User's Guide to Symbolics FORTRAN

Introduction to the Symbolics FORTRAN User's Guide

Purpose and Scope

0.0.151. Purpose

This document describes the FORTRAN implementation developed for the Symbolics Genera environment. In conjunction with the Symbolics Common Lisp and Symbolics Genera references listed in this section, it provides the information necessary to edit, compile, debug, and run Symbolics FORTRAN programs under Symbolics Genera.

0.0.152. Scope

This document provides a conceptual overview of Symbolics FORTRAN, and is designed to get you started using Symbolics FORTRAN. It is not a step-by-step instruction guide or a comprehensive reference manual. We discuss the following topics:

- The concepts that distinguish Symbolics' implementation of FORTRAN 77 from those on conventional systems.
- The benefits derived from using Symbolics FORTRAN under Symbolics Genera.
- The extensions that Symbolics integrates into FORTRAN 77.
- The tools available for running Symbolics FORTRAN.
- The interface for calling Lisp functions.

0.0.153. The User and FORTRAN

Symbolics FORTRAN is based on standard FORTRAN 77, so this document does not discuss the language itself, except to describe the extensions developed for this implementation. If you have little experience with the standard language, refer to the *American National Standard Programming Language FORTRAN, ANSI X3.9-1978* (hereafter called the ANSI Standard or the Standard), which is included with the purchase of Symbolics FORTRAN.

0.0.154. The User and Lisp

Be aware that Symbolics FORTRAN depends heavily on the Genera software environment. You can use Symbolics FORTRAN knowing relatively little about the Lisp language and Symbolics Genera, but familiarity with Lisp gives you access to the wider functionality available in Symbolics Genera.

This document, however, does *not* provide detailed instructions for using Symbolics Genera effectively; information about Symbolics Common Lisp and Symbolics Genera facilities is included only to the extent that it affects Symbolics FORTRAN.

0.0.155. Symbolics Genera References

If you know little about Symbolics Genera, consult the Genera documentation set with particular attention to the following sections:

- For a listing of documentation notation conventions used in this document, see the section "Notation Conventions".
- For a quick reference guide, in particular a summary of techniques for finding out about the software environment, see the section "Getting Help".
- For a guide to the Symbolics text editor, see the section "Zmacs Manual".
- For information on error handling, see the section "Conditions".

0.0.156. More Help

For a brief explanation of the Lisp syntax encountered in this manual, see the section "Lisp Syntax for FORTRAN Users".

0.0.157. Conventions

This *User's Guide* uses the standard conventions as well as the following additional conventions. For a listing of the standard conventions, see the section "Notation Conventions Quick Reference".

In addition to the standard conventions, this manual uses the following notation:

Appearance in document	*Representing*
`lispobject`	FORTRAN reserved words and program, subroutine, library, and package names in running text; also FORTRAN source code.

Introduction to Symbolics FORTRAN

Summary

0.0.158. Introduction

Symbolics FORTRAN implements the full ANSI FORTRAN 77 Standard (FORTRAN X3.9-1978).

The ANSI FORTRAN 77 Standard supersedes earlier dialects of FORTRAN, and, while supporting most features of these early dialects, it contains several incompatible extensions. Appendix A of the Standard gives a complete description of these incompatibilities.

0.0.159. Components

Symbolics FORTRAN consists of the following components:

- Several language extensions to FORTRAN 77, which are of particular use to Symbolics Genera programmers and help in porting from other FORTRAN 77 implementations.
- A compiler for the full FORTRAN 77 language, as described in the ANSI Standard.
- Extensions to Zmacs, the standard text editor, to support FORTRAN language editing, using the language-specific capability of Zmacs.
- Support of the Metering Interface.
- A Lisp-compatible run-time library, permitting full access to Genera's input/output facilities, including access to files over network connections.
- A symbolic Debugger, permitting debugging of FORTRAN code at the source level. You can use the Debugger from the Lisp Listener or from the Display Debugger interface.

Comparison of Symbolics FORTRAN with Other Implementations

In most computing environments...	*Under Symbolics Genera...*
You must compile entire files even if only a small change is made to the code.	Compilation is incremental; you can compile only the changed function.
You leave the editor and then compile the file.	You can compile a routine or file from the editor.
The compiler produces warnings; you must find the source manually.	The editor processes the compiler warnings and provides commands that help you locate the source associated with a compiler warning.
A link-and-load step is required for all programs.	Programs are immediately executable after compiling; you do not have to use a separate linking step.
The association between a library and a main program occurs at link-and-load time.	The association between a library and a main program is made at run time.
After loading and execution, a program disappears from memory; you must reload the	Once loaded, you can rerun programs without reloading.

program to rerun it.	
Block data subprograms are associated with main programs at link-and-load time.	Block data subprograms are associated with main programs at run time.
Variables are always reinitialized whenever the program is linked and loaded.	You can request at run time that variables be initialized at every execution of the program or left with values from previous executions.
Uninitialized variables are not detected.	Uninitialized variables are detected, unless you specifically request at compile time or at run time that all variables be initialized to zero.
To use the debugging facility you must explicitly call the debugger *before* running a main program.	The Debugger is automatically invoked on run-time errors, and you can enter it at any time during execution.

