USERS' GUIDE TO TOOLPACK/1 (RELEASE 2)
IN A UNIX ENVIRONMENT

Wayne R. Cowell and Burton S. Garbow

Mathematics and Computer Science Division

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.
Contents

Abstract.................................................................................................................................1

1. Summary ..........................................................................................................................1

2. Operation ..........................................................................................................................3

3. Fortran Analysis Facilities ..............................................................................................3

   3.1 Static Analysis ..............................................................................................................4

      3.1.1 getlst ..................................................................................................................4
      3.1.2 syn ......................................................................................................................5
      3.1.3 sem ......................................................................................................................6
      3.1.4 pfort ....................................................................................................................7
      3.1.5 statdoc ...............................................................................................................8

   3.2 Dynamic Analysis .......................................................................................................9

      3.2.1 inst ....................................................................................................................9
      3.2.2 rundoc ..............................................................................................................10

4. Fortran Transformation Facilities .....................................................................................11

   4.1 Fortran Formatting .....................................................................................................12

      4.1.1 polx ..................................................................................................................12
      4.1.2 pol ...................................................................................................................14

   4.2 Declaration Standardization .......................................................................................14

      4.2.1 decs ................................................................................................................16

   4.3 Precision Transformation .........................................................................................17

      4.3.1 apt ..................................................................................................................18
      4.3.2 dapt ................................................................................................................18
Abstract

This is a guide to the use of a collection of Unix shell scripts that extend the Fortran analyzing and transforming capabilities of Unix by invoking a set of tools from Toolpack/1 (Release 2). It is a substantial revision and update of Argonne report ANL/MCS-TM-77, which served as a Unix users' guide to the first release of Toolpack/1.

1. Summary

Toolpack/1 is an integrated suite of software tools that support the development of Fortran programs. The language accepted by the tools is "Toolpack/1 Target Fortran 77", which is standard Fortran 77 extended by the addition of certain features (see [3]). The tools are written in Fortran and operate on any host system on which the portability base TIE (Tool Interface to the Environment) has been installed. Under Berkeley Unix 4.2, TIE is implemented as a library of C routines. This guide describes a user interface for the Unix installation, consisting of a set of shell scripts.

The scripts are summarized below. For each script, the number of the section containing more detailed information is shown.

Fortran Analysis Facilities

getlst Produce a listing showing statement and token numbers. Report lexical scanning errors and warnings. (3.1.1)

syn Report errors and warnings detected by lexical scanning, parsing, and examining a set of symbol attributes. (3.1.2)

sem Report errors and warnings detected by lexical scanning, parsing, and examining an extended set of symbol attributes. (3.1.3)

pfort Report errors and warnings detected by lexical scanning, parsing, examining an extended set of symbol attributes, checking for unsafe references, and checking conformance to a portable subset of Fortran. (3.1.4)

*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.
statdoc  Place information derived from static analysis in a user-supplied report template to assist in documenting a program. Examples of information the user can request are COMMON block usage, symbol attributes, and a graph of subprogram calls. (3.1.5)

inst    Instrument a Fortran program so that the instrumented program, when executed, produces information about program execution. (3.2.1)

rundoc  Execute a program instrumented by inst, and place information derived from dynamic analysis in a user-supplied report template to assist in documenting the program. For example, the user can determine the frequency of execution of program segments. (3.2.2)

Fortran Transformation Facilities

polx     Create interactively a new, or modify an existing, Polish option file using a menu-driven editor. (4.1.1)

pol      Format a Fortran program under control of user-supplied options given in a Polish option file. (4.1.2)

decs     Rebuild the declarative part of a Fortran program. (4.2.1)

apt      Transform a single-precision version of a Fortran program to double precision, or vice versa. (4.3.1)

dapt     Convert precision and rebuild the declarations in a Fortran program, combining the functions of apt and decs. (4.3.2)

cname    Change the names in a Fortran program that satisfy conditions derived from information in the lexical token stream and/or symbol table. (4.4.1)

lname    Transform a Fortran program containing long names to a program with standard names. (4.4.2)

stf      Rebuild the flow of control in a Fortran program to a standardized form. (4.5.1)

ucs      Transform code comprising nests of DO-loops matching certain paradigms so that the transformed code executes more efficiently on vector machines. (4.6.1)

Miscellaneous Facilities

fdiff    Compare two Fortran programs at the lexical token level. (5.1.1)
dac Compare two data files, neglecting certain formatting differences and those numerical differences less than a given tolerance. (5.1.2)

vcon Create, edit, and retrieve versions of a file. (5.2.1)

discard Remove unneeded files created by other scripts. (5.3.1)

scripts Print a summary of the entire set of scripts. (5.4.1)

2. Operation

Users may obtain from the Toolpack/1 installer the name of the directory containing the scripts and can then either create aliases or include the directory name in their path environment variable.

Users interested in the philosophy and design of Toolpack/1 should refer to [5]. Documents describing the individual tools are available from the Toolpack/1 installer. However, the Unix user is insulated by the scripts from a need for detailed knowledge of the behavior of these underlying tools. Based on the user's command line, the scripts invoke those tools required to achieve the end result and manage the intermediate files used to pass information from one tool to another. With the exception of discard and scripts (which have no arguments), the command syntax and other helpful information are displayed when the name of a script with no arguments is typed.

This guide is intended to be self-contained with two exceptions: First, users should consult [2] when building a so-called "Polish option file" of formatting options that govern the reconstruction of Fortran text when a program is transformed, for example, when rebuilding its declarations and/or changing its precision (scripts decs, apt, and dapt). Second, users who intend to invoke script ucs should first read [6] and [4].

3. Fortran Analysis Facilities

Static analysis tools report information derived from examining some representation of a program without executing it. Appropriate sequences of static analysis tools may be used to detect both syntactic errors (e.g., misspellings of Fortran key words) and semantic errors (e.g., inconsistencies in type declarations between actual and dummy variables). In contrast, compilers detect many errors by the static analysis they perform but do not, in general, detect errors that result from improper connections between program units.

Representations of the program include the token/comment stream (generated by the lexer tool ISTLX), the parse tree/symbol table (generated by the parser tool ISTYP), and the parse tree/symbol table with enhanced symbol attributes (generated by the semantic analyzer tool ISTSA). In the course of constructing representations of the program these tools carry out various checks that enable them to issue warnings and error reports.

The tool ISTPF, the so-called PFORT-77 verifier, constructs nothing additional but examines the information in the extended parse tree/symbol table. It detects certain unsafe references (e.g., duplicate actual arguments for which at least one of the corresponding dummy arguments may be updated by the called program), and Fortran constructions that are likely to be non-portable (e.g., COMPLEX or DOUBLE PRECISION entities following INTEGER or REAL entities in a COMMON block).
Dynamic analysis tools "instrument" a program by inserting additional Fortran. When the instrumented program is loaded and executed, it generates files that contain run-time information such as the frequency of execution of different segments of the program. The tool ISTAN instruments a program, while ISTAL enables the user to insert information derived from either static or dynamic analysis into reports.

3.1. Static Analysis

Tools that report warnings and errors based on an analysis of a program represented as a token/comment stream or as a parse tree/symbol table are invoked from the scripts getlst, syn, sem, and pfort. Errors flagged by these tools are summarized in Appendix A.

The analysis initiated by the above scripts becomes progressively deeper as the scripts are invoked in the order shown. With the exception of scanning errors, a script does not proceed with deeper analysis when errors from shallower analysis are reported. For example, if a Fortran program contains semantic errors that would be reported by syn, these errors are also reported when sem is invoked, and sem then terminates.

The script statdoc controls tools that allow a user to extract information about the use of symbols (variable names, subprogram names, labels, etc.) from the symbol table and to insert this information into a report.

3.1.1. getlst

Produce a listing showing statement and token numbers. Report lexical scanning errors and warnings.

Invocation:

```
getlst  Fortran_source
```

where *Fortran_source* is the name of the file containing the program. An error/warning report is sent to standard output and may be redirected to a file.

