formatted according to user-selected options. Precision transformation entails changes in declarations, representation of constants, format statements, and intrinsic functions. Examples of the operation of these tools can be found in Appendix A.

The precision transformer tool [7] does not expect all the declarations to be explicit. It is more sensible to apply precision transformation prior to declaration standardization [8], as the precision transformer will change the types of variables and may introduce new intrinsics. The declaration standardizer ensures that all the declarations are made explicit and are arranged in a systematic and uniform way, with optional comments setting off the various types of declared names. Options enable the user to control the layout of the declarative section, to choose among legal possibilities for declaring arrays in common, and to remove the declarations of variables not used by the program.

Both the declaration standardizer and the precision transformer do their work by transforming a parse tree and symbol table to produce a revised token stream. The parse tree and symbol table are created from the token stream by the parser [9]. The final token stream resulting from the transformations must then be reconstituted as Fortran source, which is done using the unscanner [10]. The Fortran is formatted according to the preferences of the user as it is reconstituted from the token stream by the unscanner tool, familiarly known as "Polish" (unscanning with formatting options is sometimes called "polishing" a program). The unscanner tool provides such a wealth of user options for formatting Fortran that an options editor is provided as a separate tool. The unscanner options are contained in a file read by the tool at runtime.

The sequence of actions to accomplish the specified transformations is shown below. The name of each illustrated tool is given in square brackets, Toolpack name on the left and generic name on the right.
Figure 2.1 Precision Transformation, Declaration Standardization, and Program Formatting

It should be noted that the sequence of tool operations ISTLX-ISTYP-ISTPT may be replaced by the "monolithic" tool ISTQT. Similarly, ISTYP-ISTDS-ISTPL may be replaced by the ISTDT. The complete list of Toolpack/1 (Release 2.1) tools is given in Appendix B.

In Section 2.2.2 we shall indicate ways in which the user may invoke the tools to perform such a sequence of operations.

2.1.2. The User/Toolpack Interface

The interface that the user will see to Toolpack/1 depends on the way in which the tools are to be invoked. The interface is provided both by the tools themselves and the environment within which they are invoked. The following approaches may be adopted:
(1) Direct execution from the host operating system, using fragmented or monolithic versions of the tools.

(2) Execution from the host operating system by user provided command files.

(3) The Toolpack Command Executor (ISTCE) [11]; this may be used either directly or through the use of script files.

(4) The Toolpack Experimental Command Interpreter (ISTCI) [12].

ISTCE and ISTCI may be regarded as tools: they are written in Fortran and conform to the tool/system interface. Their appearance to the user will be summarized in Section 2.2.2. They utilize and give user access to the complete Toolpack filestore, which is described in detail later. The resulting operating regime is said to be embedded.

On the other hand, a user/Toolpack interface based on command files or direct tool invocation is host-system dependent. If it does not give access to the portable file store, the operating regime is said to be stand-alone.

There are two other operating regimes: stand-alone and stand-astride. A fuller explanation of the nature and uses of the operating regimes available is given in [13].

If the user/Toolpack interface insulates the user from any need to know dependencies among tools or how intermediate files are managed, the integrated system may be called a programming environment. The system uses existing files in a data base that it maintains, or invokes the necessary tools to create the files it needs but which do not exist in up-to-date forms.

The choice of user/Toolpack interface profoundly affects the way the user works in a Toolpack environment. The additional system resources required to support a more complex system may be offset by capabilities such as selective processing which reduces the tool invocations performed to a minimum. The choice for your site depends on the facilities offered by your host system and on the nature of your interest in Toolpack—whether primarily as a research prototype or primarily as a way of supporting existing program development activities. See Section 6 for information pertinent to the local site.

2.1.3. The Tool/System Interface

A tool/system interface is any implementation of a set of definitions called TIE (Tool Interface to the Environment). TIE defines a character set and a set of functions expressed as calls to Fortran subroutines. Conceptually, a tool/system interface is a "virtual machine" on which the tools, as Fortran programs, execute. TIE is sometimes called a "library." TIE is, in fact, a set of routine and capability definitions that define the characteristics of the Toolpack Virtual Machine. There can be, and indeed are, many implementations of the TIE definitions, and it is these that really form the library.

Various implementations of TIE currently exist. One, called TIECODE [14], is provided entirely in Fortran 77, though certain routines contain host-sensitive Fortran constructions, and a few must be implemented in a host-dependent way. An implementation tailored to a host system will be more efficient.

TIE is divided into sublibraries. The different operating regimes listed in Section 2.1.2 result from loading the tools with implementations of the appropriate sublibraries of TIE. A conceptual diagram of the split of a TIE implementation is shown below.
Additional facilities, called supplementary libraries [15-18], provide important special functions such as accessing the information in token stream, parse tree, and symbol table files produced by the scanner and the parser. Other supplementary libraries offer extended string-handling and table-handling facilities and portable screen-manipulation facilities.

The existing TIE definitions are stable; however, a redesign of the portability base in the light of installation and tool-writing experience may be undertaken at some time in the future. Supplementary libraries may change at each release, and additional libraries (e.g., for graphics) may be defined and implemented.

1.1. Design Detail

In this section we look in greater detail at the file structure of the virtual machine defined by TIE and give examples of user commands to invoke tool operations for different choices of the user/Toolpack interface.
2.2.1. Virtual Machine Files

As pointed out in Section 2.1, files exist in the Toolpack/1 filestore. This filestore actually consists of two parts: the host filestore (HFS) and the portable filestore (PFS). HFS files are simply formatted files in the host system filestore. They are always accessible to host system utilities. The virtual machine makes no assumptions about an HFS directory structure, and the only assumption about the names of HFS files is that they may not exceed $\text{maxpath}$ - 1 characters, where $\text{maxpath}$ is a virtual machine constant set by the installer. To identify an HFS file, the user adds a one-character prefix (usually #) to its name. This hostfile_id character is stripped off the name before it is used.

PFS files reside in a tree-structured directory system similar to that provided by Unix. A directory in the tree structure may contain files or other directories. Conceptually, the directories may be nested to any depth, but there are practical limitations, such as the maximum length of a name. The fully qualified name of a file or directory specifies the complete path through the tree structure, starting with slash (/), the root of the tree, followed by a (possibly empty) list of directories separated by slashes, and ending in the file or directory name, e.g.:

```
/fred
/bill/fred
/bill/joe/fred
/bill/ted/fred
```

Such a pathname may contain up to 80 characters. Each individual directory or file name in the pathname may contain up to 12 characters, the first of which is a letter. The remaining may be letters, digits, minus sign, or period (there may be at most one period in an individual file or directory name). PFS file names are not case sensitive.

The portable filestore includes the concept of a local directory that may be set by a tool. The local directory is normally set only by command interpreters; in the embedded operating regime, the setting of the local directory is passed from the command interpreter to tools that it has invoked. Any pathname not starting with a slash is presumed to start at the local directory. The default local directory is the root.

PFS files may not be directly accessible to standard host system utilities.

In addition to these disk files, there are four preconnected files available to tools. These files are available in all operating regimes. The files and their symbolic names are as follows:

1. The standard input file (stdin), normally associated with the user's keyboard.
2. The standard output file (stdout), normally associated with the user's terminal.
3. The standard error-reporting channel (stderr). The association of this channel is host-dependent but is often the user's terminal.
4. The standard list channel (stdlst). The association of this channel is host-dependent; it is intended for connection to a spooled system printer, though it may be connected to a fixed file that can be printed by the user. See Section 6 for local information.

It should be noted that these preconnected files are not redirectable in the Unix sense. However, each can be referenced by name using one of the following alternatives:

<table>
<thead>
<tr>
<th>File</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>0 or #0</td>
</tr>
<tr>
<td>stdout</td>
<td>1 or #1</td>
</tr>
<tr>
<td>stderr</td>
<td>2 or #2</td>
</tr>
<tr>
<td>stdlst</td>
<td>3 or #3</td>
</tr>
</tbody>
</table>

Almost all files within the HFS and PFS are formatted sequentially [3], but a formatted direct-access file capability is provided.
2.2.2. Examples of Tool Invocation Techniques

We elaborate here on the operational appearance of the user/Toolpack interfaces mentioned in Section 2.1.2. The user should consult Section 6 for practical information about the user/Toolpack interface chosen by his site.

2.2.2.1. Direct Invocation

The simplest form of tool usage to set up is direct invocation. In this approach, tools are treated as ordinary host system programs that may be run individually. The naming and saving of intermediate files, and the correct sequencing of tool invocations, are the responsibility of the user. This approach is sufficient if the monolithic tools are available, or if the user feels happy with invoking low-level tool functions.

Toolpack tools receive their arguments in a TIE implementation-dependent manner, usually from a named host file. The use of a host file allows the ten arguments (each potentially a line long) and other information (e.g., current local directory) to be passed easily, but it is not very convenient for the user. To enable a tool to pick up its arguments in this way would require the user to create a mock file for the tool to read. Because of this, all tools will prompt for arguments that are not supplied, so it is only necessary to run the tools and then supply information in response to the prompts. Some TIE implementations actually use the normal host argument passage techniques; these allow arguments to be specified at invocation (see Section 6 for specific details about each local site).

When using direct invocation, the monolithic forms of the tools are often more convenient as intermediate objects, and multiple tool invocations may be avoided. The only disadvantage of using the monolithic tools is that only some functions are provided (i.e., tools such as ISTME do not have monolithic versions) and additional processing on intermediate files is not possible.

2.2.2.2. Host-Dependent Command Files

The second approach is to isolate the user from the need to know about Toolpack intermediate files when using fragmented tools. This can be done through the use of host script or command files. This approach is particularly useful where a lot of software is to be processed by a fixed tool sequence or where only a limited number of different forms of processing are to be supplied. Both monolithic and fragmented tools may be invoked from command files.

As an example, Unix command files (shell scripts) have been written to manage tool invocation and file management for a subset of the Toolpack tools. These scripts offer neither the portability of ISTCE nor the generality of ISTCI, but they provide a simple Fortran tools environment in the context of Unix. Argument passage is effected by the provision of a special program for Unix sites. Following are two examples of the use of such scripts.

*Example 1:* Analyze a Fortran program for syntactic and semantic correctness and conformance to PFORT 77, a portable subset of the Fortran 77 standard. The Unix user would type the command line

```
pfort Fortran_source_file
```

where Fortran_source_file contains one or more program units to be analyzed. This command first causes invocation of the monolith tool ISTLA (equivalent to invoking ISTLX followed by ISTYP followed by ISTSA) to create a parse tree and a symbol table with extensive symbol attributes. As described in the documentation of the individual tools, various syntactic and semantic aspects of the program are checked. Next, the tool ISTPF is invoked to examine the parse tree and extended symbol table for conformance to PFORT 77. Error messages and warnings generated by any of the tools are sent to the user’s terminal and may be redirected to a file. These error messages refer to the source program by statement number and sometimes by token number. The user may obtain a listing of the program showing statement and token numbers by typing

```
getlst Fortran_source_file
```
which invokes the lexer ISTLX in a manner that produces the required listing.

*Example 2:* Carry out the precision transformation, declaration standardization, and program formatting depicted in Fig. 2.1. The user would type the command line

```plaintext
dapt <s or d> decs_option_file Polish_option_file Fortran_source_file
```

which causes invocation of the monolith tools ISTQT and ISTDT as portrayed in the figure. The first command line argument is “s” or “d” according to whether the transformed program is to be single or double precision. The other arguments are, respectively, the name of a file of options for ISTDS, the name of an options file for ISTPL, and the name of a file containing one or more program units to be transformed. The file Polish_option_file is created by using the script “polx.”

### 2.2.2.3. ISTCE

The portable command executor ISTCE provides a means of maintaining the PFS directory structure in an embedded environment and may be used to perform single or multiple tool invocations. Tutorial examples of the use of tools via ISTCE are given in Appendix A. Each tool has a two character identifier (e.g., lx identifies the lexer/scanner named ISTLX [20]). A tool may be executed by entering its identifier as a command; the user will be prompted for the arguments. Alternatively, some or all of the arguments may be specified on the ISTCE command line. Arguments are separated from the tool name by at least one space and from each other by commas. Commas can be used as space holders for arguments to be input in response to a prompt. Following are two examples of tool use in an embedded environment.

The command

```plaintext
lx #src,1,lx.tkn, lx.cmt
```

would invoke the lexer on the Fortran source in HFS file “src”, sending the token stream and comment stream to PFS files “lx.tkn” and “lx.cmt,” respectively. Error messages and a listing annotated with the number of the first token in each statement would be sent to standard output (pseudo-filename “1”).

The command

```plaintext
pl lx.tkn, lx.cmt, , #pl.opt
```

would invoke the Fortran program formatter on the token and comment streams produced above by the lexer. The user would be prompted for the name of the output file (third argument), as it was not specified. The formatting would be governed by the options in HFS file “pl.opt.” If an argument needs to include a comma (or leading or trailing spaces), then it should be enclosed in double quotes "like this"; a double quote may be entered by typing it twice, "".