Using Symbolics FORTRAN for the First Time

Introduction

0.0.160. Contents

This chapter provides instructions for installing and loading Symbolics FORTRAN. It then explains how to enter, compile, and run a sample FORTRAN program called `quadratic`, which solves the quadratic equation $ax^2 + bx + c = 0$ for unknown x.

0.0.161. Scope

This chapter runs through the essential edit-compile-debug cycle, with little explanation of the conceptual underpinnings. Crossreferences point you to the correct chapters for additional information.

Installing FORTRAN

0.0.162. Procedure

1. After you have successfully installed the Genera 8.0 system software, boot a Genera 8.0 world.

2. You need to indicate where the FORTRAN system definition resides; to do this, create the file SYS:SITE;FORTRAN.SYSTEM and type the following attribute list and form in the file.

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-

(si:set-system-source-file "fortran" "sys: fortran; fortran")
```

3. Load the contents of the Symbolics FORTRAN tape into your file system by typing the following command from a Lisp Listener:

   ```
   Restore Distribution
   ```

4. Once you load the contents of the tape onto your SYS host, type the following to the Command Processor:

   ```
   Load System Fortran :Query No
   ```

   You can now use Symbolics FORTRAN.

0.0.163. Notes

1. To avoid loading the FORTRAN system every time you want to use it, save a world on disk with FORTRAN already loaded into it. See the section "Save World Command". We strongly recommend that you run the ephemeral garbage collector when using the FORTRAN compiler. Ephemeral garbage collection is usually on by default, but if it is not, type the following to the Command Processor:

   ```
   Start GC :Ephemeral Yes
   ```

2. Symbolics FORTRAN includes the *User's Guide to Symbolics FORTRAN*. You can view the online documentation via Document Examiner. To do so, press SELECT D.

   You might want to become familiar with the overall structure of the documentation before viewing individual chapters. To do so, read in the *User Guide*'s table of contents; click Right on [Show] in the Command menu, and when prompted type the following and press RETURN.

   ```
   user's guide to symbolics fortran
   ```

   The Current Candidates window displays the complete list of chapter titles, section titles, and so on. Each entry displayed in the Current Candidates window is mouse-sensitive; clicking on any entry brings its associated documentation into the Viewer.

Loading FORTRAN

0.0.164. Procedure

1. Boot the machine, if necessary.

2. Log in.

3. Look at the *herald*, the multiline message appearing on the screen after booting. If the herald lists the FORTRAN software, see the section "Using the Editor for the First Time".

   If the herald does not list the FORTRAN software, use the Load System command to load Symbolics FORTRAN; the system is called `fortran`.

   Type the following to the Command Processor and press `RETURN`:

   ```
   Load System Fortran :Query No
   ```

Using the Editor for the First Time

0.0.165. Purpose

This section enables you to run a simple FORTRAN program immediately — without having to read through the entire manual first or understand the entire Symbolics Genera environment.
For more thorough descriptions of the topics presented in this section, see the following sections:

"The Editor"
"Editor Extensions for FORTRAN"
"Summary of Standard Editing Features"

0.0.166. Procedure

1. Invoke Zmacs by pressing `SELECT E`.

2. Use the Find File command, `c-X c-F`, to create a new buffer for the FORTRAN program `quadratic` that you are creating. To invoke the Find File command, hold down the `CTRL` key as you press the `X` and then the `F` key in succession.

   Choose a name for the buffer. When prompted, type the name into the minibuffer at the bottom of the screen and press `RETURN`. Use the pathname conventions appropriate for the host operating system where the file resides.

   Example: Suppose you want to create a new FORTRAN source file, `quadratic.fortran`, which you want to store in your home directory on a Symbolics computer called Quabbin (q for short). Type the following:

```
c-X c-F q:>fred>quadratic.fortran
```

Make sure the name includes the proper FORTRAN *file type* (extension) for your host; for example, quadratic.fortran is the correct name for a file residing on LMFS, the Genera file system. For a table of FORTRAN file types, see the section "FORTRAN File Types".

3. Use m-X Fortran Mode to set the mode of the buffer to FORTRAN: Press the META and X keys together. Then type fortran mode and press RETURN. Answer yes to the quesition of whether to set FORTRAN mode for the file and the attribute list.

   The mode line, located below the editor window, displays "Fortran".

4. Put the buffer in the **ftn-user** package. Type m-X Set Package and when prompted for a package name type ftn-user. Answer yes to the question of whether to set the package for the file and the attribute list. The attribute list is the first line of a file and looks something like:

```
c -*- Mode: FORTRAN Package: FTN-USER -*-
```

5. Enter the sample program quadratic:

```
          program quadratic
          implicit none
          real a,b,c,discriminant,x1,x2
          print 1
1         format (/)
          print 10, 'Solves the equation:'
          print 10, 'A*X**2 + B*X + C = 0 for X'
          print 10, 'Enter A, B, and C:'
10        format (a)
          read *, a, b, c
          discriminant = b**2 - 4*a*c
          x1 = (-b + sqrt(discriminant)) / (2*a)
          x2 = (-b - sqrt(discriminant)) / (2*a)
          print 20, x1, x2
20        format ('The roots are: ',2g12.5)
          end
```

   For the editor commands controlling cursor movement and text manipulation: See the section "Editor Extensions for FORTRAN". See the section "Summary of Standard Editing Features".

   Check the code in your buffer against the example.

6. If you want to save the source code in a disk file, use c-X c-S.

Compilation, Execution, and Debugging in the Example

The material in this section is covered in greater detail in these chapters:

"Compiling Symbolics FORTRAN Programs"
"Executing FORTRAN Programs"
"Debugging Symbolics FORTRAN Programs"

0.0.167. Procedure

1. With the cursor positioned anywhere within the program text, use `c-sh-C` to compile the program. The *typeout window* at the bottom of the screen displays the stages of the compilation as they complete. If the compilation completes successfully, the typeout window displays "QUADRATIC compiled". Go to Step 3.

2. If the typeout window displays error messages, press the space bar to erase the typeout window and return to the editor window; correct the code.

   If the typeout window displays compiler warnings, press the space bar, if necessary, to erase the typeout window and use `m-X` Edit Compiler Warnings to resolve the warnings.

3. Press `SUSPEND` to enter a Zmacs breakpoint. A small window appears at the top of the screen, overlaying a portion of the editor window.

4. Type:
   ```
   Execute Fortran quadratic
   ```

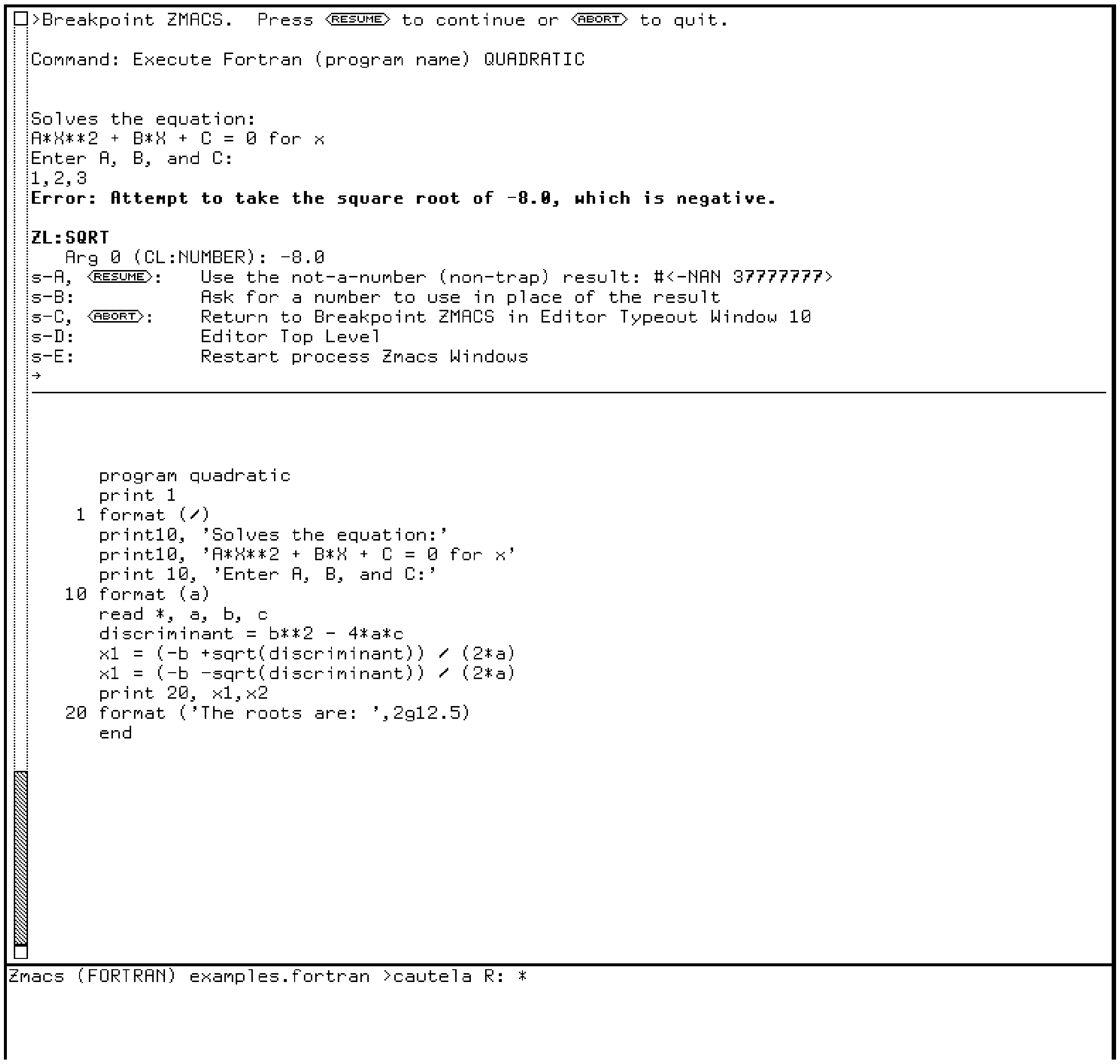
5. The program prompts:
   ```
   Enter A, B, and C:
   ```
   Type:
   ```
   1,0,-9 RETURN
   ```
   The window displays the following:
   ```
   The roots are:   3.0000     -3.0000
   ```

6. Rerun the program, this time causing a run-time error. When the program prompts for input, type:
   ```
   1 2 3 RETURN
   ```
   Symbolics Genera automatically invokes the Debugger (identifiable by `Error:`), which displays a descriptive error message ("Attempt to take the square root of -8.0, which is negative"), and then a list of suggested actions and their outcomes. If the error occurred in a Lisp function, `press` `c-B` to see the backtrace, and click on the FORTRAN subprogram that caused the error. See figure !, 154

For example, pressing s-D or ABORT causes you to return to the editor breakpoint, from which you can rerun quadratic. Pressing c-m-W enters the Display Debugger, which presents you with a full-screen interface to the Debugger. All the commands available in the full-screen interface to the Debugger are also available directly in the Lisp Listener, or in the typeout window.

7. Press ABORT twice — once to leave the Debugger and a second time to leave the breakpoint and return to the editor window.

   This step completes the edit-compile-debug cycle for the sample program. We suggest that you spend some time editing the code, recompiling, and rerunning the program until you feel comfortable with the process.