Example: getlst operating on the program

```
program progr
  do 10 i = 1,5
    write(6,*) 'The number ' i
    i = i + 2
  10 continue
end
```

reports the misspelling of the keyword *continue*, as follows:

In the following listing, the first number is the statement number and the second number is the number of the first token in the statement. Statements are numbered within a program unit, tokens within the file. Scanning errors are indicated. When a scanning error occurs, the scanner resets and continues.
3.1.2. syn

Report errors and warnings detected by lexical scanning, parsing, and examining a set of symbol attributes.

Invocation:

```
syn Fortran_source
```

where *Fortran_source* is the name of the file containing the program. An error/warning report is sent to standard output and may be redirected to a file. *syn* reports warnings and errors generated in the course of building the parse tree/symbol table and in making certain checks on information in the symbol table. If the program contains lexical scanning errors that would be reported by *getlst*, *syn* first reports these errors (no listing is generated) and then attempts to continue its deeper analysis with the corrupted "lower-level" information (i.e., the token/comment stream). If any errors are reported, the user is advised to obtain a listing using *getlst* so that the messages from *syn* can be correlated with the program.

Example: *syn* operating on the above program with the keyword misspelling fixed, i.e.,

```
program progr
  do 10 i = 1,5
    write(6,*) 'The number' i
    i = i + 2
  10 continue
end
```

gives the messages

**Error:** Syntax Error at statement 3 in PROGR
  detected at 'The number' @I (token number 19)

**Error:** Assignment to DO variable - I at statement 4 in PROGR
  detected at <end-of-statement> (token number 26)

The errors may be located by referring to the listing from *getlst* (note the explanation of the numbers in the above example):
3.1.3. sem

Report errors and warnings detected by lexical scanning, parsing, and examining an extended set of symbol attributes.

Invocation:

```
sem Fortran_source
```

where `Fortran_source` is the name of the file containing the program. An error/warning report is sent to standard output and may be redirected to a file. If the program contains errors that would be reported by `getlst` or `syn`, `sem` reports these errors and terminates. If there are no such errors, `sem` reports errors and warnings generated in the course of building an extended symbol table and in making certain checks on information in the symbol table. If any errors occur, the user is advised to obtain a listing using `getlst` so that the messages from `sem` can be correlated with the program.

Example: `sem` operating on the program

```
program fill
real a(100,100)
external matfil
read(5,*) rc
call matfil(rc,rc,a)
stop
end
subroutine matfil(size,col,mat)
integer size, col
real mat(100,100),filler
external filler
do 10 j = 1,2
   do 20 i = 1,size
      mat(i,col) = filler(j,i)
   20 continue
   col = col + 1
10 continue
return
end
```

gives the message

Error: Wrong type of argument to MATFIL in FILL
The program will pass sem if the variable rc in the main program fill is typed as integer.

3.1.4. pfort

Report errors and warnings detected by lexical scanning, parsing, examining an extended set of symbol attributes, checking for unsafe references, and checking conformance to a portable subset of Fortran.

Invocation:

\[ \text{pfort } \text{Fortran_source} \]

where Fortran_source is the name of the file containing the program. An error/warning report is sent to standard output and may be redirected to a file. If the program contains errors that would be reported by getist, syn, or sem, pfort gives the same report as syn or sem (depending on the level at which the error is detected) and terminates. If there are no such errors, pfort reports errors and warnings generated by examining the attributes in the extended symbol table for correct inter-program-unit communication, unsafe references (illustrated in the example below), and conformance to a portable subset of the Fortran 77 standard. If any errors occur, the user is advised to obtain a listing using getist so that the messages from pfort can be correlated with the program.

Example: pfort operating on the program

```fortran
program fill
  real a(100,100)
  integer rc
  external matfil
  read(5,*) rc
  call matfil(rc,rc,a)
  stop
end

subroutine matfil(size,col,mat)
  integer size, col
  real mat(100,100),filler
  external filler
  do 10 j = 1,2
      do 20 i = 1,size
          mat(i,col) = filler(j,i)
  20 continue
      col = col + 1
  10 continue
  return
end
```

gives the error/warning report
Warning: Missing function FILLER

******************************************************************************************
* *
* Error(s) have been detected by PFORT-77  *
* *
******************************************************************************************

Error: Type 2 unsafe reference to MATFIL from FILL at statement 6
Actual arguments 1 and 2 are duplicated
and at least one of them is updated

The error message reported by pfort will go away if an integer variable in the main program is defined, say irc, set equal to rc after the read statement, and substituted for one of the occurrences of rc in the call to matfil.

3.1.5. statdoc

Place information derived from the symbol table in a user-supplied report template.

Invocation:

    statdoc Fortran_source_file input_file output_file

input_file is a user-written template, containing troff requests, of a report to be written as output_file containing information derived from the symbol table of the program in Fortran_source_file. At those places in input_file where information is to be inserted, the user writes a special request of the form "\al{} X" where "X" specifies the information to be inserted. Following are these special requests, showing the nature of the information that is inserted:

\al{} _CALL An indented display showing the subroutines and functions called by each program unit.
\al{} _X A cross-reference showing the program units called by each program unit.
\al{} _F A cross-reference showing the program units that call and are called by each program unit.
\al{} _CO A report of COMMON block usage by program unit.
\al{} _SY A list of symbols and their attributes by program unit.
\al{} _W Error messages about various semantic errors (e.g., a variable referenced but not set) and warnings that point to possible problems.

After executing statdoc, output_file may be processed like any text file containing troff requests to produce the required report. No external troff macros are inserted, nor is preprocessing of output_file (e.g., by eqn) assumed by statdoc. The user is, of course, free to assume that such facilities can be used to process output_file and to construct input_file accordingly.
3.2. Dynamic Analysis

Script \texttt{inst} controls the operation of the tools that instrument a program and produce a reference report, while script \texttt{rundoc} runs an instrumented program that has been compiled and loaded, and controls the tool that enables information derived from the run to be inserted into a report.

Reports from \texttt{inst} and \texttt{rundoc} use the notion of \textit{segments} in referring to parts of the program. A \textit{segment} is a "straight-line" section of code; a new segment begins after a branch or conditional statement, or at the target of a possible branch or conditional statement.

The Fortran source instrumented by invoking \texttt{inst} may contain a complete program or only some program units. In the latter case, the instrumented units may be compiled and loaded with the remaining uninstrumented units. However, if the instrumented program units do not include the main program, the normal termination point (e.g., the \texttt{STOP} statement) in the uninstrumented main program must be replaced by

\begin{verbatim}
CALL RZZ4QX
\end{verbatim}

so that information generated during execution of the instrumented program may be written to external files that are read when \texttt{rundoc} is invoked.

The Fortran source instrumented by invoking \texttt{inst} may contain embedded \textit{assertions}, that is, comments of the form

\begin{verbatim}
*$\texttt{ASS}$(E)
\end{verbatim}

where \texttt{E} is a Fortran logical expression. For example, if the source contains the comment

\begin{verbatim}
*$\texttt{ASS}$(IVAR .EQ. JVAR .OR. IVAR .LE. 0)
\end{verbatim}

instrumentation is inserted to check whether the logical expression is true when the code where it appears is executed. Whenever the expression (the assertion) is false, the count of violations of this particular assertion is incremented by 1.

The Fortran source instrumented by invoking \texttt{inst} may also contain special comments called \textit{SED switches}, namely,

\begin{verbatim}
*$\texttt{DX}$=ON
\end{verbatim}

and

\begin{verbatim}
*$\texttt{DX}$=OFF
\end{verbatim}

that demarcate sections of code from which comments may be extracted for the report generated by \texttt{rundoc}.

3.2.1. \texttt{inst}

Instrument a Fortran program.

Invocation:

\begin{verbatim}
inst Fortran\_source instrumented\_Fortran\_source
\end{verbatim}
**Fortran_source** is the name of a file containing one or more Fortran program units to be instrumented (see Section 3.2). The instrumented source, containing subroutines and calls to collect various kinds of run-time data, is placed in instrumented_Fortran_source which should be given a "f" extension. Files _.ANSTATICSUM and _.ANANNOLIST are created in the working directory for use by script rundoc.

A report consisting of two parts is sent to standard output and may be redirected to a file. The first part is a listing of Fortran_source annotated with identification numbers for segments and assertions. Users can refer to this listing when they interpret the output from rundoc. The second part of the report is a static count of Fortran key words and other program elements by program unit.

instrumented_Fortran_source should be compiled and loaded with any uninstrumented parts of the program to create an executable module for use when rundoc is invoked.