Command lines may also make use of the nine local variables &1 to &9 available within ISTCE. Each of these variables can be set to a string using the commands $1 to $9, and these strings can then be recovered using &1 to &9. Substitution of local arguments takes place prior to interpretation of the command; for example, the command sequence

```plaintext
lx src,l lx.tkn,lx.cmt
pl lx.tkn,lx.cmt,pol,pl.opt
```

is the same as

```plaintext
s1 lx
lx src,l,&1.tkn,&1.cmt
pl &1.tkn,&1.cmt,pol,pl.opt
```
ISTCE is also capable of executing script files stored in the directory "/com" or elsewhere. Any command that is not recognized by ISTCE as being either an internal command or a tool scheduling request is assumed to be the name of a script file in the directory "/com." Script files may take up to nine arguments on the command line, interpreted in the script as the local variables &1 to &9 (the variable &0 is the name of the script file itself). Script files may be nested up to ten deep. A script file not in "/com" may be executed using the SC command.

To simplify the writing of script files, and to make them reasonably portable between Toolpack sites that use the embedded environment, a “preferred” method for storing and naming files within the PFS has been devised. This is not a compulsory method, just an approach that has been found to make the use of script files easier. It is recommended that a directory "/pu" (pu is for program unit) be set up to hold Fortran 77 files. Each directory under "/pu" is associated with one Fortran 77 program unit. The program unit is in a file called "src" under the appropriately named directory. For example, the contents of the host file FRED.FTN (containing the program unit Fred) would be stored in the PFS file "/pu/fred/src." All the other files derived from FRED are also stored in this directory. (A list of the “preferred” names is included in Appendix B.) The following is a script file called POL that makes use of this fixed naming convention to polish (unscan and format) a Fortran 77 file:

```
WD /PU/&0
LX SRC., LX.TKN, LX.CMT
PL LX.TKN, LX.CMT, POL, PL.OPT, &2
WD/
```

The script file can be invoked by simply typing

```
POL FRED
```

from within any directory. The WD command changes the current local directory to "/pu/fred." This is followed by the lexer command LX, which is informed that the source code is in a local file called "src": no list file is required. The third command, PL, invokes the polish tool. Its final argument, &2, allows a single option override for the unscanner to be specified, e.g.,

```
POL FRED, LMARGS=10
```

The script file POL changes the local directory to the name specified, then uses the “preferred” names as it invokes the scanner and unscanner tools. On completion, the local directory is set to the root.

Because command lines are expanded before being interpreted, it is also possible to set up script files that use arguments in place of commands; thus the following script file (MYPROG) will perform the operation named on the command line on the three files FRED, TOM, and BILL:

```
&1 FRED
&1 TOM
&1 BILL
```

To polish all of the files mentioned in MYPROG and then to format all their documents, the following two commands can be entered:

```
MYPROG POL
MYPROG PRINT
```

A number of script files are provided with the tool ISTCE; these are described in the ISTCE Users’ Guide.

It should also be noted that the monolithic forms of the tools may also be used from ISTCE.
This would allow, for instance, source-to-source polishing to be specified in the single command line

\[ \text{lp src, pol, pl.opt} \]

2.2.2.4. ISTCI

The Toolpack/l experimental command interpreter provides a novel user interface that does not obey the normal verb-object syntax. The command interface is \textit{object oriented} in the sense that the actions to be performed are implicit in the name of the specified object. Derived objects are regarded as views of an original object: for example, a formatted version of a Fortran program is a view of the original program. All commands to ISTCI are requests to display a view of an object. If the specified view already exists as the result of a prior operation, then it is displayed; if it does not exist, then ISTCI will create it and display it. In effect the user may operate as if all possible views of all available objects exist at any time. The process of creating the requested view is transparent to the user except for the computing time required.

Following are examples of view specifications for ISTCI:

- \texttt{a.f:pol} (the formatted view of the source file \textit{a.f})
- \texttt{a.ref:pol} (the formatted view of all the source files named in the reference file \textit{a.ref})
- \texttt{a.f:pol:scn} (the listing produced from lexical analysis of a polished version of \textit{a.f})
- \texttt{a.f:viw} (a symbol table listing derived from \textit{a.f} by ISTVS)

Further information on the Odin interpreter from which ISTCI was originally derived can be found in [1] and [21]. The operation may be likened to an interactive form of the Unix MAKE facility described in [22].

2.3. Toolpack/l Fortran Language

The language processed by Toolpack/l tools consists of standard Fortran 77 [3] plus some extensions. These extensions, which are summarized below, are accepted by all tools except the standard conformance and portability verifiers (ISTSA and ISTPF). The form of Fortran 77 defined here is known as Toolpack/l Target Fortran 77.

Toolpack/l accepts symbolic names of any length rather than being limited to six characters as specified in the Fortran 77 standard. Additionally, the names may start with a letter or dollar sign and may contain letters (either case), digits, underscores (_), and dollar signs ($). Keywords may be in upper, lower, or mixed case.

An initial line of a statement may contain a tab character in the label field. The tab character implies "tab to column 7." A statement of this form may not contain more than 72 significant characters after expansion of the tab. A warning will be given for each line of this form.

Comments may start with any character. However, a warning will be issued for all comments (except blank line comments) that start with a character other than "C" or "*.*"

Trailing blanks on a line need not be supplied (e.g., blanks within a character string broken over two or more statement lines).

The following data types are recognized in addition to those defined in the standard: DOUBLE COMPLEX, INTEGER*2, LOGICAL*2, LOGICAL*1, REAL*16, Hollerith data type.

Additionally, the following data types are recognized as equivalent to standard data types:

1. COMPLEX*8 is equivalent to COMPLEX
2. INTEGER*4 is equivalent to INTEGER
3. LOGICAL*4 is equivalent to LOGICAL
4. REAL*4 is equivalent to REAL
5. REAL*8 is equivalent to DOUBLE PRECISION
The following standard intrinsic functions accept a double-complex argument: CMPLX, DBLE, INT, REAL, ABS, COS, EXP, LOG, SIN, SQRT. ABS returns a double-precision result; COS, EXP, LOG, SIN, and SQRT return double-complex results.

The following additional intrinsic functions are allowed:

1. CDABS — the specific name for ABS applied to a double complex datum.
2. DCMPLX — creates a double complex datum. The argument may be a pair of numbers that must be integer, real, or double precision or may be a complex or double-complex number.
3. DCONJG — the double-complex analog of CONJG.
4. DIMAG — the double complex analog of AIMAG.

The following standard intrinsic functions will accept a half-precision integer argument: INT, REAL, DBLE, CMPLX, DCMPLX, ABS, MAX, MIN, MOD, SIGN.

The following standard intrinsic functions will accept a quadruple-precision real argument: INT, REAL, DBLE, CMPLX, DCMPLX, NINT, ANS, COS, EXP, LOG, MAX, MIN, MOD, SIGN, SIN, SQRT, ACOS, ANINT, ANINT, ASIN, ATAN, ATAN2, COSH, DIM, LOG10, SINH, TAN, TANH.

The document Toolpack/1 Target Fortran 77 [3] details these extensions and lists some restrictions on their use.
3. Tools Available in Toolpack/1

The tools provided in Toolpack/1 come from a variety of sources. Many of the tools were designed specifically for the TIE environment and cooperation with other tools. Some tools were produced for the Toolpack project before the TIE definition emerged and required subsequent modification to work with TIE. Others were adapted from the LBL/Unix Software Tools [23], as implemented by the project described in [24]. This task was made simpler by the inclusion of primitive functions from [24] in the TIE definition. Other tools are simply useful Fortran-based tools that have been modified to make use of the TIE portability base. These latter tools vary in the extent to which they have been integrated into the tool suite and are provided in the second release as portable Fortran tools of a general nature. Work continues to integrate certain of these tools more completely into the tool suite. It is hoped that Toolpack/1 will continue to stimulate the creation of new tools and the conversion of old tools to the Toolpack framework.

The tools may be split, functionally, into three areas—general, documentation, and Fortran processing, though some tools have uses in more than one category. The following subsections describe briefly the more important tools available; a full list of tools is given in Appendix B.

3.1. General Tools

ISTDC
Is intended mainly for comparing files of numeric values, though it is also capable of comparing files with embedded text.

ISTET
Expands tabs. It processes a file to remove tab characters and replace them with spaces.

ISTFI
Finds all the include files that a file needs.

ISTGP
Searches multiple files for occurrences of a regular expression.

ISTHP
Provides limited help information about tools. The user enters regular expressions are matched against keyword lists and trigger the outputting of text. The information in the help system is based on that provided in Appendix B to this document.

ISTMP
May be used to preprocess a file. The processor provides macro replacement, inclusion, conditional replacement, and processing capabilities to enable complex file processing to be performed. This tool also has uses in Fortran processing and documentation preparation.

ISTSP
Breaks up a file using split markers that specify the individual files and their names. This is the TIE conforming version of the SPLIT utility.

ISTSV
Saves and restores subtrees of the PFS.

ISTTD
Compares two test files.

ISTVC
Sets up and recalls versions of a file either by version number or date/time.

3.2. Documentation Generation Tools

ISTAL
Can be used to generate Fortran specific information from intermediate files created by other tools. The information that can be generated includes callgraphs, cross-reference listings, segment execution frequencies, and symbol information. ISTAL can also strip marked areas of comments from source code for inclusion into documentation. Commands to ISTAL are embedded in
the document intermixed with standard text. ISTAL will also produce information formatted for direct display when used interactively.

**ISTCB**
Acts as a comparator. Change bar requests are inserted into a document wherever it is found to differ from the original.

**ISTDX**
Extracts previously marked documentation from a Fortran program.

**ISTRF**
Takes in unformatted text and embedded formatting commands and produces formatted output text suitable for immediate printing. This tool is similar to the Unix roff facility [25].

### 3.3. Fortran 77-Oriented Tools

**ISTAN**
Allows the collection of runtime execution frequency information. This tool was developed out of software described in [26].

**ISTCN**
Changes names in a Fortran 77 program at the token stream level.

**ISTCR**
Changes names in a Fortran 77 program at the symbol table level.

**ISTDS**
Rebuilds the declarative parts of a Fortran 77 program unit according to a programmable template.

**ISTED**
Permits line-based editing. This tool is derived from the Software Tools ED editor [24]. It has additional provisions to assist the Fortran 77 programmer, including a macro facility with all keywords pre-defined. The editor includes programming capabilities and some intrinsic functions.

**ISTFD**
Works on token streams rather than source text. This comparison tool can ignore layout and (optionally) comments.

**ISTFR**
Converts the format of real, double precision, and complex constants to a standard form.

**ISTGI**
Changes all intrinsic function references within Fortran 77 code to their generic forms (where this is legal or possible).

**ISTIN**
Inserts the contents of include files into a file marking the bounds with source embedded directives.

**ISTJS**
Joins strings in FORMAT statements, optionally converts holleriths to strings, and can convert X descriptors.

**ISTLS**
Changes long Fortran names to the ANSI standard of 6 characters. When a name longer than 6 characters is encountered, the tool prompts the user for a replacement. This mapping is recorded in a file so that any further occurrences of that long name can be substituted automatically.

**ISTLX**
Scans Fortran 77 source text and produces a token stream and separate comment file. The token stream does not include information about layout but provides identification of keywords and string/numeric values within the original source.
ISTME    Manipulates Fortran expressions to reduce the depth of the intermediate value stack.

ISTMF    Takes as input two Fortran programs, one in source form, the other in token stream form, and produces a merged output source. The differences are marked with comments.

ISTMP    May be used to process Fortran source text. The “eval” expression evaluation facility of ISTMP includes the Fortran intrinsic functions in addition to arithmetic facilities. As ISTMP is loaded with the host libraries the intrinsic functions will be the same as those that the processed program will use, this can be useful in the expansion of fixed constants within portable tools (this tool also has uses in general macro expansion and documentation preparation).

ISTPF    Acts as a portability verifier for Fortran 77 based on the Fortran 66 tool PFORT.

ISTPL    Serves as a Fortran 77 polisher or pretty printer. Options for this tool may be set or changed using the separate menu-driven polish option editor ISTPO.

ISTPP    Ensures the consistency of PARAMETER statements over Fortran program units.

ISTPT    Converts between REAL and DOUBLE PRECISION, changing intrinsics, constants, formats, and declarations.

ISTSA    Checks conformance of code to the Fortran 77 standard.

ISTST    Takes as input a Fortran program, analyzes the flow of control, and rebuilds the program in a structured form.

ISTUN    Reverses the effect of ISTIN. It removes included files that are marked by SEDs [2], placing include statements instead.

ISTYP    Builds parse trees.

ISTYF    Flattens parse trees.

A special subgroup of the Fortran 77 tools comprises three DO loop unrolling tools:

ISTCD    Combines consecutive DO loops.

ISTSB    Recombines and substitutes expressions.

ISTUD    Unrolls DO loops.