### 3.2.2. rundoc

Execute a program instrumented by inst and place information derived from the run in a user-supplied report template.

Invocation:

```
rundoc execute_command_line input_file output_file
```

execute_command_line is a Unix command line, written without blanks, for the execution of a program part or all of which has been instrumented with script inst. The instrumented program has identically the same requirements for input as the uninstrumented program and produces the same output, in addition to the reports described below. input_file is a user-written template containing troff requests for a report that is written as output_file containing information derived from the execution of execute_command_line. At those places in the report where information is to be inserted, the user writes a special request of the form ".al X" where "X" specifies the information to be inserted. Following are these special requests, showing the nature of the information that is inserted:

- `.al _SE` A summary, by program unit, of segment execution counts. *(Segments are described in Section 3.2.)*
- `.al _TO` A summary, by program unit, of segment execution counts as fractions of the total.
- `.al _Z` A summary, by program unit, of segments not executed.
- `.al _L` A listing of the program showing segment execution counts.
- `.al _AS` A summary, by program unit, of assertion failures. *(Assertions are described in Section 3.2.)*
- `.al _DY` A summary, by program unit, of statement type execution counts.
- `.al _IC` Any comments contained in sections of the program demarcated by SED switches. *(SED switches are described in Section 3.2.)*

Files _.ANSTATICSUM and _.ANANNOLIST, created in the working directory by inst, contain information about the instrumented program being executed. These files must be present when rundoc is invoked.
Independently of input_file and output_file, rundoc creates a file called _XREPORT in the working directory containing a summary report of segment execution counts and assertion failure counts.

When rundoc executes execute_command_line, it is possible to record the execution of each segment in a selected range, thereby creating a trace of segment numbers through the selected range of segments. When rundoc is invoked, tracing is activated if there exists a file named Trace_Requests in the working directory. The file has a series of entries, one per line beginning in column 1, of the form

IIIIcJJJJ

where IIII and JJJJ are integers right-justified in a field of four columns (i.e., integers with format I4) and c is comma, plus, or minus. For example, Trace_Requests might consist of the four lines

5, 12
10, 20
18+, 3
30-, 5

These lines are interpreted as follows: Trace each execution in the segment ranges 5 to 12 and 10 to 20; trace each execution of segment 18 and the three segment executions that follow; trace each execution of segment 30 and the five preceding segment executions. Note that the ranges may overlap; the first two lines above could be replaced by

5, 20

The trace results are written as entries to a file named Trace_Output that rundoc creates in the working directory. In this file, the unsigned entries are segment numbers. A pair of entries like 5 -8 means that segment 5 was executed eight times in succession.

After executing rundoc, output_file may be processed like any text file containing troff requests to produce the required report. No external troff macros are inserted, nor is preprocessing of output_file (e.g., by eqn) assumed by rundoc. The user is, of course, free to assume that such facilities can be used to process output_file and to construct input_file accordingly.

4. Fortran Transformation Facilities

Transformation refers to the mapping of Fortran source code from one form to another. The transformations may be motivated by the desire to improve the way the code is presented to humans, by reformatting, rebuilding declarations, and restructuring, thereby making the program easier to understand and hence easier to maintain, communicate, and modify. Even well-engineered code may need to be transformed so that it conforms to conventions for a software collection. Other transformations are motivated by the need to change the precision of the arithmetic or to take advantage of a particular hardware architecture.

Humans see a Fortran program represented as Fortran text. In Section 3 we mentioned other representations, namely, token/comment streams and parse tree/symbol tables. A transformation may be regarded as a mapping of the program from one representation to another (possibly of the same type) in the course of which particular characteristics of the
program are changed. For example, the precision transformer tool ISTPT maps a parse tree/symbol table to a token-comment stream that represents an "equivalent" program in which the arithmetic is performed to different precision. As with all transformations, precision transformation requires the operation of a sequence of tools. The original text is first transformed to a token-comment stream, which is then transformed to a parse tree/symbol table. The changes to convert precision are made as the parse tree/symbol table representation is "flattened" to a token-comment stream. Finally, Fortran text is constructed from the token-comment stream. The user invokes a script that sets up and invokes the proper sequence of tools.

The scripts take Fortran text as input and produce Fortran text as output, but none of the underlying tools converts Fortran text directly into Fortran text. Rather, the transformation tools manipulate representations of Fortran in which all the formatting information has been removed. When the transformed program is mapped back into Fortran text, it is reformatted according to options supplied by the user in a so-called "Polish option file". (The tool ISTPL, which transforms a token/comment stream to Fortran text, is known as "Polish".) To use any of the transformation scripts, the user should be familiar with the use of the Polish option editor ISTPO, invoked by script polx, to create Polish option files (see Section 4.1.1).

4.1. Fortran Formatting

As pointed out above, Fortran formatting is always a part of Fortran transformation and indeed may be the objective of the transformation. The script pol provides for Fortran formatting with a Polish option file that can be constructed with script polx, interactively. Typically, Fortran programmers construct and retain Polish option files that satisfy their formatting tastes or the requirements of journals and working groups.

4.1.1. polx

Create interactively a new, or modify an existing, Polish option file using a menu-driven editor.

Invocation:

    polx Polish_option_file

where *Polish_option_file* is the name of a new or existing Polish option file. If the file is new, default option settings are displayed in a set of menus. If the file exists, the option settings it contains are displayed in the menus.

Upon invoking polx the user receives a prompt and may respond with

    MENU DIR

to obtain a menu of menus, as follows:
Menus available:

- BASIC  - Basic Operating Parameters
- COMMON - Commonly-used Options
- UNCOMMON - Uncommon Customisations
- CONVERSION - Conversion Options
- BLANK_LINES - Blank Line Insertion
- LINE_BREAK - Long Line Break Priorities
- SPACING1 - Token Spacing Parameters
- SPACING2 - More Token Spacing Parameters

In addition to commands for changing options, there are eight commands in the editor that provide for displaying menus, obtaining help, and reading and writing Polish option files. These commands may be displayed by typing

? 

A user who wishes to display the BASIC menu would type

MENU BASIC

The display would appear as

Menu: BASIC

SEQRQD: Add sequence numbers = .FALSE.
RLBFMT: Relabel FORMAT statements = .FALSE.
RLBSTM: Relabel executable statements = .FALSE.
MOVEF: Move FORMAT statements to end = .FALSE.
DOCONI: End each DO-loop on a CONTINUE = .FALSE.
IOTHCO: Put CONTINUE on each labelled stmt = .FALSE.
TRACE: Display progress messages = .FALSE.
ERRCMT: Insert error messages as comments = .TRUE.
CONCHR: Continuation character = '+'

To change the setting of an option, the user types a new value that is checked for acceptability by the editor. For example, a user who wants Fortran formatted so that all the executable statements are relabelled would type

RLBSTM=.TRUE.

Actually the user would only need to type

rlbs=.t
since the input commands are case insensitive and the editor recognizes any unambiguous curtailment of an acceptable option and setting. As a further illustration, the command

\textit{md}

is equivalent to

\textit{MENU DIR}

When all desired changes have been made, the command

\textit{EXIT}

or, simply,

\textit{e}

writes the Polish option file and exits from the editor.

If they wish to do so, users may exercise control over such formatting particulars as the number of spaces before particular tokens or the priorities used in breaking continuation lines; altogether a total of 473 available options. Any option not changed retains its previous setting, which is the default if the option has never been changed.

4.1.2. \textit{pol}

Format a Fortran program.

Invocation:

\texttt{pol Polish_option_file Fortran_source_file}

\textit{Polish_option_file} is a Polish option file that controls the formatting of the Fortran program units in \textit{Fortran_source_file}. Default formatting options are used if "-" appears instead of a file name for the argument \textit{Polish_option_file}. The resultant Fortran is sent to standard output and may be redirected to a file.

4.2. Declaration Standardization

The tool ISTDS rebuilds the declarative part of a Fortran program. This results in the explicit declaration of all items implicitly typed (either by IMP\textsc{licit} statements or Fortran naming conventions) and the specification of array declarators in type statements rather than DIM\textsc{ension} statements. IMP\textsc{licit} statements are removed, and separate declarations are made for arrays and scalars of any given type. 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 for variables not used in the program.

As an illustration, the program
subroutine example(in,out,1state)
implicit logical(1)
real in
common /boxes/ box1,box2,box3
out = log(in) + exp(box1) - box2**2
if(out.lt.box3) then
  lstate=.true.
else
  lstate=.false.
  box3=box3+1.
endif
return
end

would be transformed by ISTDS to the following (assuming all declaration standardization and formatting options have default values):

SUBROUTINE example(in,out,1state)
C       .. Scalar Arguments ..
REAL in,out
LOGICAL lstate
C       ..
C       .. Scalars in Common ..
REAL box1,box2,box3
C       ..
C       .. Intrinsic Functions ..
INTRINSIC exp,log
C       ..
C       .. Common blocks ..
COMMON /boxes/box1,box2,box3
C       ..
out = log(in) + exp(box1) - box2**2
IF (out.LT.box3) THEN
  lstate = .true.
ELSE
  lstate = .false.
  box3 = box3 + 1.
END IF
RETURN
END

The comments like "C       .. Common blocks .." are called section headers and the comments "C       .." are section trailers. Section headers and trailers are inserted by ISTDS only if not already present; in particular, ISTDS could be applied to a program unit more than once without duplicating these comments.
The script `decs` invokes the sequence of tools required to effect declaration standardization.

4.2.1. decs

Rebuild the declarative part of a Fortran program.

Invocation:

```
    decs decs_option_file Polish_option_file Fortran_source_file
```

The declarative part of each Fortran program unit in `Fortran_source_file` is rebuilt according to the options in `decs_option_file` (discussed below). If `decs_option_file` is "-" instead of a file name, default options are used. The declaration-standardized program is formatted according to the formatting options in `Polish_option_file`. If this argument is "-" instead of a file name, default formatting options are used. The resultant Fortran is sent to standard output and may be redirected to a file.

The options in `decs_option_file` are separated by blanks, tabs, or carriage returns. Any unambiguous curtailment of an option name or option value is acceptable; the first four characters of a name or value are always unambiguous. If the same option name occurs more than once in the file, the last value prevails. The possible options follow.

- **ardich**
  If this option is present, the array declarators for arrays in COMMON are included in the COMMON statements. If the option is absent (the default), these array declarators are in type statements.

- **chlkbrk**
  If this option is present, separate CHARACTER statements for character variables of different lengths are produced. If the option is absent (the default), all character variables are declared in a single CHARACTER statement.

- **exhdr**
  If this option is present, a section header "Executable Statements" is inserted immediately preceding the first executable statement of each program unit. If the option is absent (the default), such a section header is not inserted.

- **ictwcb**
  If this option is present, the type declarations for all COMMON variables immediately follow the COMMON statement that contains them. If the option is absent (the default), the sections "Scalars in Common" and "Arrays in Common" are created to contain these type declarations.

- **mode=<value>**
  `<value>` is either `declare_implicit_names` or `rebuild_declaratives`. If the option setting is `mode=declare_implicit_names`, ISTDS adds a section "Previously untyped names" that includes type declarations for implicitly typed names; other sections are not rebuilt, and none of the other options, except `exhdr`, have meaning. If the option is absent or its setting is `mode=rebuild_declaratives` (the default), the declaration sections are rebuilt according to the other options.

- **notrailers**
  If this option is present, section trailers are not added. If the option is absent (the default),
section trailers are created when a section is created.

order=<arg>,<arg>...

This option, when present, specifies the order in which the type statements are placed within each section. The arguments are type codes as follows:

<table>
<thead>
<tr>
<th>Type Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE COMPLEX</td>
<td>7</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>5</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>4</td>
</tr>
<tr>
<td>REAL</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>3</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>6</td>
</tr>
</tbody>
</table>

If this option is present, all seven arguments must be included. The default order is shown above and is equivalent to the option setting order=7,5,4,2,1,3,6.

remove_unused_names=<value>

<value> is yes, log, or no. If the option setting is remove_unused_names=yes, the declarations for names not actually used in the program are silently removed. If the option setting is remove_unused_names=log, unused names are removed and a warning message is issued; unused EQUIVALENCE-associated or COMMON variables and dummy arguments are not removed. If the option is absent or its setting is remove_unused_names=no (the default), unused names are not removed.

4.3. Precision Transformation

The tool ISTPT converts a program in which the arithmetic is done in single precision to one in which the arithmetic is done in double precision, or vice versa. Irreversible differences occur only when the code is originally of mixed precision or an exceptional intrinsic function is encountered; see (6) below. A warning or error message is accompanied by the insertion of a corresponding comment in the transformed code.

Following are the major details of the transformations performed by ISTPT:

(1) The keyword REAL is changed to/from DOUBLE PRECISION, and the keyword COMPLEX is changed to/from DOUBLE COMPLEX.

(2) Declarations are added for implicitly typed names that have changed precision.

(3) Real constants are transformed to/from double precision and, if appropriate, "'DO'" is added to or deleted from the end of the constant.

(4) Single-precision complex constants are transformed to/from double-precision complex constants. If both parts of a single-precision complex constant are integers, the first is made into a double-precision constant by adding the suffix "'DO'".

(5) The E-format format descriptor is transformed to/from the D-format descriptor. (This does not change the "'Ew.dEe'" form of the E-format descriptor because there is no D-format equivalent.)

(6) REAL-valued intrinsic functions are transformed to/from DOUBLE PRECISION functions by replacing the name. Similarly, COMPLEX-valued intrinsic functions are transformed to/from DOUBLE COMPLEX functions. Generic functions are not converted except when used as arguments to a routine (in which case they lose their generic properties). ISTPT transmits an error message if it finds an occurrence of an intrinsic function that it cannot transform. Certain difficult cases are treated as follows:
Single to Double

DPROD(expr1, expr2) is transformed to expr1*expr2, with parentheses around the expressions if necessary.

MAXI(...) is transformed to INT(MAX(...)).

AMAXO(...) is transformed to DBLE(MAX(...)).

MIN1 and AMIN0 are treated similarly to MAX1 and AMAX0.

AIMAG(...) is transformed to DBLE(AIMAG(...)).

Double to Single

DBLE(AMAXO(...)) is transformed to AMAXO(...).

DBLE(AIMAG(...)) is transformed to AIMAG(...).

(7) EQUIVALENCE statements are checked to ensure that their meaning does not change during the transformation; if it does, an error message is produced.

Precision transformation is effected by the scripts apt and dapt. The latter performs declaration standardization (see section 4.2) on the precision-transformed program, thus combining the effects of apt and decs.

4.3.1. apt

Transform a single-precision Fortran program to double precision, or vice versa.

Invocation:

apt <s or d> Polish_option_file Fortran_source_file

Each Fortran program unit in Fortran_source_file is transformed to single or double precision according to whether the first argument is s or d. The precision-transformed Fortran is formatted according to the options in Polish_option_file. The resultant Fortran is sent to standard output and may be redirected to a file.

4.3.2. dapt

Transform a single-precision Fortran program to double precision, or vice versa, and standardize the declarations.

Invocation:

dapt <s or d> decs_option_file Polish_option_file Fortran_source_file

Each Fortran program unit in Fortran_source_file is transformed to single or double precision according to whether the first argument is s or d. The declarative part of each precision-transformed program unit is standardized according to the options in decs_option_file, and the precision-transformed, declaration-standardized Fortran is formatted according to the options in Polish_option_file. The resultant Fortran is sent to standard output and may be
4.4. Name Changing

Toolpack/1 provides two facilities for modifying strings in names in a Fortran program. The first facility functions like a Fortran-intelligent stream editor by permitting string changes to be specified in terms of Fortran-language conditions. The second facility enables long names (up to 31 characters) to be converted to standard-conforming names.

Fortran-intelligent stream editing is provided by two tools working at different levels of representation of a Fortran program. ISTCN maps a token/comment stream into a new token/comment stream with specified changes. ISTCR transforms a symbol table into a new symbol table with specified changes. In both cases, the changes are specified in terms of regular expressions, as described in Appendix B. Script cname controls the use of both tools so that, for the user, there is a single stream editing capability permitting two classes of changes.