3.4. Facilities for Tool Construction

In addition to the tools provided on the tape, there is support for the user to generate local tools or to convert existing tools to Toolpack conventions. There are a number of reasons that such support may be desirable:

(1) The user may wish to provide a tool capability not included in Toolpack/1.
(2) The user may already have a local tool that he would like to integrate into the local Toolpack system.

(3) The user may wish to integrate host utilities (e.g., compilers, loaders) into the local Toolpack system.

All of these approaches are possible and are facilitated by provision of the following in Toolpack/1:

(4) Tool Writers' Guide [27], which gives general information on writing software that conforms to the Toolpack conventions.

(5) Definitions of all the TIE routines.

(6) Supplementary libraries to provide additional string- and table-handling capabilities as well as access to token streams, parse trees, and symbol tables.

(7) The tool fragments themselves: thus the tool writer need no longer concern himself with the need to produce, or unscan, a token stream; these capabilities already exist.
4. Using the Tools

As pointed out in Section 2.1.2, the manner in which the user invokes the tools depends on the user/Toolpack interface at his or her Toolpack/1 installation. Here we first give some further examples of Toolpack/1 usage when the user/Toolpack interface is ISTCE. Then we give some advise about tool invocation sequences that accomplish certain Fortran analysis and transformation tasks. The advice is intended primarily for anyone designing a user/Toolpack interface but also serves to guide a user who may wish to invoke each individual tool. Note also that a tutorial guide, describing complete Toolpack/1 sessions is given in Appendix A.

4.1. Examples of Tool Usage

4.1.1. Polishing a Fortran 77 Source File

Using ISTCE, a Fortran 77 source file may be polished in a number of ways. The four most useful ways are described below.

- The source file may be polished by invoking the scanner (ISTLX) and polisher (ISTPL) individually, using commands of the form shown below. This technique is suitable for a one-off polish operation or where it is not desirable to copy the source into the PFS. This example creates a polished form of the host file test.ftn, and places it in the host file test.pol. The intermediate files are deleted.

  \[
  \text{lxl \#test.ftn, \-, test.tkn, test.cmt} \\
  \text{pl test.tkn, test.cmt, \#test.pol, pl.opt} \\
  \text{df test.tkn, test.cmt} 
  \]

- Using the monolithic tools, this may be replaced with the following commands. Again the resulting polished code is in the file test.pol.

  \[
  \text{lp \#test.ftn, \#test.pol, pl.opt} 
  \]

- The script file POLISH provided with ISTCE may be used. For POLISH to work the source file must be stored in the PFS in the preferred manner described in the ISTCE documentation. To import TEST.FTN into the PFS, the script file IMPORT can be used:

  \[
  \text{IMPORT TEST, \#TEST.FTN} 
  \]

  This importation will also allow the other ISTCE script files to be used on the source file. The file may now be polished by typing

  \[
  \text{POLISH TEST} 
  \]

- If the file has been imported as above, then polishing may be performed by the general script file PROCESS as follows:

  \[
  \text{PROCESS TEST} 
  \]

  The resulting polished code in this case is in the file lx.pol.

4.1.2. Complex Processing of a Fortran 77 Source File

This example shows how a Fortran 77 source file may be subjected to more complex processing than simply polishing it. Although the tool fragments can be invoked individually, as for the previous example, this is less desirable for complex processing because of the number of stages involved.

A script file to perform several types of processing on a Fortran 77 source file is provided. The script file, called PROCESS, can be used to polish the source file, transform its precision, standardize its declarations, or replace non-generic intrinsic function references with their generic forms. PROCESS
Toolpack/1 - Introductory Guide

does not delete intermediate objects, e.g., token streams, and will always test to see if a tool invocation is required before performing it. For example, if the scanner output has a more recent time stamp than the source code, then there is no need to rescan. PROCESS is run as follows:

```
PROCESS <file> [ , <operation> ] [ , <polish option override> ]
```

Polishing is done according to the options in the file PL.OPT, but one option can be overridden using the PROCESS command line argument. The operations are as follows:

- **PL**: Polish the source code leaving the result in LX.POL (this is the default operation if none is specified).
- **GI**: Replace non-generic intrinsic function references in the source code with their generic versions, leaving the result in GI.POL (this is compilable source code).
- **PT**: Transform the arithmetic precision of the source code (between single and double precision or vice versa) and then standardize the declarations. The result is left in PT.POL (this is compilable source code).
- **DS**: Standardize the declarative part of the source code, leaving the result in DS.POL (this is compilable source code).

The operation of PROCESS and all other script files may be followed by using the echo flag in ISTCE (the "-e" command).

### 4.13 Handling Documentation

Toolpack provides a number of tools to aid in producing and processing documentation. Documentation may be generated at three levels (as well as simply in text files) for processing using Toolpack facilities.

- **TXT**: This form is suitable for printing directly.
- **DOC**: Files of this type contain text interspersed with ISTRF formatting commands. ISTRF is a text formatter similar to RUNOFF or the Unix facility troff/eqn. ISTRF can process a DOC file to produce a TXT file that contains only formatted text. DOC files may also contain the simple documentation macros defined in the file TDM and described in the ISTRF Users' Guide.
- **TDM**: Files of this type contain text mixed with both ISTRF commands and the more comprehensive ISTRM-processed Toolpack Documentation Macros. This form is not often used: it was provided to extend the capabilities of ISTRF for Release 1.
- **ANL**: Files of this form contain text mixed with ISTRF formatting commands and commands to the documentation generation aid ISTAL. Files of this form are processed to DOC files by ISTAL.

The DOC form is suitable for the production of all general-purpose documents, while the ANL form is specifically designed to produce documentation of Fortran 77 source code by producing source-specific inserts for the document (e.g., call graphs, cross-reference listings, and dynamic segment execution frequency counts). ISTAL is also capable of stripping specially marked comment blocks from source code files and inserting them into the processed document.
4.2. Tool Invocation Sequences

First, some general advice:

1. Precision transformation should precede declaration standardization.
2. After using ISTGI, it is advisable to flatten the parse tree and rebuild it later if required.
3. The small manipulative tools ISTME, ISTJS, and ISTFR should be applied immediately prior to polishing.

The following are common forms of analysis or transformation. For each type of operation, one or more sequences of tool invocations are suggested. Selection of an appropriate tool sequence will depend on the resources and tool fragments available.

Polishing: The simplest form of transformation is polishing, or pretty printing. The polish utility (ISTPL) is capable of a number of additional processing functions including relabeling and moving FORMAT statements. These processing options may be selected using the option editor ISTPO. The tool sequence is ISTLX-ISTPL. This tool sequence may be shortened to ISTLP if the monolithic tools are available.

Full-Form Conformance: This form of processing is for when code is to be fully conformed to a standard layout. Because of the extra resources required to perform this processing, it would normally be used only occasionally on the code, partial conformance being applied more regularly. The tool sequence suggested for this form of processing is

ISTSX-ISTSYP-ISTSIG-ISTSYP-ISTSME-ISTSFR-ISTSJS-ISTSYP-ISTSST

with a pre-arranged set of options for ISTPL. The first two tools ISTLX-ISTSYP may be replaced by the monolith ISTLY.

Partial-Form Conformance: This form of processing is for reconforming code after edits or changes have taken place. The tool sequence advised is ISTLX-ISTSYP-ISTSDS-ISTPL, with a pre-arranged set of options for ISTPL. This tool sequence may be replaced by ISTQD if the monolithic tools are available.

Precision Transformation: The arithmetic precision transformer may be used to transform software between single and double precision (or vice versa). This is useful for generating an alternative precision form of software, as only one precision needs to be maintained, the other always being a mechanical derivative. The simplest form of this processing is ISTLX-ISTSYP-ISTPT-ISTPL (monolithic form is ISTQP). However, it is better if the precision transformation is followed by declaration standardization (sometimes ISTPT inserts duplicate INTRINSIC declarations). This form is performed by the tool sequence ISTLX-ISTSYP-ISTPT-ISTSYP-ISTSDS-ISTPL which may be replaced by the monolithic tool sequence ISTQT-ISTDT.

Static Analysis: A number of stages of static semantic analysis may be performed. The sequence for checking conformance of software to the Fortran 77 standard is ISTLX-ISTSYP-ISTSFSA (monolithic form is ISTLA). This may be followed by an additional processing step to check conformance to the PPORT 77 standard: ISTLX-ISTSYP-ISTSFSA-ISTSFPF (monolithic form is ISTLA-ISTSFPF). If the fragmented forms are used, then additional material may be derived using the viewing tools ISTVS (view symbols), ISTVT (view tree), ISTVA (view attributes), and ISTVW (view warnings). Additional information may also be derived and viewed by using ISTAL interactively (input filename “0,” output filename “1”).

Dynamic Analysis: Dynamic analysis is currently available only through ISTAN. A suggested sequence of tools for this function is given in the example in Appendix A.

Unrolling DO Loops: DO loops may be unrolled mechanically to make the Fortran code more easily optimized by compilers on vector processors. The tool fragments concerned with unrolling DO loops are ISTUD, ISTCD, and ISTSB. These three tools need to be invoked in the sequence outlined in [30].
When processing is complete, the tool sequence ISTYP-ISTDS-ISTPL should be invoked to correct the declarations and produce compilable code (relabelling may be performed by ISTPL). This tool sequence may be replaced by the single monolithic tool ISTDT.
5. Before You Start

5.1. Nature of the Support for Toolpack/1

Toolpack/1 is not a commercial product but is the result of a collaborative project which has received U.S. and U.K. public funds. It is being made available by NAG as a public distribution service; that is, the software is provided for a handling charge only, intended to cover tape copying, documentation, and other administrative costs.

Public domain ventures like Toolpack are intended to provide useful facilities and stimulate interest in further technological development. In the case of Toolpack the facilities are the software tools, and the field of interest is software technology—more specifically, enhanced support for the Fortran 77 programmer.

Though considerable care has been taken in preparing Toolpack/1 for release, it is not to be regarded as a commercially supported product and may not be suitable for providing the sole support for large-scale, commercially valuable software projects. In particular, all prospective users of Toolpack/1 should note the disclaimer given at the front of this guide.

If your site has arranged some support service with the tape distributors, this should be noted in Section 6 of this guide.

5.2. Prospect for Further Development

It is hoped that the second release of Toolpack/1 will continue to stimulate further development and discussion of tools and programming environments for the Fortran programmer. To this end the following are being undertaken:

1. A redefinition of TIE will take place to improve its efficiency and make it easier to install, particularly for those sites wishing to use only part of the available capabilities. The redefinition will take into account the installation and tool-writing experience gained from the first and second releases.

2. The documentation for Toolpack/1 includes a brief tool writers guide to assist users in writing their own tools. Additional tutorial material for tool writers is being prepared. The generation of Fortran manipulation tool is greatly eased by the existence of separate tool fragments for predefined tasks (e.g., scanning and unscanning) and documented access functions in the supplementary libraries.

3. Tools written using TIE (or converted to use TIE) may be submitted to NAG for consideration for inclusion, as public domain products, in future releases or revisions of the Toolpack tape.

4. Additional installed versions of TIECODE for particular host environments are being developed.

5.3. Other Documents to Read

The prospective users of Toolpack tools should first refer to Section 6 of this Guide to learn of the local arrangements for using the tool suite.

The reader requiring further information on Toolpack/1 is referred to Appendix D which gives the complete list of tool and TIE users' and installers' guides.

5.4. What to Do Next

What you should do next depends on your interest in Toolpack/1.

Toolpack Installer. If you are the Toolpack Installer for your site, then the next step is to read the printed installation documentation provided with the tape. The documents will help you to decide which implementation of TIE should be installed. If there is not a suitable host-specific implementation of TIE available, then it will be necessary to install the portable, TIECODE, implementation. Once the tools are running, do not forget to provide the information for Section 6 of this Guide and printed copies of tool documents for users to access. If you are able to
install TIECODE, or produce a customized TIE implementation, please inform the tape distributor so that the information can be passed to other sites.

**Toolpack User.** If you are interested in Toolpack/1 primarily as a user, then it is suggested that, having consulted Section 6, you try the tutorial examples given in Appendix A as your first Toolpack/1 sessions.

**Tool Writer.** If you are a tool writer, then you should first become familiar with Toolpack/1 as a user. If you plan to write a major Fortran processing tool that you intend to make available to other Toolpack/1 users, it may be worth contacting the addresses given in the next section in case the work is already under way elsewhere or further advice can be offered.

### 5.5. Contact Addresses

The following contact addresses may be used for Toolpack related matters. Although every reasonable effort will be made to respond to written queries, you should note the limits of the support offered to your site.

**U.S.A., Canada, and Mexico**

Numerical Algorithms Group, Inc.
1101 31st Street, Suite 100
Downers Grove, IL 60515-1263
United States

**Rest of the World**

Numerical Algorithms Group, Ltd.
NAG Central Office
Mayfield House
256 Banbury Road
Oxford OX2 7DE
United Kingdom
6. Local Installation Information

This section gives information specific to your local Toolpack/1 site. It is obviously not possible to include this information in the distributed form. However, the following examples based on NAG’s own uses of Toolpack indicate the kind of information that should be provided locally by the Toolpack installer.