The Fortran standard limits the length of a name to six characters, and this limitation is enforced by Unix f77. The tool ISTLS, controlled by script lname, detects long names in a program and substitutes Fortran 77 standard-conforming names supplied by the user. (One use of the tool might be simply to render compilable a program for which no archival change is desired.) The mapping of long names to standard-conforming names is defined in a conversion file that pairs the long names with their standard replacements.

4.4.1. cname

Change strings in names in a Fortran program that satisfy conditions derived from information in the lexical token stream and/or symbol table.

Invocation:

cname Polish_option_file command_file_1 command_file_2 Fortran_source_file

Each Fortran program unit in Fortran_source_file is modified according to files command_file_1 and command_file_2, which contain specifications for token-stream-related and symbol-table-related changes, respectively. If one or the other of these files is not present, its place in the command line must be occupied by "-". The transformed Fortran is formatted according to the options in Polish_option_file. If this argument is "-" instead of a file name, default formatting options are used. The resultant Fortran is sent to standard output and may be redirected to a file.

Each change specification in command_file_1 is one line having the format

<names> <comments> <strings> <hol> <fold> <pat> = <rep>

where each of <names>, <comments>, <strings>, <hol>, and <fold> is t or f (or upper case T,F), signifying "true" or "false". The first four of these designate whether the change is to be applied to tokens of type NAMES (of variables, program units, COMMON blocks, etc.), COMMENTS, CHARACTER CONSTANTS, or HOLLERITH CONSTANTS, respectively. <fold> designates whether the pattern matching that determines if a change will be made is to be case insensitive. Changes are made to tokens of the specified type that match the pattern specified by <pat>, which is a regular expression as described in Appendix B. The pattern may not have leading or trailing spaces (except as defined in character classes; see (7) in Appendix B) or an embedded "=". The replacement string <rep> (see Appendix B) also may

redirected to a file.
not have leading or trailing spaces. When a name is changed, the substitute is checked for legality. If the test fails, an error occurs.

Some examples of change specifications related to the token stream follow. (Note that the spacing of the t/f flags and the patterns is not significant.)

(1) t t t t f FRED = I

Change the string FRED to I in all names, comments, and strings (both character and Hollerith). Note that FRED in names will be changed to I, but FREDA and IFRED will not. In strings and comments all occurrences of FRED will be changed.

(2) t f f f t I<*>* = &>1A

Change all names starting with I or i to the string of characters following the I or i, in upper case, with A appended. Thus IFRED will be changed to FREDA, ibiLl to BILLA, and i itself to A.

(3) t t f ft ?* = &>0

Change all names and comments to upper case.

(4) ft f f %<?*?[ ]+<?*>* = &1 &>2&<3

Change all comments by removing leading blanks and converting the text to lower case, except for the first non-blank character in each comment which is converted to upper case. The comment character itself is preserved.

command_file_1 may contain a maximum of 256 change specifications. It may contain comment lines that either are blank or begin with "##" in column 1.

Each change specification in command_file_2 is one line having the format

```
/[<pu-re>/] <s_type>[:<d_types>] ([<qual>]) <pat> = <rep>
```

(Components enclosed in "[]" are optional.)

<pu-re> is an optional regular expression that specifies the program units in which the change is be applied. For example, if <pu-re> is "*/S*/", the change is applied in any program unit whose name starts with "S". If this component is absent, the change is applied in all program units.

<s_type> is a flag that specifies the symbol type to be affected by the change. <s_type> may be any of the following:

- COM A COMMON block name.
- ENT An entry point.
- NAM Names of unknown usage, e.g., variables declared but not used.
- PAR A parameter.
PRO A procedure reference.
PU Program unit names. An unnamed main program is called "$MAIN"; an unnamed BLOCK DATA subprogram is called "$BLOCKDATA".
STA A statement function.
VAR A variable.

<d_types> is an optional string of data type qualifiers for the symbol type, permitting specification, for example, of INTEGER procedures and REAL variables. If this string is not present, all data types are acceptable. The notion of "data type" is enlarged to permit inclusion of program units, BLOCK DATA, and generic intrinsic functions as "types". The type qualifiers and the data types they qualify are

- B: block data
- P: program unit
- I: integer
- R: real
- L: logical
- X: complex
- D: double precision
- C: character
- G: generic, for standard intrinsic functions

<qual> is an optional qualifier (the "( )" are not optional) that specifies the use of a symbol. If this qualifier is present, a symbol must be used in the specified way in order to be changed. If the qualifier is not present, any appearance of the symbol is acceptable. The usage qualifiers and the uses they specify are

- ARG: is used as an argument.
- ASS: appears in an ASSIGN statement.
- COM: appears in a COMMON statement.
- DAT: appears in a DATA statement.
- DUM: is a dummy argument or formal parameter to the program unit.
- EQU: appears in an EQUIVALENCE statement.
- EXP: appears in an expression.
- EXT: appears in an EXTERNAL statement.
- FUN: is used as a function name.
- IND: is used as a DO-loop index.
- INT: appears in an INTRINSIC statement.
- REA: appears in a READ input list.
- SET: is set by an assignment statement.
- SF: is a dummy argument or formal parameter of a statement function.
- STA: is a standard intrinsic function.
- SUB: is used as a subroutine name.
USE Is used in some way. This is equivalent to the union of ARG, ASS, DAT, DUM, EQU, FUN, IND, REA, SF, and SUB.

Changes are made to symbols that meet the qualifications and match the pattern specified by <pat>, which is a regular expression as described in Appendix B. The pattern may not have leading or trailing spaces (except as defined in character classes; see (7) in Appendix B) or an embedded "=". The replacement string <rep> (see Appendix B) also may not have leading or trailing spaces.

Note that not all qualifiers or data types are relevant to all symbol types. For example, neither data types nor qualifiers have any meaning when applied to COMMON block names.

After a replacement has been made, the resulting symbol is checked for both legality and uniqueness within the program unit. If either of these tests fails, an error occurs.

Some examples of change specifications related to the symbol table follow. (Note that the spacing of the components of the change specification and the patterns is not significant.)

(1) VAR:I() FRED = I

In all program units, change integer variable FRED to I, whatever its usage.

(2) VAR:() FRED = I

In all program units, change variable FRED to I, whatever its usage or data type.

(3) /S*/ VAR:IR(INDEX) I = LOOP

In all program units whose names start with S, change any integer or real variable I used as a DO-loop index to LOOP.

(4) VAR:I (USE) I<?*> = &1A

In all program units, change any used integer variable whose name starts with I so that its new name consists of the characters following the I, if any, with A appended at the end. For example, IFRED will be changed to FREDA.

(5) PU: ( ) <????>? <?> = I&1&2
    ENT: I ( ) <????>? <?> = I&1&2
    PRO: (FUNC) <????>? <?> = I&1&2

Change all integer function and integer function entry point names that have six characters by dropping the fifth character and prefixing an I. Change all references to those functions as well. (Note that the uniqueness of a new program unit name is not checked; uniqueness checks are made only on changes within a program unit.)

(6) CO () <???> <???> = &2&1

Change the names of all COMMON blocks that have six characters to reverse the first and last groups of three characters.
command_file_2 may contain a maximum of 1000 change specifications. It may contain comment lines that either are blank or begin with "#" in column 1.

4.4.2. Iname

Transform a Fortran program containing long names to a program with standard names.

Invocation:

\texttt{Iname name_conversion_file Polish_option_file Fortran_source_file}

\textit{name_conversion_file} is the name of either a new file to be created by this invocation of \texttt{Iname} or a file created by a previous invocation. The file contains a list of long names and their replacements. When a long name that is already in \textit{name_conversion_file} is discovered in any Fortran program unit in \textit{Fortran_source_file}, it is replaced by the specified name, provided the replacement name is standard conforming and does not duplicate a name already used in \textit{Fortran_source_file}. If the long name is not in \textit{name_conversion_file} or if the proposed replacement name is unacceptable, the user is prompted for a different replacement, until the long name and its replacement are added to the conversion file. Names longer than 31 characters are truncated to 31 characters. The data type of a replacement name is not checked, but script decs, for instance, can be used to ascertain data types so that a suitable replacement may be selected. The same conversion file may be used for other programs.