6.1. Example 1: Toolpack on the Apollo, NAG Central Office

The Toolpack system on the Apollo uses the standard TIECODE implementation of TIE. To use Toolpack on the Apollo, the following actions are required:

1. Create a DIR and DATA portable filestore pair in your intended working directory. These are created using "/toolpack/tie/support/mkpfs."

2. The executable forms of all tools are kept in the directory "tools." This may be searched automatically by changing the command search rules

(Aegis command csr ~/com /com /tools)

3. Use the tool ISTSV to recover the files from "bob/store." This will ensure that all the files required by tools (e.g., TDM required by ISTRF) are placed in appropriate directories. It will also read in ISTCE script files.

4. You may now use the tools in the working directory containing DIR and DATA. Tools may be invoked either by using ISTCE or directly (as the Apollo uses spawning, there is no difference between an embedded and a stand-atop regime).

If you wish to generate your own tools, you will need the following information:

5. The TIE library (embedded regime) can be found in "/toolpack/tie/tie_library."

6. To macro expand a .MAC file use the command file "/toolpack/control/mac."

7. The supplementary libraries can be found in subdirectories of "/toolpack/tie."

8. Note that block data has to be separately specified to the binder. For an example of a tool using supplementary libraries binding see "/toolpack/toolsfistyp/makefistyp."

WARNING: If a tool or command interpreter abnormally aborts (e.g., you use control/Q to terminate it), then DIR and DATA may be corrupted beyond recovery. Use ISTSV to keep a backup of important files.

Documentation may be found in the directory "/toolpack/doc." If you have any problems, contact members of the Software Engineering Group.

6.2. Example 2: Toolpack on the ICL Perq, NAG Central Office

The Toolpack system on the Perq uses the C-coded TIEC implementation of TIE. To use Toolpack on the Perq, the following actions are required:

1. Set up a directory called TOOLPACK in your current working directory.

2. See the documentation NP1308 [28].

Because of lack of disk space, Toolpack tool development on the Perq is not advised.

Copies of all the documentation may be obtained from the Software Engineering Group, who should also be contacted if any problems are encountered.

6.3. Example 3: Toolpack on OUCS/VAX2

The Toolpack system on OUCS/VAX2 uses the mixed Pascal/ForTRAN implementation of TIEVMS. This implementation, as well as usage information, is given in the Toolpack document on TIEVMS [29].
In this implementation, arguments may be passed to the tools in the command line when they are scheduled directly or via command files.

Copies of all the documentation may be obtained from the Software Engineering Group, who should also be contacted if any problems are encountered.
GLOSSARY

Words in this form are defined elsewhere in the glossary.

Attribute
A feature of a program unit element discovered by semantic analysis. Attributes may be stored in a separate attribute table or in the symbol table.

Command Executor
Command Interpreter
A software tool that interprets a user-written command line and schedules the invocation of software tools to produce a user-requested result.

Command File
Script File
A file containing commands for a command interpreter or software tool that can be used as an alternative to interactive input.

Documentation Generation Aid
A software tool aimed at automatically deriving or processing machine-readable documentation.

Dynamic Analyzer
A software tool that performs an analysis of a program unit by execution of the program unit under analysis. Such tools often consist of an instrumentor and a runtime library that controls the execution, or collects data about the program unit under test while it is executing.

Embedded Regime
IST regime
The operating regime used by software tools that are embedded within IST. The flow of control is governed by a command interpreter, and both the PFS and HFS are available.

Fragmented
Fragmentation
The process of splitting a software tool into independent, reusable fragments that can then be reassembled in sequence to obtain the desired functionality.

Fully Qualified
A term used to describe a pathname that defines all intermediate directories between the specified object and the root directory.
HFS
Host File Store
The file store of the host system. This file system is always available to Toolpack tools. Because the directory structure and file name format of the HFS are host-dependent, an additional PFS (portable file store) is also provided.

Host System
A physical computer, together with an operating system, on which the TVM (Toolpack Virtual Machine) may be installed.

Instrumentor
A software tool that inserts instrumentation statements into a program unit so that it can be dynamically analyzed.

Integrated
A term used to describe a tool suite in which software tools can operate on a common set of objects in a consistent manner, and can be combined to form powerful tool sequences.

IST
Integration System for Tools
The tool dependency data, a tree-structured file store, and a command interpreter, that together provide the means for unifying the software tools of Toolpack.

ISTCI
IST Command Interpreter
A particular Fortran-written, TIE - conforming command interpreter based on Odin.

Leaf
A node in a tree structure that has no children or branches.

Lexer
Scanner
Lexical Analysis
A software tool that performs lexical analysis (or scanning), i.e., that converts source text into a token stream.

Local Directory
The directory in which pathnames that are not fully qualified are assumed to start.
Maxpath
A macro defining the maximum length of a pathname.

Mechanical Derivation
The process of deriving one form of software or information from another by the use of software tools without the intervention of the user or programmer.

Monolithic
Forms of software tools where multiple small tool fragments have been put together into a single executable process for efficiency or improved usability.

Node
An element of a tree structure. Although a node can be an element of any tree (e.g., a TSFS) it normally refers to an element of a parse tree. There are two special types of nodes: the root and leaf.

Object-Oriented
Object
A form of command that places the name of an object first, instead of the more normal verb-object form. The object is a view of information held in the file system that may be modified by the available software tools.

Odin
An advanced, C-written command interpreter.

Parser
Syntactic Analysis
A software tool that performs syntactic analysis of a program unit. In the case of Fortran 77 a parser ensures that the statements in the program unit are correctly formed and ordered. The result of a parsing operation may be a report of errors or a parse tree and symbol table.

Parse Tree
A low-level representation of a program unit in which each element of the program unit is represented by a parse tree node. The nodes are arranged as a tree structure rooted on a program node. Elements of the program unit correspond to nodes in the tree such as subroutine_node, array_reference_node. The parse tree will contain references to a symbol table where the stored strings associated with some nodes are kept.

Pathname
The name of a file or directory consisting of a (possibly empty) list of directory names separated by "/" and ending in the name of a directory or file.
PDS
Portable Directory System
The software that provides the directory structure for the PFS. The PDS is used in the TIECODE implementation to provide the TSFS.

PFS
Portable File Store
The combination of a PDS and a PIOS that provides a portable TSFS.

PIOS
Portable I/O System
The software that enables input/output to PFS files in conformance with TIE definitions. PIOS is used in the TIECODE implementation.

PU
Program Unit
A single, individually compilable part of a Fortran 77 program.

PUG
Program Unit Group
A collection of program units.

Programming Environment
A complete system for supporting the programmer during the development and maintenance of his software.

PUG List
A list of program unit names.

Regular Expression
A pattern-matching definition (see Appendix B).

Root
The single root node in any tree structured system to which all other nodes are connected, directly or indirectly.

Semantic Analysis
The process of collecting semantic information about a program unit.
Software Tool
Programming Aid
Mechanical Aid
A program to assist in developing, testing, maintaining, or distributing computer software.

Source-Embedded Directives
SED
A Fortran comment in a standard format that includes an identification of the tool that inserted it or that is expected to use it. Usually SEDs also contain a command, switch, or parameter to that tool.

Stand-Alone Regime
The tool operating regime in which flow of control is governed by the host system and for which the HFS is available but the PFS is not.

Stand-Astride Regime
The tool operating regime in which flow of control is governed by a command interpreter and for which the HFS is available but the PFS is not. There is an implicit assumption that the HFS is a TSFS.

Stand-Atop Regime
The tool operating regime in which flow of control is governed by the host system and for which both the PFS and the HFS are available.

Static Analyzer
A software tool that performs an analysis of a program unit without execution of the program unit under analysis. Such tools may also be transformers.

Supplementary Library
A library of additional capabilities for use with TIE.

Symbol Table
A data structure associated with a parse tree and used for storing the saved strings, or symbols, associated with parse tree nodes. The symbol table may also be used for storing the attributes of the symbols.

TIE
Tool Interface to the Environment
A set of definitions that completely characterize the TVM, or Toolpack Virtual Machine.

TIECODE
A Fortran 77 implementation of TIE, portable except for a subset of host system - dependent routines.
Token
Lexeme
Atom
An identifier for a defined keyword or language element. Tokens are created as a result of the *lexical analysis* of source text. A sequence of tokens created from a single piece of source text is normally called a token stream.

Tool Dependency Data
A directed graph of *software tool* dependencies used by a *command interpreter* to schedule the sequence of *software tool* invocations required to produce a result requested by the user.

Toolpack
A portable, integrated, and extensible suite of Fortran 77 *software tools*.

Toolpack/1
The public release of software resulting from the *Toolpack* project.

Toolpack Installer
The person at a site responsible for the installation, or mounting, of the *Toolpack* software, including the *TIE* library, the *supplementary libraries*, and the *software tools*. This person is also responsible for providing the site-dependent information that forms Section 6 of this Guide.

Transformer
A *software tool* that transforms its input into an alternative state, or representation. An example of a transforming tool is a *lexer*.

Tree
Tree Structure
A form of data structure that may be used to show the association between data elements. A tree starts at a single *root node*. This node may have one or more children, or branches, each of which may in turn have one or more children and so on. In this way the structure is built up like a family tree with the root conceptually at the top (i.e., the oldest common ancestor). The language of the family tree is often used, so one node may be said to be the parent of another. A node without any children is called a *leaf*.

TSFS
Tree-Structured File Store
A file system possessing plain files that contain arbitrary text, directories that list the names of files (either directories or plain files), and a root directory that originates the tree.

TVM
Toolpack Virtual Machine
A set of functions for input/output, file management, string handling, and bit manipulation, together with a character set, a *PFS*, and access to the *HFS*. The functions are defined by *TIE* and are available...
to a tool as calls to routines in an implementation of TIE. The tool developer writes tools for the TVM, thereby making them independent of the host system.

Unscanning
Polishing
The opposite process to lexical analysis in which a token stream is converted to source text.

View
A mechanical derivation of an object created by ISTCI or Odin.

VFS
Virtual File Strategy
A feature of Odin that allows all files that could be created to be available to the user without a distinction as to whether or not they already exist. If a file is requested, it is either found or created; creation is invisible to the user except in the increased access time required.
REFERENCES


(2) Cohen, M., and Iles, R., Toolpack/1 Target Fortran 77. Toolpack/1, Release 2.1, NAG Publication: NP1313.


(13) Iles, R., What is TIE? Toolpack/1, Version: 2.1, NAG Publication: NP1281.

(14) Iles, R., TIE Installers' Guide. Toolpack/1, Version: 2.1, NAG Publication: NP1278.


(20) Iles, R., ISTLX Users' Guide. Toolpack/1, Version: 2.1, NAG Publication: NP1289.


Appendix A - Tutorial Examples

This appendix contains tutorial examples of the use of Toolpack/1 tools to enter and process some Fortran 77 source code. The user/Toolpack interface is assumed to be ISTCE. For a description of a user/Toolpack interface consisting of UNIX scripts, see [31].

The following points are briefly covered:

1. General use of ISTCE to move around and examine the portable filestore.
2. Use of the Toolpack/1 editor to enter and modify Fortran 77 code, including on-line lexical analysis.
5. Examining of information derived by static analysis.
6. Dynamic analysis of a Fortran program.

The tutorial should initially be worked through in its entirety (this will not take too long); later, it may be used for reference.

The tutorial assumes that the following applies to your local Toolpack site:

1. The command executor (ISTCE) is available, and the additional files supplied with it (in the file "script") have been recovered using ISTSV.
2. The tools are loaded in the embedded regime (i.e., the tools may be scheduled by ISTCE and the portable file store is available).
3. The editor (ISTED) is available and has the scanner installed. If this is not the case, then the initial part of the exercise dealing with the editor may be skipped.

No use is made of combined, or monolithic, tool fragments (e.g., ISTLP, the combined scan/polish utility). This is in order to emphasize the fact that tool functionality is fragmented.

Example 1

The user should start by logging in to the host system and running the Toolpack/1 command executor ISTCE.