The transformed Fortran is sent to standard output and may be redirected to a file. It is formatted according to the options in \textit{Polish_option_file}. If this argument is "-" instead of a file name, default formatting options are used. Note that replacement names are mapped to upper case in \textit{name_conversion_file}; however, their ultimate case in the resultant Fortran is governed by a Polish option setting. A file ".LOG" showing the name changes by token number is created in the working directory. The token numbers may be correlated with the program using the listing produced by script getlst.

4.5. Fortran Structuring

The tool ISTST rebuilds the flow of control within each program unit to a standardized form, based on the structuring algorithm described in [1]. Unlike Unix \texttt{struct} which is based on the same algorithm, the input and output languages for ISTST are both Toolpack/1 Target Fortran 77 (see [3]).

Following are major features of the restructuring transformations:

(1) In the transformed Fortran, CONTINUE statements appear only as the final statements of DO-loops or the initial statements of loops created using GO TO statements.

(2) Arithmetic IF statements are changed to logical IF statements unless the three transfer labels are distinct.

(3) Assigned GO TO statements are changed to computed GO TO statements, and the ASSIGN statements themselves are changed to assignment statements. If there is only one possible target for an assigned GO TO, it is changed to a simple GO TO that may be removed during further structuring.

(4) ASSIGN statements that assign FORMAT statement labels (for use in READ and WRITE) are left unchanged.

(5) Any transfer of control to the END statement of a subroutine or function is replaced by a RETURN.
(6) Executable statements (except for END, END IF, etc.) that follow an unconditional transfer of control are removed.

(7) Comments that are either before or embedded within an executable statement remain with that statement in the transformed code. Comments before a flow-of-control statement (e.g., CONTINUE, GO TO, or END IF) appear in the transformed code at the place they would occupy if the comments themselves were executable statements.

(8) A program that contains a loop entered in two places is not altered. Such a program can always be manually converted to one that can be structured by ISTST.

The script stf invokes the sequence of tools required to effect program structuring.

4.5.1. stf

Rebuild the flow of control in a Fortran program to standardized form.

Invocation:

    stf Fortran_source_file

The Fortran program units in Fortran_source_file are restructured and then reformatted using default formatting options. The transformed Fortran is sent to standard output and may be redirected to a file.

As an illustration, let Fortran_source_file contain the program

```fortran
LOGICAL A, B
1     READ(5,*) A, B
   IF (.NOT. A) GO TO 3
   WRITE(6,*) 'A IS TRUE'
   IF (B) GO TO 2
   WRITE(6,*) 'B IS FALSE, MORE INPUT, PLEASE'
   GO TO 1
3     WRITE(6,*) 'EITHER A IS FALSE OR BOTH A AND B ARE TRUE'
      STOP
2     GO TO 3
END
```

Its structured form follows. (The lower-case names are the result of a formatting option.)

```fortran
LOGICAL a, b
10 CONTINUE
    READ (5,FMT=*) a, b
    IF (a) THEN
        WRITE (6,FMT=*) 'A IS TRUE'
        IF (.NOT.,b) THEN
            WRITE (6,FMT=*) 'B IS FALSE, MORE INPUT, PLEASE'
            GO TO 10
        END IF
    END IF
END IF
```
WRITE (6,FMT=*) 'EITHER A IS FALSE OR BOTH A AND B ARE TRUE'
STOP
END

As a second example, let Fortran_source_file contain the program

INTEGER A,B,C,D
READ(5,*) A,B,C,D
C Quit when first integer is -1.
  IF(A.EQ.-1) GO TO 6
  IF (A.EQ.B .OR. C.EQ.D) GO TO 200
  DO 10 I=1,10
  WRITE(6,*)'THROUGH THE LOOP'
10    IF ((A+B).EQ.I) GO TO 220
200   DO 210 I=1,2
210   WRITE(6,*)'WRITE THIS TWICE'
220   GO TO 5
6     CONTINUE
END

The structured form of this program is

INTEGER a,b,c,d
10 CONTINUE
  READ (5,FMT=*) a,b,c,d
C Quit when first integer is -1.
  IF (a.NE.-1) THEN
    IF (a.NE.b .AND. c.NE.d) THEN
      DO 20 i = 1,10
      WRITE (6,FMT=*) 'THROUGH THE LOOP'
      IF ((a+b).EQ.i) GO TO 10
20    CONTINUE
  END IF
  DO 30 i = 1,2
  WRITE (6,FMT=*) 'WRITE THIS TWICE'
30    CONTINUE
  GO TO 10
END IF
END
4.6. DO-Loop Transformation

The performance of programs executing on vector computers is significantly improved when the number of accesses to memory can be reduced. The tools ISTUD, ISTCD, and ISTSB perform certain transformations on nests of DO-loops that typically occur in routines that perform linear algebra computations. The transformed Fortran, when compiled on a vectorizing compiler, accesses memory less frequently than the original. The script ucs provides access to these tools.

These are research tools. They were constructed as part of a project to investigate programming environments for machines with advanced architectures. Prospective users of ucs should consult [6] and [4] for detailed discussions of the transformations performed by ISTUD, ISTCD, and ISTSB and of the Fortran code to which these transformations may be usefully applied.

4.6.1. ucs

Transform nests of DO-loops matching certain paradigms so that the transformed code executes more efficiently on vector machines.

Invocation:

```
ucs unrolling_depth Polish_option_file Fortran_source_file
```

The Fortran program units in `Fortran_source_file` are transformed by the tools ISTUD, ISTCD, and ISTSB. The integer `unrolling_depth` specifies the depth to which outer loops of nests are "unrolled" by ISTUD. The transformed Fortran is sent to standard output and may be redirected to a file. It is formatted according to the options in `Polish_option_file`. If this argument is "-" instead of a file name, default formatting options are used.

5. Miscellaneous Facilities

This section contains a description of the file comparison and version control facilities available in Toolpack/1. Also covered are a facility for removing unneeded files created by the operation of the scripts and a simple "help" facility.

5.1. File Comparison

The tool ISTFD enables two Fortran programs to be compared at the level of their token/comment stream representations (textual differences such as spacing and continuations have disappeared). Moreover, the comparison is completely insensitive to the case of the text. The script fdiff invokes the tools that effect this comparison.

The tool ISTDC enables two files containing numerical data to be compared whereby differences less than a given tolerance are ignored. For example, the two files might be outputs from the same program run on two computers with different round-off algorithms. Case differences of text in the files are ignored as with ISTFD. The script dac invokes ISTDC.

5.1.1. fdiff

Compare two Fortran programs at the token/comment stream level.

Invocation:

```
fdiff Fortran_source_file_1 Fortran_source_file_2 [nc]
```
Fortran_source_file_1 and Fortran_source_file_2 are files containing one or more Fortran program units. The token/comment stream representations of these two files of Fortran programs are compared. A report of their differences is sent to standard output and may be redirected to a file. If the optional third argument \texttt{nc} is present, differences in comments are ignored.

It is worthy of note that the default formatting options used with Fortran transformation tools are such that \texttt{fdiff} with the \texttt{nc} argument present reports no differences between a program and its transformation under \texttt{pol} using the default options. The formatting tool may insert blank lines, but these are considered comments.

5.1.2. \texttt{dac}

Compare two data files, neglecting certain formatting differences and those numerical differences less than a given tolerance.

Invocation:

\begin{verbatim}
dac data_file_1 data_file_2 [n]
\end{verbatim}

A report of differences between \texttt{data_file_1} and \texttt{data_file_2} is sent to standard output and may be redirected to a file. The optional argument \texttt{n} is a positive integer threshold; numbers in the data files are considered equal if their absolute difference is less than $10^{-n}$. If \texttt{n} is not specified, the default value \texttt{n=6} is used. The case of letters in text or as part of a numerical datum is ignored. For example, the three lines

\begin{verbatim}
output = .0000001
OUTPUT = 2.0E-7
OUTput = 3.d-07
\end{verbatim}

are regarded as identical when the tolerance is the default. On the other hand, when entries are flagged as non-identical, the tool may count a larger number of differences than are truly significant.