General Use of ISTCE

When ISTCE is running, it will give the "ce:" prompt. Enter the help command "?” to see what direct commands ISTCE allows you:

```
ce: ?
COMMENTS ARE:
#                  COMMENT LINE
?                  LIST COMMANDS
!                  LIST TOOLS
CD <Directory name> CREATE DIRECTORY
CF <File name>     CREATE FILE
CL                 CLEAR FLAG
CP <File name>,<File name> COPY FILE
DD <Directory name> DELETE DIRECTORY
DF <File name>     DELETE FILE(S)
EC                 TOGGLE ECHO FLAG
EX                 QUIT
```
This may be followed by the command "!" to see which tools are currently recognized by ISTCE. Note that ISTCE is not case sensitive. Commands and tool names may be entered in lower, upper, or mixed case. Note also that this table does not mean that the tools have actually been installed on your host system, only that ISTCE will recognize the name.

```
  ce: !
  ... TOOL NAMES RECOGNIZED ...
  al an cb cd cn cr
dc ds dt dx ed et
  fd fi fl fp fr gi
gp hp in jf js la
lp  ls lx ly me mf
mp ni nl p2 pf pl
  po pp pt qd qp qt
rf sa sb sp st sv
td ud un va vc vs
vt vw x1 x2 x3 yf
yp
```

To see what is in the local directory, enter the "ld" command without any parameters:

```
  ce: ld
 /
 NAME       ACC TYPE  <- last modified ->
com        Seq Dir   10:53:28  06 MAY 1986
 tdm        Seq File  10:18:54  06 MAY 1986
zdflib      Seq File  10:19:11  06 MAY 1986
 pu         Seq Dir   10:19:26  06 MAY 1986
```

The first line of the listing (/) tells you which directory is being listed, in this case the local directory (as no parameter was specified) which happens to be the root. In the example listing shown, there are two sequential files and two directories. The files are data required by ISTMP and ISTRF; the directories contain script files ("/com") and the program units, or Fortran code ("/pu").

As we are going to create and manipulate some Fortran, we need to move into a new directory under the existing "/pu" directory (this is the preferred system described in the ISTCE Users' Guide). To do this, it is necessary to move to the "/pu" directory and create a new directory:
This example shows that there is already a directory called "test," so let us call ours "trial":

As you can see, we have set the local (or working) directory to be "/pu/trial." The directory is currently empty.

Use of the Editor

Let us now enter our source text using the editor.

The editor informs us that the file did not exist, so now we can enter our program:
PROGRAM TEST

REAL A(10), MEAN

PRINT *, 'This is a test program.'
PRINT *, 'Please input 10 real numbers:'
READ *, A
X = MEAN(A, 10)
PRINT *, 'The mean is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
CALL SQUARE(A, 10)
PRINT *, 'The squares of the numbers are:', A
X = SUM(A, 10)
PRINT *, 'The sum of the squares is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
STOP
END

Having entered the main program, we now enter its associated routines as well. The next routine can be added using the editor's tabbing capability:

SUBROUTINE SQUARE(A, N)

INTEGER I, N

DO 100 I = 1, N
A(I) = A(I) * A(I)
100 CONTINUE
RETURN
END

And now, using the editor's tabbing and built-in macro facilities:

SUM = 0
DO 100 I = 1, N
SUM = SUM + A(I)
100 CONTINUE
RETURN
END
Finally, for the rest of the code:

```
0: $a
    REAL FUNCTION MEAN(A,N)
    REAL A(N)
    INTEGER N, I

    MEAN = 0
    DO 100 I = 1, N
100 MEAN = MEAN + A(I)
    MEAN = MEAN/N
    END

    REAL FUNCTION STAND(A,N)
    REAL A(N)
    INTEGER N
    REAL M, MEAN

    M = MEAN(A,N)
    STAND = 0
    DO 100 I = 1, N
100 STAND = STAND + (A(I) - M)**2
    STAND = SQRT(STAND)
    IF (STAND .EQ. 0) RETURN
    END
```

All the code is now entered. In case any editing errors are made later, let us actually write the code out to the file now before looking at it (using the browse command) to make sure it is all right (note that the name “src” here refers to the file “/pu/trial/src”):
0: wri
58 lines written to file src
0: 1 bro

PROGRAM TEST

REAL A(10), MEAN

PRINT *, 'This is a test program.'
PRINT *, 'Please input 10 real numbers:'
READ *, A
X = MEAN(A, 10)
PRINT *, 'The mean is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
CALL SQUARE(A, 10)
PRINT *, 'The squares of the numbers are:', A
X = SUM(A, 10)
PRINT *, 'The sum of the squares is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
STOP
END

SUBROUTINE SQUARE(A, N)
REAL A(N)
INTEGER I, N

0:

The browse command lets us look at a "window" of information at a time. We can change the size of the window easily in the browse command itself. Let us turn on line numbering, then list a shorter window:

0: ?= 0: 1 bro 10

PROGRAM TEST

REAL A(10), MEAN

PRINT *, 'This is a test program.'
PRINT *, 'Please input 10 real numbers:'
READ *, A
X = MEAN(A, 10)
PRINT *, 'The mean is:', X
X = STAND(A, 10)

0:

We can check that what we enter is legal Fortran 77 (at the lexical level) using the built-in scanner in the editor. The following commands will scan a single statement for us. We can edit it to be illegal, rescan, detect the error, and then correct the error before continuing:
This technique can be useful to detect simple typing errors when entering code. The following command will scan the whole buffer to see if you have entered the code correctly; if you haven't, make corrections using the substitute command (see the ISTED Users' Guide):

```
0: par b
```

Now rewrite the code to the file (using “wri”), and leave the editor. The “ld” command will confirm that the file now exists:

```
0: wri
58 lines written to file src
0: qui
ce: ld
/pu/trial
NAME                        ACC TYPE            <- last modified ->
src                          Seq File 11:47:40 06 MAY 1986
ce:
```

The source file may be displayed to the terminal using the “sh” command, or may be copied to the
Polishing a Fortran File

Since the program source, as entered, was poorly formatted, the next step might be to tidy it up using the polish utility. First, let us create a polish option file using the option editor, ISTPO. The following command will run the polish option editor, which is interactive and menu driven (the command "m" will print the current menu). Help on option setting is available within ISTPO. Try changing a couple of options (e.g., set the left margin for statements to 13, LMARGS=13) and then return to the command executor using the "exit" command to create the option file "pl.opt":

```
ce: po pl.opt
New file
```

```
PO> LMARGS = 13
```

```
PO> e
```

We can now polish the file using either direct tool invocation or one of the ISTCE script files. First, let us do it the hard way, by invoking the tools starting with the scanner:

```
ce: lx src, lx.lst, lx.tkn, lx.cmt
[1STLX Normal Termination]
```

```
ce:
```

The scanner is invoked and scans the source code. A list file is created (lx.lst) which gives the token number for the first token in each statement. The resulting token stream and comment file are in lx.tkn and lx.cmt. Had any of the names been missing from the command (e.g., lx src,, lx.tkn, lx.cmt), the scanner would have prompted for it; this is true of all Toolpack/1 tools. The list file may be omitted by specifying the name "-". Any of these files can be listed using the "sh" command. Next comes the polishing action:

```
ce: pl lx.tkn, lx.cmt, lx.pol, pl.opt
[1STPL Normal Termination]
```

```
ce:
```

The polished output is placed in the file lx.pol and may be displayed using the "sh" command. Depending on your polish options, the main program should look like this:

```
ce: sh lx.pol
```

```
PROGRAM TEST
```

REAL A(10), MEAN

PRINT *, 'This is a test program.'
PRINT *, 'Please input 10 real numbers:
READ *, A
X = MEAN(A, 10)
PRINT *, 'The mean is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
CALL SQUARE(A, 10)
PRINT *, 'The squares of the numbers are:', A
X = SUM(A, 10)
PRINT *, 'The sum of the squares is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
STOP
END

SUBROUTINE SQUARE(A, N)
REAL A(N)
INTEGER I, N

Using the supplied ISTCE script files, the same operation could have been performed by the single command

CE: process trial
[ISTYP Normal Termination]
lx.pol
CE:

This approach has the additional advantage that if there is an existing token stream that is up to date, the scanning command is skipped. Parsing is performed as a side effect of polishing by this command file. There is also a command file called polish which replaces the file "src" with a polished version of itself. This can be used as

CE: polish trial
CE:

Both of these command files are found in the directory "/com". Listing the "/com" directory (using "Id /com") will show the command files available. The operation of the files can be watched by toggling the echo flag, e.g.,

CE: ec
Echo flag set ON
CE: polish trial
To understand the commands in the script file, you should read the ISTCE Users’ Guide.

Processing a Fortran File

The script file “process/*rq is capable of performing much more complex processing, for example, the declaration standardizing and precision conversion of the file src using ISTDS and ISTPT. The following is an example of precision transformation using “process” with the echo flag off:

```
   ce: process trial,pt
   [ISTPT Normal Termination]
   [ISTYP Normal Termination]
   [ISTDS Normal Termination]
   [ISTPL Normal Termination]
   pt.pol
   ce:
```

Note that the initial scanning and parsing stages were not required, as we have already done them during polishing (parsing is not actually required for polishing but is performed as a side effect). The resulting code can be found in “pt.pol”:

```
   ce: sh pt.pol
   PROGRAM TEST
   C .. Local Scalars ..
   DOUBLE PRECISION X
   C ..
   C .. Local Arrays ..
   DOUBLE PRECISION A(10)
   C ..
   C .. External Functions ..
   DOUBLE PRECISION MEAN, STAND, SUM
   EXTERNAL MEAN, STAND, SUM
   C ..
   C .. External Subroutines ..
   EXTERNAL SQUARE
   C ..
   C .. Executable Statements ..
   PRINT *, 'This is a test program.'
   PRINT *, 'Please input 10 real numbers:'
   READ *, A
   X = MEAN(A,10)
   PRINT *, 'The mean is:', X
   X = STAND(A,10)
   ok? n
   ce:
```

Examining Static Analysis Information

If we now look at the directory “/pu/trial,” we will find many intermediate files that the tools have left behind. These files contain intermediate representations that can be used by other tools to
perform further transformations or provide us with information:

c: wd /pu/trial
c: Id
/pu/trial
NAME                ACC TYPE      <- last modified ->
src                 Seq File 12:14:05 06 MAY 1986
pl.opt               Seq File 11:48:41 06 MAY 1986
lx.err               Seq File 11:50:17 06 MAY 1986
lx.pol               Seq File 12:00:23 06 MAY 1986
ds.pol               Seq File 12:07:02 06 MAY 1986
lx.tkn               Seq File 12:14:23 06 MAY 1986
lx.cmt               Seq File 12:14:26 06 MAY 1986
yp.tre               Seq File 12:15:17 06 MAY 1986
yp.sym               Seq File 12:15:20 06 MAY 1986
yp.cmi               Seq File 12:15:23 06 MAY 1986
pt.tkn               Seq File 12:16:35 06 MAY 1986
pt.cmt               Seq File 12:16:38 06 MAY 1986
pt.tre               Seq File 12:17:31 06 MAY 1986
pt.sym               Seq File 12:17:33 06 MAY 1986
pt.cmi               Seq File 12:17:36 06 MAY 1986
ds.tkn               Seq File 12:18:35 06 MAY 1986
ds.cmt               Seq File 12:18:39 06 MAY 1986
pt.pol               Seq File 12:19:36 06 MAY 1986
c:

The easiest way of looking at this information is using the documentation generation aid in its interactive mode. The following example session shows some of the information that can be derived; for full details, see the ISTAL Users' Guide:

c: al 0, 1
al: folding = yes
al: Verbose = yes
al: callgraph = yp.sym

The following callgraph shows the routine dependencies of those routines and entry points detailed within the specified symbol table files. Where an entry is followed by a number in brackets, the number refers to the line on which that entry's expansion has already been shown. If a name is followed by a question mark, this indicates that the routines symbol table was not provided.

1 TEST
2 MEAN
3 SQUARE
4 STAND
The following subsections show the routine dependencies of those routines and entry points detailed within the specified symbol table files.

TEST
CALLS:
    MEAN, SQUARE, STAND, SUM
NOT CALLED

MEAN
CALLS NOTHING:
    CALLED BY:
        TEST, STAND

SQUARE
CALLS NOTHING:
    CALLED BY:
        TEST

STAND
CALLS:
    MEAN, SQRT
    CALLED BY:
        TEST

SUM
CALLS NOTHING:
    CALLED BY:
        TEST

SQRT
    [Standard Intrinsic]
    CALLED BY:
        STAND

al: warnings = stand

The following table shows warnings derived from the symbol tables of the specified program units:

Warnings for program unit: STAND
    Implicitly typed Variable: I - INTEGER
    External procedure not in EXTERNAL: MEAN - REAL
    Intrinsic procedure not in INTRINSIC: SQRT - Generic

al: symbol = test
The following table shows the symbol usage for the specified program units:

Symbol table information for program unit: TEST

Variables:
- A - REAL (declared as an array)
  - Explicitly typed
  - In READ input list
  - Used as an actual argument
  - In an expression
- X - REAL
  - Assigned to on lhs of "=
  - In an expression

Procedures:
- MEAN - REAL
  - Explicitly typed
  - Called as a function
  - In an expression
- SQUARE - Routine
  - Called as a subroutine
- STAND - REAL
  - Called as a function
  - In an expression
- SUM - REAL
  - Called as a function
  - In an expression

al: quit
[ISTAL Normal Termination]
ce:

The file may be further analyzed using the script file "check." This script will perform extensive static analysis using ISTLX, ISTYP, ISTSA, and ISTPF. The file will be checked for conformance to the Fortran 77 standard as well as the PFORT 77 standard.

NOTE: The output from ISTPF exceeds the width of the page and has been modified in this example.

cce: check trial
[ISTLX Normal Termination]
[ISTYP Normal Termination]
[ISTSA - Toolpack Semantic Analyzer, Version 1.1
[Test processed]
[SQUARE processed]
[SUM processed]
[MEAN processed]
[STAND processed]
[Global processing completed]
[ISTSA Normal Termination]
[ISTPF - Toolpack/1 PFORT-77 Portability Verifier
Warning: Non-standard char 'h' in character const at st 3 in TEST
Warning: Non-standard char 'l' in character const at st 4 in TEST
Warning: Non-standard char 'h' in character const at st 7 in TEST
Note that if a string contains a number of lower-case characters, only the first one found is reported.

Dynamic Analysis of a Fortran File

So far, all the analysis done on our source code has been static. Toolpack/1 also contains dynamic analysis capabilities. The following command will instrument the original program, placing the result in the host file INS.FTN:

c: an lx.tkn, lx.cmt, #ins.ftn, sum, an.tkn, an.cmt, rep, rundata='run'
[ISTAN Normal Termination]
c: ex

The program INS.FTN can now be compiled, loaded, and run as described in the ISTAN Users' Guide. During execution, the instrumented program will create a host file called "run" which will contain the trace information:

$ compile ins.ftn
$ load ins.bin
$ run ins
This is a test program.
Please input 10 real numbers:
1 2 3 4 5 6 7 8 9 10
The mean is: 5.500000
The standard deviation is: 9.082951
The squares of the numbers are: 1.000000 4.000000 9.000000 ......
The sum of the squares is: 385.0000
The standard deviation is: 102.5207
On completion, the instrumented program reports the execution frequencies for the code segments. Now rerun the command executor ISTCE, move to the correct local directory, and re-enter an interactive ISTAL session to look at the information:

```bash
$ run ISTCE
cc: wd /pu/trial
cc: pl an.tkn, an.cmt, an.pol, pl opt
[ISTPL Normal Termination]
cc: al 0, 1
al: verbose = yes
al: folding = yes
al: annotated = an.pol
al: run = #run
al: summary = sum
al: listing

The following listing of the instrumented program has been annotated with the segment execution frequencies and assertion failure counts taken from the file:
#run

SEGMENT 1: 1

PROGRAM TEST

REAL A(10) , MEAN

PRINT *, 'This is a test program.'
PRINT *, 'Please input 10 real numbers:'
READ *, A
X = MEAN(A, 10)
PRINT *, 'The mean is:', X
X = STAND(A, 10)
PRINT *, 'The standard deviation is:', X
```
CALL SQUARE(A,10)
PRINT 'The squares of the numbers are:',A
X = SUM(A,10)
PRINT 'The sum of the squares is:',X
X = STAND(A,10)
PRINT 'The standard deviation is:',X
STOP

END

SEGMENT 2: 1
SUBROUTINE SQUARE(A,N)
REAL A(N)
INTEGER I,N

DO 100 I = 1,N
SEGMENT 3: 10
A(I) = A(I)*A(I)
100 CONTINUE
SEGMENT 4: 1
RETURN

END

SEGMENT 5: 1
REAL FUNCTION SUM(A,N)
REAL A(N)
INTEGER I,N

SUM = 0
DO 100 I = 1,N
SEGMENT 6: 10
SUM = SUM + A(I)
100 CONTINUE
SEGMENT 7: 1
RETURN

END

SEGMENT 8: 3
REAL FUNCTION MEAN(A,N)
REAL A(N)
INTEGER N,I

MEAN = 0
DO 100 I = 1,N
SEGMENT 9: 30
MEAN = MEAN + A(I)
100 CONTINUE
SEGMENT 10: 3
MEAN = MEAN/N
END

SEGMENT 11: 2
REAL FUNCTION STAND(A,N)
REAL A(N)
INTEGER N
REAL M,MEAN
M = MEAN(A,N)
STAND = 0
DO 100 I = 1,N
SEGMENT 12: 20
    STAND = STAND + (A(I)-M)**2
    CONTINUE
SEGMENT 13: 2
    STAND = SQRT(STAND)
SEGMENT 14: 0
    IF (STAND.EQ.0) RETURN
SEGMENT 15: 2
END

al: segment = ?*

The following table shows the execution frequencies for the various segments. The first count for each program unit is also the invocation frequency for that unit.

SEGMENT EXECUTION FREQUENCIES

<table>
<thead>
<tr>
<th>NAME</th>
<th>FIRST SEG</th>
<th>EXECUTION FREQUENCIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>( 1 ) :</td>
<td>1</td>
</tr>
<tr>
<td>SQUARE</td>
<td>( 2 ) :</td>
<td>1, 10, 1</td>
</tr>
<tr>
<td>SUM</td>
<td>( 5 ) :</td>
<td>1, 10, 1</td>
</tr>
<tr>
<td>MEAN</td>
<td>( 8 ) :</td>
<td>3, 30, 3</td>
</tr>
<tr>
<td>STAND</td>
<td>( 11 ) :</td>
<td>2, 20, 2, 0, 2</td>
</tr>
</tbody>
</table>

al: totals = ?*

The following table gives information derived from the static and dynamic statistics specified.

SUMMARY TOTALS

<table>
<thead>
<tr>
<th>PROGRAM UNIT</th>
<th>STATEMENTS</th>
<th>SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INVOCATION</td>
<td>TOTAL EXEC.</td>
</tr>
<tr>
<td></td>
<td>NUMBER</td>
<td>UTABLE EXECUTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOTAL NUMBER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>FREQUENCY</th>
<th>NUMBER</th>
<th>UTABLE EXECUTED</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>1</td>
<td>17</td>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>SQUARE</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>SUM</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>MEAN</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>STAND</td>
<td>2</td>
<td>12</td>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>

- TOTAL 8 55 40 100 15 93

al: static = sum

This table contains a count of the
**Statements in the specified program unit, split by statement type.**

**STATIC SUMMARY FOR PROGRAM UNIT: SUM**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERTIONS</td>
<td>0</td>
</tr>
<tr>
<td>COMMENTS</td>
<td>0</td>
</tr>
<tr>
<td>ERRORS</td>
<td>0</td>
</tr>
<tr>
<td>TOKENS</td>
<td>48</td>
</tr>
<tr>
<td>STATEMENTS</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statement</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>0</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>0</td>
</tr>
<tr>
<td>BLOCK DATA</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>0</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>0</td>
</tr>
<tr>
<td>CLOSE</td>
<td>0</td>
</tr>
<tr>
<td>COMMON</td>
<td>0</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>0</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>1</td>
</tr>
<tr>
<td>DATA</td>
<td>0</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>0</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>0</td>
</tr>
<tr>
<td>DO</td>
<td>1</td>
</tr>
<tr>
<td>ELSE IF</td>
<td>0</td>
</tr>
<tr>
<td>ELSE</td>
<td>0</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>0</td>
</tr>
<tr>
<td>END IF</td>
<td>0</td>
</tr>
<tr>
<td>END</td>
<td>1</td>
</tr>
<tr>
<td>ENTRY</td>
<td>0</td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>0</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>0</td>
</tr>
<tr>
<td>FORMAT</td>
<td>0</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>1</td>
</tr>
<tr>
<td>--CHARACTER</td>
<td>0</td>
</tr>
<tr>
<td>--COMPLEX</td>
<td>0</td>
</tr>
<tr>
<td>--DOUBLE PRECISION</td>
<td>0</td>
</tr>
<tr>
<td>--INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>--LOGICAL</td>
<td>0</td>
</tr>
<tr>
<td>--REAL</td>
<td>1</td>
</tr>
<tr>
<td>--UNALLOCATED</td>
<td>0</td>
</tr>
</tbody>
</table>

**al: dynamic = sum**

This table contains a count of the statements actually executed in the specified program unit, split by statement type.

**DYNAMIC SUMMARY FOR PROGRAM UNIT: SUM**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>0</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>0</td>
</tr>
</tbody>
</table>
### SEGMENTS NOT EXECUTED

<table>
<thead>
<tr>
<th>NAME</th>
<th>FIRST SEG</th>
<th>SEGMENTS NOT EXECUTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>( 1) :</td>
<td></td>
</tr>
<tr>
<td>SQUARE</td>
<td>( 2) :</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>( 5) :</td>
<td></td>
</tr>
<tr>
<td>MEAN</td>
<td>( 8) :</td>
<td></td>
</tr>
<tr>
<td>STAND</td>
<td>(11) :</td>
<td>14,</td>
</tr>
</tbody>
</table>

al: quit

[ISTAL Normal Termination]

Documentation Processing

A brief example document in the file "/pu/test/anl" demonstrates the use of ISTAL and ISTRF formatting commands. The document may be copied to the "trial" directory and used as an example. Processing may be performed using the provided script file "print" as shown below:

```
ce: cp /pu/test/anl, anl
ce: print trial
[ISTAL Normal Termination]
[ISTRF Normal Termination]
ce:
```

The resulting text file, "/pu/trial/txt," may be printed out on a line printer or displayed on the screen (using the "sh" command).
Example 2

This example shows the use of the structuring program ISTST. A small, unstructured test program will be created, scanned, parsed, and structured using the command executer.

First run ISTCE according to the instructions for your particular machine. Create and enter a new working directory, and then call the editor ISTED:

```
run ISTCE
ce: cd /pu/struct
ce: wd /pu/struct
ce: ed src
Empty initial file specified
```

Then use the append command to start inputing the new program, ending with a period on a new line:

```
0: app
  PROGRAM TEST
  READ *, I
  IF (I .NE. 1) GOTO 10
  X=3.0
  Y=10.0
  Z=X*Y
  GOTO 20
10  X=1.0
   Y=5.0
   Z=X/Y
20  CONTINUE
   PRINT *, Z
END
```

Refer to the previous example for the use of the browse (bro) command and substitute commands to check for and correct any typing errors. Write the file, and quit the editor:

```
0: wri
14 lines written to file src
0: qui
Fortran STOP
```

The first step in the processing of this file is to scan the program. The standard file names are used here for the intermediate files:

```
ce: lx src,lx.lst,lx.tkn,lx.cmt
[ISTLX Normal Termination]
Fortran STOP
```

Then the parser should be called:

```
ce: yp lx.tkn,lx.cmt,yp.tre,yp.sym,yp.cmi
[ISTYP Normal Termination]
Fortran STOP
```
As ISTST calls Toolpack/l access routines to convert the flattened tree to source code, the polish options editor should be used to create or change the options file. The example here shows a new options file being created:

```
c: pl.opt
 [New file]

PO> LMARGS=10

PO> e
Fortran STOP
```

All the preprocessing for the structuring tool has now been done, and ISTST may be called:

```
c: styp.tre,yp.sym,yp.cmi,lx.cmt,struct.pl.opt
 [ISTST Normal Termination]
Fortran STOP
```

The structured program can now be listed:

```
c: sh struct
PROGRAM TEST
     READ *,I
     IF (I.NE.1) THEN
         X = 1.0
         Y = 5.0
         Z = X/Y
     ELSE
         X = 3.0
         Y = 10.0
         Z = X*Y
     END IF
     PRINT *,Z
END
```

Example 3

This tutorial follows on from the previous example. The initial unstructured program will be altered using the editor, the Merge Fortran differencing tool will be used to list the changes. It is assumed that the output files from ISTLX created in the previous example exist and are up to date.