5.2. Version Control

A version file contains the original text of a text file, the differences that define successive versions of the file, comments associated with each version, and the time of creation of each version. Successive versions of documents or codes under development can be maintained in a version file. Such a file consumes less space than that for a series of complete copies (because versions are reconstructed from differences) and contains information about the evolution of the text (differences, comments, and time of creation). Software libraries that exist in different manifestations (e.g., single/double, IBM/CDC/DEC) can be archived in version files, saving space because the duplicate information appears only once. The tool ISTVC permits a version file to be created and updated, comments or differences to be retrieved, and any version of the text to be retrieved either by version number or by the time when the version was current. The script \texttt{vcon} invokes ISTVC and the \texttt{vi} editor.

5.2.1. \texttt{vcon}

Create, edit, and retrieve versions of a file.
Invocation:

vcon option version_file [text_file]

version_file is the name of a new or existing version file and option is a code that specifies the action to be taken—retrieval, update, etc. text_file is a new or initial version to be incorporated into the version file; it is present on the command line for two options.

Possible values of option follow.

-u
Update (create) version_file using text_file as the next (first) version. The user is prompted to make comments that become associated with the version created. Comments are terminated by a line containing the single character "".

-n
Same as -u except that the user is not prompted to make comments. (Bypassing the prompt and response is useful when vcon is called from other scripts.)

-e
If version_file exists, invoke Unix vi to edit the most recent version and update version_file with the edited version. If version_file does not exist, invoke vi to create Version 1. The user is prompted to make comments.

-r
version_file, assumed to exist, is reinitialized using the latest version as Version 1. The user is prompted to make comments.

-vn
Retrieve version n. If n is omitted, retrieve the current version. The retrieved text is sent to standard output and may be redirected to a file.

-\Y/M/D/H/m/S
Retrieve version current as of year Y, month M, day D, hour H, minute m, second S. Least significant units are zero if omitted. The retrieved text is sent to standard output and may be redirected to a file.

-dn
Retrieve differences between versions n and n-1. If n is omitted, retrieve all differences. (Note: -d1 gives the text of Version 1.) A report is sent to standard output and may be redirected to a file.

-cn
Retrieve time created and comments appended for version n. If n is omitted, retrieve all comments. A report is sent to standard output and may be redirected to a file.

There is a limit to the number and complexity of the differences between successive versions. When this limit is exceeded in an attempt to update the version file, the proposed new version, although too different to permit updating, may be used to create a new version file. In particular, with option -u or -n, text_file exists and may be used to create a new version file; with option -e, the edited text, if too different to permit updating, is saved in a file called "_.SAVEDFILE" in the working directory and may be used to create a new version file.
ISTVC recognizes the special string "V#" in text being retrieved and replaces it with the version number. This occurs using option -e, -r, -v, or -t. Thus, for example, the version number can be automatically generated in comments or output messages in source code. However, this feature should be used with caution since the replacement will not be reversed if the retrieved text is used to construct an update.

5.3. File Removal

The scripts and the tools they invoke create files and subdirectories in the working directory. Some of these are scratch files and are removed before normal termination of the script. Others may be retained if they contain either error messages or part of the normal output specified in this guide. Created files include "IST.CMD" and the directory "_.TOOLPACK" which are always removed in normal operation. The names of other Toolpack-created files always have "_." as their first two characters. The utility script discard enables the user to remove unneeded Toolpack-created files easily.

5.3.1. discard

Remove unneeded files created by Toolpack/1.

Invocation:

discard

The file "IST.CMD" and the directory "_.TOOLPACK" are removed if they exist. Additionally, the user is asked about removing each file in the working directory whose name begins with "_.". A response beginning with "y" causes the file to be removed. Any other response causes it to be retained.

5.4. Help

As pointed out in Section 2, the command syntax and other helpful information are displayed when the name of a script with no arguments is typed (discard and scripts, which have no arguments, are the only exceptions). Further, the script scripts causes the summary from Section 1 to be displayed.

5.4.1. scripts

Print a summary of the Unix shell scripts that control the Toolpack/1 tools.

Invocation:

scripts

The summary of the scripts from Section 1 of the Users' Guide is sent to standard output and piped through Unix more.

Acknowledgments

We have profited from discussions with Toolpack developers Malcolm Cohen, Stephen Hague, Ian Hounam, and Robert Iles from NAG, Ltd., and Jeff Kvam from NAG, Inc. We thank Gail Pieper for her help in preparing the manuscript.
References


Appendix A. Semantic Errors

Following are the semantic errors reported by the indicated scripts. Syntactic errors are not covered here, nor are the warning messages from semantic checking.

Errors reported by syn.

Note: sem and pfort also report these errors and then terminate without attempting deeper analysis.

(1) A letter is given more than one IMPLICIT type setting.
(2) DO-loops or IF-blocks are illegally nested.
(3) There is a transfer into the range of a DO-loop or an IF-block.
(4) There is an assignment within a DO-loop to the control variable.

Errors reported by sem.

Note 1: pfort also reports these errors and then terminates without attempting deeper analysis.

Note 2: The underlying tool, ISTSA, makes several passes, each continuing to the next only if no errors are reported. Hence, after all reported errors are corrected, sem (or pfort) should be run again to find possible further errors reported in a later pass.

(1) An array that is not a dummy argument has a bound that is not a constant expression.
(2) A variable in an array bound of an adjustable array is neither a dummy argument nor in a COMMON block.
(3) The variable on the left side of an assignment statement is an unsubscripted array.
(4) The expression on the right side of an assignment statement is not compatible with the type of the variable on the left.
(5) A variable appears in more than one COMMON statement.
(6) A dummy argument appears in a COMMON block.
(7) Blank COMMON appears in a BLOCK DATA subprogram. (Actually this is illegal only if the COMMON block is initialized, but ISTSA does not check at that level of detail.)
(8) A COMMON block is initialized in a DATA statement in other than a BLOCK DATA subprogram.
(9) An item initialized in a DATA statement in a BLOCK DATA subprogram is not in a COMMON block.
(10) A repetition count in a DATA statement is not a positive integer.
(11) A DATA statement does not have exactly the same number of values as items.
(12) A value in a DATA statement is not of compatible type with the associated item.
(13) A name given as a value in a DATA statement is not a PARAMETER.
(14) A dummy argument appears as an item in a DATA statement.

(15) The control variable in a DO statement is not a scalar of type INTEGER, REAL, DOUBLE PRECISION, half-precision integer, or quadruple-precision real.

(16) A limit expression in a DO statement is not of type INTEGER, REAL, DOUBLE PRECISION, half-precision integer, or quadruple-precision real.

(17) The terminal statement of a DO-loop is an unconditional GO TO, assigned GO TO, arithmetic IF, RETURN, STOP, END, DO, block IF, ELSE IF, ELSE, or END IF statement.

(18) An intrinsic function reference has an incorrect number of arguments or arguments of incorrect type or structure. (Two items have different structure if, for example, one is an array name and the other a variable name, or one is a variable name and the other is an external function name.)

(20) A reference to a function or subroutine, present in the file being checked, has an incorrect number of arguments or arguments of incorrect type or structure.

(21) The name of a subprogram used as an actual argument is declared neither INTRINSIC nor EXTERNAL.

(22) Two or more references to the same function or subroutine have inconsistent argument lists.

(23) An expression in a length specification, PARAMETER statement, or array declarator is not a constant expression.

(24) A subscript or subrange expression within a data-implied-DO list is neither the control variable nor a constant expression.

(25) An expression in a computed GO TO statement, length specification, data-implied-DO limit, RETURN expression, array bound, or substring range limit is not of type INTEGER.

(26) An expression in a logical IF, block IF, or ELSE IF statement is not LOGICAL, half-precision logical, or quarter-precision logical.

(27) An expression in an arithmetic IF statement, DO statement, input/output implied-DO list, or array subscript is not of type INTEGER, REAL, DOUBLE PRECISION, half-precision integer, or quadruple-precision real.

(28) The expression in a PAUSE or STOP statement is neither an unsigned integer of at most five digits nor a character string.

(29) A format identifier is not a label, a scalar or array of type CHARACTER, or an asterisk.

(30) An array reference has an incorrect number of subscripts.

(31) A rule set forth in the standard, extended as described in Numerical Algorithms Group Publication NP1313, for constructing expressions using the +, -, *, /, and ** operators is violated.

(32) A substring refers to an unsubscripted array (rather than to a variable or array element).