The file "src" from Example 2 is edited to change, insert, and delete lines. The editor is entered and the program retrieved and listed:

```
A change is made in line 3:

0: 3
  IF (I.NE.1) GOTO 10
0: s/.1/.2/p
     IF (I.NE.2) GOTO 10

A new line is inserted at line 9:

0: 9
  Y=5.0
0: ins
  !1X=X*X

Line 13 is deleted:

0: 13
  PRINT *,Z
0: d

The amended program can be listed using the browse command, written to file, and the edit session closed:

0: 1 bro
  PROGRAM TEST
  READ *,I
  IF (I.NE.1) GOTO 10
  X=3.0
  Y=10.0
  Z=X*Y
  GOTO 20
  10 X=1.0
     X=X*X
     Y=5.0
     Z=X/Y
  20 CONTINUE
As the comparison program has already been scanned by ISTLX, in Example 2, all that is necessary is to run the Merge Fortran program. Note that ISTMF assumes that the original program is the token stream version, and this is reflected in the comments changed from/to and inserted/deleted:

```
PROGRAM TEST

READ *, I
* $mf$ Changed From
* IF (I.NE.2) GO TO 10
* $mf$ Changed To
  IF (I.NE.1) GO TO 10
* $mf$ End Change
  Y = 10.0
  Z = X*Y
  GO TO 20

  10  X = 1.0
* $mf$ Deletion
* X=X*X
* $mf$ End Deletion
  Z = X/Y
  20  CONTINUE
* $mf$ Insertion
  PRINT *, Z
* $mf$ End Insertion
END
```

Conclusions

You have now seen several of the major Fortran facilities of Toolpack/1 at work, and you should feel more confident about using them yourself in the future.
Appendix B - Quick Reference to Tool Information

This appendix contains a quick reference guide to the tools provided in the second release of Toolpack/1. The information given in this appendix can also be obtained on-line using the simple help tool (ISTHP).

For each tool the following information is given:

1. Its name, which is always of the form ISTxx.
2. A brief description of the operation of the tool.
3. The prerequisites of the tool—a list of the tools that must be run first to ensure that the required intermediate files are available.
4. The arguments required by the tool. Arguments are prefixed with "<" to indicate that they are inputs, ">" to indicate that they are outputs, and "<>") indicate that the file is modified. Arguments enclosed in square brackets [] are optional. After each argument a bracketed expression shows the default file used if none is provided; a "*" indicates that there is no default and that the file is prompted for by the tool. For file name arguments the file type is also given (see Section 2). If a file is not required (e.g., there is no option file for ISTPL), then use the dummy name "--".

**ISTAL - Documentation Generation Aid**

This tool assists in the generation of documentation and may also be used on-line to generate symbol usage, call graph, common block usage, and other information. The prerequisites depend on the information required. This tool may be used interactively (input file = 0, output file = 1).

**Prerequisites:**

ISTLX, ISTYP, ISTAN

**Arguments:**

\[
\begin{align*}
<\text{input file} &: (*) \text{- any-} \\
>\text{output file} &: (*) \text{- .rf or - .tdm}
\end{align*}
\]

**ISTAN - Fortran 77 Instrumentor**

This tool instruments Fortran 77 program units to allow program flow tracing and the monitoring of expressions via assertions.

**Prerequisites:**

ISTLX

**Arguments:**

\[
\begin{align*}
<\text{Token stream file} &: (*) \text{- .tkn} \\
<\text{Comment stream file} &: (*) \text{- .cmt} \\
>\text{Instrumented output file} &: (*) \\
>\text{Statement summary file} &: (*) \text{ sum} \\
>\text{Annotated token stream file} &: (*) \text{ an.tkn} \\
>\text{Annotated comment stream file} &: (*) \text{ an.cmt} \\
>\text{Summary report file} &: (*) \text{ rep} \\
<\text{Option string} &: (*)
\end{align*}
\]
Toolpack/1 - Introductory Guide

ISTCB - Change Bar Processor

This tool adds change bars to a document.

Prerequisites:
None.
Arguments:
<input file 1 (*)
<input file 2 (*)
>output file (*)

ISTCD - Combine DO loops

One of the three DO loop unrolling tools (ISTUD, ISTCD, and ISTSB). This tool is concerned with combining consecutive DO loops.

Prerequisites:
ISTLX, [ISTUD, ISTSB]
Arguments:
<input token stream (*) --.tkn
<input comment stream (*) --.cmt
>output token stream (*) --.tkn
>output comment stream (*) --.cmt

ISTCE - Command Executor

This is a command executor capable of maintaining the PFS system and scheduling tools. The command executor provides a portable interface to the Toolpack/1 tool suite.

Prerequisites:
None
Arguments:
None

ISTCI - Command Interpreter

This is an experimental programming environment based on the Odin command interpreter.

Prerequisites:
None
Arguments:
None

ISTCN - Name Changer (token)

A token-based name changer, this tool may read from a source file or a token file and may write to a token file or polished source file.

Prerequisites:
[ISTLX]
Arguments:
either <Command file (*)
or <Input source file (*)
ISTCR - Name Changer (symbol)

This tool is a symbol table-based name changer.

Prerequisites:
ISTLX, ISTYP

Arguments:

<Input symbol table  (*)
>Output symbol table  (*)
<Command file  (*)

ISTDC - Data File Comparison Tool

This tool compares files of data.

Prerequisites:
None

Arguments:

<standard data file  (*) --any--
<comparison data file  (*) --any--
>output file  (*) --any--
[ <Options ]  (*)

ISTDS - Declaration Standardizer

This tool standardizes the declarative parts of Fortran 77 program units.

Prerequisites:
ISTLX, ISTYP

Arguments:

<input parse tree  (*) --.tre
<input symbol table  (*) --.sym
<input comment index  (*) --.cmi
<input comment stream  (*) --.cmt
>output token stream  (*) ds.tkn
>output comment stream  (*) ds.cmt
[ <Options ]  (*)

ISTDT - Monolithic tool

This tool is a token stream-to-source transformer which performs declaration standardization. It can be used as a postprocessor to ISTQT. ISTDT equates to ISTYP-ISTDS-ISTPL.
Prerequisites:
ISTLX
Arguments:
<input token stream> (*)
<input comment stream> (*)
<output source> (*)
DS option string (*)
<PL option file> (*)
[ <polish option overrides ]

ISTDX - Documentation Extractor
This tool extracts embedded documentation from a Fortran program.

Prerequisites:
None
Arguments:
<input file> (*)
<output file> (*)

ISTED - Fortran Aware Editor
This is a basic line based editor with some Fortran 77 awareness.

Prerequisites:
None
Arguments:
<>source file (*) - any -

ISTET - Tab Expander
This tool expands tabs, replacing them with the appropriate number of spaces.

Prerequisites:
None
Arguments:
<input file> (*)
<output file> (*)
<input format> (*)

ISTFD - Fortran Intelligent Differencer
This is a difference program that compares token streams.

Prerequisites:
ISTLX
Arguments:
>first token file (*) --.tkn
>first comment file (*) --.cmt
>second token file (*) --.tkn
>second comment file (*) --.cmt
<output file> (*) fd.lst
[ >options (*) ]
ISTFI - Find Include Files

This tool finds all the include files that a file needs.

Prerequisites:
None.
Arguments:

>input file          (*) -any-
<output file         (*) -any-
[ >option string ]   (pfs) pfs or hfs

ISTFL - File Length Calculator

This simple tool finds the length of a sequential file.

Prerequisites:
None
Arguments:

<file to be measured -any-

ISTFP - Fast Polish

This tool performs a very quick token stream-to-source code conversion. Comments are lost.

Prerequisites:
ISTLX
Arguments:

<token stream file
<comment stream file
>output source text

ISTFR - Real Number Format Changer

This tool converts the format of real, double precision, and complex constants to a standard form.

Prerequisites:
ISTLX
Arguments:

<Token stream          ()
<Comment stream        ()
>Token stream          ()
>Comment stream        ()

ISTGI - Make Generic Intrinsics

This tool converts intrinsic function references to their generic forms.

Prerequisites:
ISTLX, ISTYP
Arguments:

<symbol table          (*) . . . sym
>symbol table          (*) gi . sym
Toolpack/1 - Introductory Guide

ISTGP - Generalized Pattern Matcher

This pattern finder searches multiple files for the occurrence of a regular expression.

Prerequisites:
None

Arguments:
>pattern (*)
>file name or option (*) - any -

ISTHP - Tool Help System

This on-line help system contains information on the available tools based on this document.

Prerequisites:
None

Arguments:
None.

ISTIN - Include File Processor

This tool inserts the contents of include files into the input file, marking the bounds with source-imbedded directives.

Prerequisites:
None.

Arguments:
<Input source file (*)
>Output source file (*)
<Option string (*)

ISTJF - Symbolic Differentiator

This is a TIE-conforming version of the JAKEF symbolic differentiator. This tool currently operates only on Fortran 66.

Prerequisites:
None

Arguments:
>input source code (*) src
<output source code (*) src

ISTJS - Join Strings in Format Statements

This tool joins strings and does some conversions in FORMAT statements.

Prerequisites:
ISTLX

Arguments:
<Token stream (*)
<Comment stream (*)
>Token stream (*)
>Comment stream (*)
<Option stream (*)
ISTLA - Monolithic Semantic Analyzer

This static semantic analyzer performs lexical, syntactic, and semantic analysis on a source code file. It equates to ISTLX-ISTYP-ISTSA.

Prerequisites:
None
Arguments:

- <input source file> (*)
- >output comment stream (*)
- >output parse tree (*)
- >output symbol table (*)
- >output comment index (*)
- >output attribute file (*)

ISTLP - Monolithic Polish

This tool is a source-to-source polish utility. It equates to ISTLX-ISTPL.

Prerequisites:
None.
Arguments:

- <input source file> (*)
- >output source file (*)
- <PL option file> (*)
- [ <polish option overrides ]

ISTLS - Long Name Changer

This tool changes long Fortran names.

Prerequisites:
ISTLX
Arguments:

- <Token stream> (*)
- <Comment stream> (*)
- >Token stream (*)
- >Comment stream (*)
- <*>Name Conversion file(*)
- >Log file

ISTLX - Fortran 77 Scanner

This is a Fortran 77 scanner.

Prerequisites:
None
Arguments:

- <source file> (*) src
- >list file (*) lx.lst
- >token file (*) lx.tkn
- >comment file (*) lx.cmt
ISTLY - Monolithic Parser

This tool, a combined lexer and parser, may be used as a preprocessor to other tool sequences when the token stream is not required. It equates to ISTLX-ISTYP.

Prerequisites:
None
Arguments:

<input source file (*)>
>output comment stream (*)
>output parse tree (*)
>output symbol table (*)
>output comment index (*)

ISTME - Expression Manipulator

This tool manipulates expressions to reduce the depth of the intermediate value stack.

Prerequisites:
ISTLX, ISTYP
Arguments:

<input tree file (*)>
>output tree file (*)

ISTMF - Merge Fortran

This is a differencing program. It takes as input two Fortran programs, one in source form, the other in token stream form, and produces a merged output source. The differences are marked with comments for Insertion, Deletion, and Change.

Prerequisites:
ISTLX (Note: for one of the programs only)
Arguments:

<input source file (*) -any-
<input token stream (*) --.tkn
<input comment file (*) --.cmt
>output merged source (*)
<Options (*)
<POLISH option file (*) pl.opt
<Options (*)
[<Additional POLISH options]

ISTMP - Macro Processor

This is a general macro processor that can be used for expanding TIE macros, Toolpack Documentation Macros, or any macro file.

Prerequisites:
None
Arguments:

<input file (*) -any-
>output file (*) -any-
<operation flag (*)
ISTP2 - Source-to-Source Program PARAMETER Standardizer

This tool provides a monolithic equivalent to ISTLX followed by ISTPP.

Prerequisites:
None
Arguments:

\texttt{<input source file >output file}
[ \texttt{>input library file} ]

ISTPF - PFORT 77

This is a portability verifier for Fortran 77.

Prerequisites:
ISTLX, ISTYP, ISTSA
Arguments:

\texttt{<parse tree file (*) \textasciitilde \textbackslash tre}
\texttt{<symbol table file (*) \textasciitilde \textbackslash sym}
\texttt{<attribute file (*) \textasciitilde \textbackslash att}
[ \texttt{<library attribute files (*) \textasciitilde \textbackslash att} ]

ISTPL - Polish 77

This is a programmable Fortran 77 pretty printer, polisher, or unscanner.

Prerequisites:
ISTLX
Arguments:

\texttt{<token file \textasciitilde \textbackslash tkn}
\texttt{<comment file \textasciitilde \textbackslash cmt}
\texttt{>polished output file \textasciitilde \textbackslash pol}
[ \texttt{<option file \textasciitilde \textbackslash pl.opt}]
[ \texttt<option overrides}]

ISTPO - Polish Option File Editor

This is a menu-driven editor for ISTPL option files.

Prerequisites:
None
Arguments:

\texttt{<>option file name \textasciitilde \textbackslash pl.opt}

ISTPP - Program Parameter Standardizer

This tool ensures consistency of PARAMETER statements across program units.

Prerequisites:
ISTLX
Arguments:

\texttt{<input token stream (*)}
\texttt{<input comment stream (*)}
\texttt{>output token stream (*)}
ISTPT - Transform Precision

This tool converts the precision of a Fortran 77 program unit.

Prerequisites:
ISTLX, ISTYP

Arguments:
<parse tree> (*) tre
<symbol table> (*) sym
<comment index> (*) cmi
<comment file> (*) cmt
>token stream> (*) pt.tkn
>comment file> (*) pt.cmt
[ options ]

ISTQD - Monolithic Declaration Standardizer

This tool is a source-to-source declaration standardization operation. It equates to ISTLX-ISTYP-ISTDS-ISTPL.

Prerequisites:
None

Arguments:
<input source file> (*)
>output source file> (*)
DS option string> (*)
<PL option file> (*)
[ <polish option overrides ]

ISTQP - Monolithic Precision Convertor

This is a source-to-source arithmetic precision transformation operation. It equates to ISTLX-ISTYP-ISTPT-ISTPL.

Prerequisites:
None

Arguments:
<input source file> (*)
>output source file> (*)
PT option string> (*)
<PL option file> (*)
[ <polish option overrides ]

ISTQT - Monolithic Precision Convertor

This is a source-to-token stream transformer which performs arithmetic precision transformation. It can be used as a preprocessor to ISTDT. It equates to ISTLX-ISTYP-ISTPT.

Prerequisites:
None
Arguments:

- `<input source file (*)`
- `>output token stream (*)`
- `>output comment stream (*)`
- `PT option string (*)`

**ISTRF - Text Formatter**

This is a general text formatter.

**Prerequisites:**

(ISTMP if TDMs used)

**Arguments:**

- `<input file (*) - any`
- `>output file (*) txt`
- `<macro file (*)`

**ISTSA - Static Analyzer**

This is a static semantic analyzer.

**Prerequisites:**

ISTLX, ISTYP

**Arguments:**

- `<parse tree file (*) - tre`
- `<symbol table file (*) - sym`
- `>parse tree file (*) sa.tre`
- `>symbol table file (*) sa.sym`
- `<attribute file (*) sa.att`

**ISTSB - Substitute Expressions**

This is one of the three DO loop unrolling tools (ISTUD, ISTCD, and ISTSB). This tool is concerned with recombining and substituting expressions.

**Prerequisites:**

ISTLX, [ISTUD, ISTCD]

**Arguments:**

- `<input token stream (*) - .tkn`
- `<input comment stream (*) - .cmt`
- `>output token stream (*) - .tkn`
- `>output comment stream (*) - .cmt`

**ISTSP - File Splitter**

This is a TIE-conforming version of the Toolpack distribution SPLIT tool.

**Prerequisites:**

None

**Arguments:**

- `<file to be split - any`
ISTST - Fortran 77 Structuring Tool

This tool takes as input a Fortran program, analyzes the flow of control, and rebuilds the program in a structured form.

Prerequisites:
ISTLX, ISTYP

Arguments:
- <Input parse tree> (*) - tre
- <Input symbol table> (*) - sym
- <Input comment index> (*) - cmi
- <Input comment file> (*) - cmt
- <Output source file> (*)
- >POLISH option file (*) pl.opt
[>Additional POLISH options]

ISTSV - File Save and Restore Utility

This is a file archiver/restorer.

Prerequisites:
None

Arguments:
- >option (*)
- <>storage file (*) arc
- <output file (stdout)
[>files to save] (*) - any -

ISTTD - Text Differencer

This is a standard text-differencing tool.

Prerequisites:
None

Arguments:
- <first file> (*) - any -
- <second file> (*) - any -
[<output file (stdout)]
[options]

ISTUD - Unroll DO loops

This is one of the three DO loop unrolling tools (ISTUD, ISTCD, and ISTSB). This tool is concerned with unrolling the DO loops.

Prerequisites:
ISTLX

Arguments:
- <input token stream> (*) - tkn
- <input comment stream> (*) - cmt
- >output token stream (*) - tkn
- >output comment stream (*) - cmt
- unrolling depth (*)
Toolpack/1 - Introductory Guide

ISTUN - Undo include file inclusion

This tool reverses the effect of ISTIN. It takes out the include files, which are delimited by SEDs, and replaces them with INCLUDE statements.

Prerequisites:
ISTIN
Arguments:
<input file> (*)
<output file> (*)

ISTVA - Attribute Viewer

This is an attribute table viewer.

Prerequisites:
ISTLX, ISTYP, ISTSA
Arguments:
<input symbol table> (*)
<input attribute file> (*)
<output listing file> (*)
<header text> (*)

ISTVC - Version Controller

This is a version controller that can maintain a number of separate versions of a file in an ISTVC archive file.

Prerequisites:
None
Arguments:
<options> (*)
<version file> (*) ver
[ <optional file name> (*) ]

ISTVS - View Symbol table

This tool produces a brief report of the contents of the symbol table for a file.

Prerequisites:
ISTLX, ISTYP
Arguments:
<input symbol table> (*) --sym
<output listing file> (*) vs.lst
<header text> (*)

ISTVT - View Parse Tree

This tool provides a means of viewing and printing out parse trees.

Prerequisites:
ISTLX, ISTYP
Arguments:
<input parse tree> (*) --tre
<input symbol table> (*) --sym
ISTVW - View Warnings

This tool provides a capability similar to ISTVS, but only warnings are displayed. The warnings are the same as those reported by ISTAL.

Prerequisites:
ISTLX, ISTYP

Arguments:
<input parse tree (*)
<input symbol table (*)

ISTX1 - Experimental Expert System

This is a simple experimental expert system.

Prerequisites:
None
Arguments:
None

ISTX2 - Experimental Expert System

This is a simple experimental expert system.

Prerequisites:
None
Arguments:
None

ISTX3 - Experimental Expert System

This is a simple experimental expert system.

Prerequisites:
None
Arguments:
<Rule file (*)
<Question file (*)

ISTYF - Flatten ISTYP Parse Tree

This tool converts an ISTYP format parse tree to a token stream.

Prerequisites:
ISTLX, ISTYP

Arguments:
<parse tree (*) --tre
<symbol table (*) --sym
<comment index (*) --cmi
<comment stream (*) --cmt
>token stream (*) yf.tkn
>comment file (*) yf.cmt
ISTYP - Fortran 77 Parser

This is a Fortran 77 parser.

Prerequisites:
ISTLX

Arguments:

```
< input token stream      (*) --.tkn
< input comment stream   (*) --.cmt
> output parse tree      (*) --.tre
> output symbol table    (*) --.sym
> output comment index   (*) --.cmi
```

File Types

The following list contains details of the predefined file names for intermediate file types. These names should be used if the ISTCE standard operating procedures are being followed.

Several file types are given that start with "--." This is the generic form of the file. Normally a tool that requires a file type as input does not care which tool created that file; thus "--.tkn" could be "lx.tkn," "yf.tkn," "pt.tkn," or any other "*.tkn" file.

- any- An indication that any sequential file is legal.
mac Fortran 77 source code with include files and ZDLIB macros suitable for expansion using the "ti" switch on ISTMP.
src Legal compilable Fortran 77 source code.
pol Polished legal compilable Fortran 77 source code. This may be used as type src.
txt Formatted or unformatted text.
doc Input for the ISTRF tool.
tdm Text containing embedded ISTRF commands and Toolpack Documentation Macros (TDMs) of the form expanded by ISTMP.
anl Text containing ISTAL commands.
arc An ISTSV format archive file.
ver An ISTVC format version control archive file.
--.err The generic form of an error report file. These may normally be treated as txt files.
--.lst The generic form of a list file. These may normally be treated as txt files.
--.ref The generic form of a file containing a list of names of program units.
--.opt An option file.
--.tkn The generic form of a token stream.
--.cmt The generic form of a comment stream.
--.tre The generic form of an ISTYP format parse tree.
--.sym The generic form of an ISTYP format symbol table.
--.cmi The generic form of an ISTYP format comment index.
--.att An attribute file of the form generated by ISTSA.
This appendix contains a quick reference guide to the regular expressions used by tools. These regular expressions control pattern matching in the editor (ISTED), command executor (ISTCE), documentation generation aid (ISTAL), pattern matcher (ISTGP), and other tools and are defined in the string supplementary library.

1. An ordinary character (not defined below) matches itself.
2. A ? matches any single character, digit, or symbol.
3. A % at the beginning of a regular expression matches the empty string at the beginning of a line.
4. A $ at the end of a regular expression matches the null character at the end of a line.
5. A simple regular expression followed by a * matches zero or more occurrences of that regular expression (closure).
6. A simple regular expression followed by a + matches one or more occurrences of that regular expression (anchored closure).
7. A string of characters enclosed in square brackets [ ] matches any character in the string unless the first character is a ~ when the regular expression matches any character NOT in the string (other than newline). The string of characters may be abbreviated to a character range of the form a-z, 0-9, P-Y, etc.
8. The characters < and > open and close tag fields and are not part of the matching process. It is possible to tag up to 9-character fields in the regular expression that can be recalled in any order replacement text. Tag fields are opened using the symbol < and closed using the symbol >. Tag fields may be nested but may not overlap. The fields are numbered 1 to 9 in the order in which they are opened.
9. Any character preceded by an escape (@) character including the character @ itself matches the character without the @ sign even if that character normally has a special meaning.
10. The two-character symbol @n matches the newline character. The two-character symbol @t matches the tab character (ctrl/l).
11. A : matches a transition between alpha-numeric characters and non-alpha-numeric characters or vice versa.
12. A concatenation of regular expressions is itself a regular expression.

NOTES:

A simple regular expression is any one of 1, 2, 7, 9, or 10 mentioned above.

Characters specified within a class (i.e., using [ ]) are always case sensitive.

WARNING:

The use of @n within a regular expression will cause any patterns shown in Appendix D to fail to match; i.e., matches can be performed only on a single line.
## Appendix D - Toolpack/1 Release 2.1 Documentation

<table>
<thead>
<tr>
<th>Title</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Function Supplementary Guide</td>
<td>NAG Publication NP1283</td>
</tr>
<tr>
<td>Experimental Expert System Guide</td>
<td>NAG Publication NP1294</td>
</tr>
<tr>
<td>Flow of Control Using TIE</td>
<td>NAG Publication NP1275</td>
</tr>
<tr>
<td>General Tools User Guide</td>
<td>NAG Publication NP1292</td>
</tr>
<tr>
<td>ISTAL User Guide</td>
<td>NAG Publication NP1303</td>
</tr>
<tr>
<td>ISTAN User Guide</td>
<td>NAG Publication NP1304</td>
</tr>
<tr>
<td>ISTCE Users' Guide</td>
<td>NAG Publication NP1276</td>
</tr>
<tr>
<td>ISTCI User Guide</td>
<td>NAG Publication NP1297</td>
</tr>
<tr>
<td>ISTCN/CR User Guide</td>
<td>NAG Publication NP1310</td>
</tr>
<tr>
<td>ISTDC User Guide</td>
<td>NAG Publication NP1305</td>
</tr>
<tr>
<td>ISTDS User Guide</td>
<td>NAG Publication NP1298</td>
</tr>
<tr>
<td>ISTED Users' Guide</td>
<td>NAG Publication NP1274</td>
</tr>
<tr>
<td>ISTFD User Guide</td>
<td>NAG Publication NP1290</td>
</tr>
<tr>
<td>ISTGI User Guide</td>
<td>NAG Publication NP1302</td>
</tr>
<tr>
<td>ISTIF User Guide</td>
<td>NAG Publication NP1307</td>
</tr>
<tr>
<td>ISTLS User Guide</td>
<td>NAG Publication NP1321</td>
</tr>
<tr>
<td>ISTLX User Guide</td>
<td>NAG Publication NP1289</td>
</tr>
<tr>
<td>ISTMF User Guide</td>
<td>NAG Publication NP1316</td>
</tr>
<tr>
<td>ISTMP User Guide</td>
<td>NAG Publication NP1317</td>
</tr>
<tr>
<td>ISTPF User Guide</td>
<td>NAG Publication NP1312</td>
</tr>
<tr>
<td>ISTPL/ISTPO User Guide</td>
<td>NAG Publication NP1296</td>
</tr>
<tr>
<td>ISTPT Users' Guide</td>
<td>NAG Publication NP1287</td>
</tr>
<tr>
<td>ISTRF User Guide</td>
<td>NAG Publication NP1318</td>
</tr>
<tr>
<td>ISTSA User Guide</td>
<td>NAG Publication NP1311</td>
</tr>
<tr>
<td>ISTST User Guide</td>
<td>NAG Publication NP1315</td>
</tr>
<tr>
<td>ISTSV User Guide</td>
<td>NAG Publication NP1293</td>
</tr>
<tr>
<td>ISTTD User Guide</td>
<td>NAG Publication NP1291</td>
</tr>
<tr>
<td>ISTUD/CD/SB User Guide</td>
<td>NAG Publication NP1319</td>
</tr>
<tr>
<td>ISTVA User Guide</td>
<td>NAG Publication NP1320</td>
</tr>
<tr>
<td>ISTVC User Guide</td>
<td>NAG Publication NP1288</td>
</tr>
<tr>
<td>ISTVS User Guide</td>
<td>NAG Publication NP1299</td>
</tr>
<tr>
<td>ISTVT User Guide</td>
<td>NAG Publication NP1309</td>
</tr>
<tr>
<td>ISTVF User Guide</td>
<td>NAG Publication NP1301</td>
</tr>
<tr>
<td>ISTYP User Guide</td>
<td>NAG Publication NP1300</td>
</tr>
<tr>
<td>Monolithic Tools Guide</td>
<td>NAG Publication NP1314</td>
</tr>
<tr>
<td>Quick Reference Guide</td>
<td>NAG Publication NP1284</td>
</tr>
<tr>
<td>Screen/Window Library Guide</td>
<td>NAG Publication NP1306</td>
</tr>
<tr>
<td>String Supplementary Library Guide</td>
<td>NAG Publication NP1282</td>
</tr>
</tbody>
</table>
Internal:

J. M. Beumer (2)
W. R. Cowell (80)
H. G. Kaper
A. B. Kriściunas
G. W. Pieper (5)
E. P. Steinberg

ANL Patent Dept.
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (163)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
   J. L. Bona, Pennsylvania State University
   T. L. Brown, U. of Illinois, Urbana
   P. Concus, LBL
   S. Gerhart, MCC, Austin, Texas
   G. H. Golub, Stanford U.
   W. C. Lynch, Xerox Corp., Palo Alto
   J. A. Nohel, U. of Wisconsin, Madison
   D. Austin, ER-DOE
   G. Michael, LLL