(33) The conditional statement of a logical IF is a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

(34) A statement function is recursive.
(35) A statement function uses a substring specifier that is not in the argument to a function reference.

(36) A statement function expression is of incompatible type with the statement function name.

(37) The dummy arguments in a statement function are not distinct.

(38) Control variables in nested input/output implied-DO-loops are not distinct.

(39) An internal file specifier in an input/output statement is not a variable, array, array element, or substring.

(40) An auxiliary input/output statement specifies an internal file.

(41) List-directed formatting is used with an internal file.

(42) List-directed formatting is used with a direct access file.

(43) An end-of-file specifier (END=) is used with a direct access file.

(44) The control information list of an input/output statement does not contain exactly one unit identifier.

(45) The list in an INQUIRE statement contains neither exactly one file identifier nor exactly one unit identifier.

(46) More than one format specifier appears in the control information list of an input/output statement.

(47) A format specifier occurs in a statement other than READ, WRITE, or PRINT.

(48) The variable in an assigned GO TO statement either is not an integer scalar or is not set by an ASSIGN statement.

(49) A BLOCK DATA subprogram contains either an EXTERNAL statement or a non-specification statement.

(50) The variable in an ASSIGN statement is not an integer scalar.

(51) A RETURN statement appears in a main program.

(52) A statement is out of the order set forth in the standard.

(53) Dummy arguments in a SUBROUTINE or FUNCTION statement are not distinct.

(54) An ENTRY or program unit name appears in a SAVE statement.

(55) A dummy argument appears in a SAVE statement.

(56) A referenced statement label is not defined.

(57) A COMMON block mentioned in a SAVE statement is not defined.

(58) Two EQUIVALENCE statements are in conflict.

(59) An EQUIVALENCE statement contains a dummy argument.

(60) An EQUIVALENCE statement conflicts with the definition of a COMMON block.

(61) Variables in two different COMMON blocks are associated by EQUIVALENCE.

(62) A COMMON block extends backward as a result of association by EQUIVALENCE.

(63) Either a program unit name appears more than once or there is more than one main program or unnamed BLOCK DATA subprogram.

(64) Either named COMMON blocks that have the same names are of different size or appear inconsistently in SAVE statements, or a COMMON block appears in more than one
BLOCK DATA subprogram.

Errors reported by pfort.

(1) A variable is referenced but not set anywhere.
(2) The function value in a function subprogram is not set.
(3) A COMMON block not appearing in a SAVE statement is contained in each of two program units but is not contained in any program unit that is always active when either of the two containing program units is active. (The COMMON block becomes undefined between references to the two subprograms.)

*Note 1:* "Errors" 4-8 are actually "unsafe references." They are not deviant unless the value of the dummy argument in the called subprogram is changed at execution time. Further, the underlying tool, ISTPF, does not evaluate array indices; it treats the entire array as a single item. Hence, it may report an unsafe reference when the values of the array's subscripts are such that the reference could never actually be unsafe. To help the user identify this situation, errors 4-8 are flagged "unsafe" when they involve arrays and "error" when they do not.

(4) An actual argument is an expression, and the corresponding dummy argument may be modified by the called subprogram.
(5) An actual argument is associated with two dummy arguments at least one of which is not an array, and either of them may be modified by the called subprogram.
(6) An actual argument is in a COMMON block to which the called subprogram or a subprogram that it calls has access, and either the corresponding dummy argument or the COMMON block may be modified by the called subprogram.
(7) A statement function definition includes a function reference in which the dummy argument of the statement function is an actual argument in the function reference, and the corresponding dummy argument may be modified by the referenced function.
(8) An actual argument is an active DO-loop index and the corresponding dummy argument may be modified by the called subprogram.

*Note 2:* "Errors" 9-26 are not actual violations of the Fortran 77 standard; they are Fortran constructions that may not be portable to all machines.

(9) Within a COMMON block, COMPLEX and DOUBLE PRECISION entities do not precede all other entities.
(10) A non-initial element of an array is associated by EQUIVALENCE with an item of different type.
(11) A function with the same name as a standard intrinsic function is used to replace a standard intrinsic function.
(12) A standard intrinsic function is explicitly typed.
(13) An external non-intrinsic function is not explicitly typed.
(14) A value in a DATA statement is not of the same type as the associated item.

(15) A variable used as part of an adjustable dimension in an array declarator is declared following its appearance in the array declarator.

(16) A DO-loop has a non-integer control variable.

(17) There is a PAUSE, BACKSPACE, or ENDFILE statement.

(18) An assigned GO TO statement has an attached label list.

(19) A format identifier is an INTEGER, REAL, or LOGICAL array.

(20) There is a TL or T format edit descriptor.

(21) A FUNCTION name is used as the DO-loop control variable in the function.

(22) A DO-loop control variable appears in a limit expression for that DO-loop.

(23) A name is common to two or more of the following in the same program unit: a COMMON block, a statement function name, an ASSIGN variable, a dummy variable, or a statement function dummy variable.

(24) A relational operator between strings is other than .EQ. or .NE. (The intrinsic functions LGE, LLE, etc. must be used instead.)

(25) A character string variable exceeds 255 characters.

(26) A character constant exceeds 64 characters.
Appendix B. Regular Expressions

An ordinary character is any member of the character set of a Toolpack installation plus the empty string at the beginning and the null character at the end of lines. We assume that the character set contains at least the standard Fortran character set plus ‘?’, ‘%’, ‘$’ (currency symbol’), '[' , ']' , ‘‘’’, ‘‘<’’, ‘‘>’’, ‘‘@’’, ‘‘&’’, ‘‘newline’’, and ‘‘tab’’. A metacharacter is a string of ordinary characters that has been given special meaning in the context of string matching. Below are the metacharacters defined in terms of the strings of ordinary characters that they match. A regular expression is a string of metacharacters, that is, a string of ordinary characters with their special meanings.

1. An ordinary character not given special meaning in (2) - (11) matches itself.

2. A ‘‘?’’ matches any single ordinary character.

3. A ‘‘%’’ at the beginning of a regular expression matches the empty string at the beginning of a line.

4. A ‘‘$’’ (currency symbol') at the end of a regular expression matches the null character at the end of a line.

5. A simple metacharacter\(^2\) followed by a ‘‘*’’ matches zero or more occurrences of that metacharacter (closure).

6. A simple metacharacter\(^2\) followed by a ‘‘+’’ matches one or more occurrences of that metacharacter (anchored closure).

7. A string (or class\(^3\)) of ordinary characters enclosed in square brackets ‘‘[ ]’’ matches any character in the string unless the first character is a ‘‘~’’ in which case the class matches any character not in the string (other than newline). Appropriate classes of characters may be abbreviated to a character range of the form a-z, 0-9, P-Y, etc.

8. The metacharacters ‘‘<’’ and ‘‘>’’ open and close tag fields and are not part of the matching process. For information on recalling tag fields, see below.

9. Any ordinary character preceded by an ‘‘@’’ (escape), including the character ‘‘@’’ itself, matches the character as an ordinary character, that is, without any special meaning it may have.

10. The two-character symbol ‘‘@n’’ matches the newline character. The two-character symbol ‘‘@t’’ matches the tab character (cntrl-I).


A concatenation of regular expressions is itself a regular expression.

It is possible to tag up to 9 character fields in a regular expression. During substitutions the tagged fields may be recalled in any order in the replacement text. Tag fields are opened using the metacharacter ‘‘<’’ and closed using the metacharacter ‘‘>’’. Tag fields may be nested but may not overlap. The fields are numbered 1 to 9 in the order in which they are opened. Tag field 0 is the whole matched pattern. In the replacement text the tag fields are recalled using from 1 to 3 character symbols of the following forms (where n is 0 to 9):

---

1. The currency symbol will vary from country to country.
2. A simple metacharacter is any one of (1), (2), (7), (9), or (10).
3. Characters specified within a class are always case sensitive.
& Equivalent to \"&0\".

&n Recall tagged field \"n\" as originally entered

&>n Recall tagged field \"n\" shifted to upper case

&<n Recall tagged field \"n\" shifted to lower case
Distribution for ANL-87-12

Internal:

J. M. Beumer (2)
W. R. Cowell (40)
B. S. Garbow (40)
H. G. Kaper
A. B. Krisclunas
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