```
>Breakpoint ZMACS.  Press RESUME to continue or ABORT to quit.

Command: Execute Fortran (program name) QUADRATIC


Solves the equation:
A*X**2 + B*X + C = 0 for x
Enter A, B, and C:
1,2,3
Error: Attempt to take the square root of -8.0, which is negative.

ZL:SQRT
   Arg 0 (CL:NUMBER): -8.0
s-A, RESUME:   Use the not-a-number (non-trap) result: #<-NAN 37777777>
s-B:           Ask for a number to use in place of the result
s-C, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window 10
s-D:           Editor Top Level
s-E:           Restart process Zmacs Windows
→



      program quadratic
      print 1
    1 format (/)
      print10, 'Solves the equation:'
      print10, 'A*X**2 + B*X + C = 0 for x'
      print 10, 'Enter A, B, and C:'
   10 format (a)
      read *, a, b, c
      discriminant = b**2 - 4*a*c
      x1 = (-b +sqrt(discriminant)) / (2*a)
      x1 = (-b -sqrt(discriminant)) / (2*a)
      print 20, x1,x2
   20 format ('The roots are: ',2g12.5)
      end

Zmacs (FORTRAN) examples.fortran >cautela R: *
```

Figure 65. Producing a run-time error in the sample program.

Extensions to FORTRAN 77

Summary

0.0.168. Eight Major Extensions

Symbolics FORTRAN defines several extensions to standard FORTRAN 77. Several of these result from the strong hardware data-type checking provided by the Genera computing environment. This feature offers a greatly increased ability to detect errors that on conventional systems are discovered only by more laborious means, if at all.

Symbolics FORTRAN supports the following language extensions:

- Arbitrary-precision integers
- Detection of uninitialized variables
- Strong data-type checking
- Syntactic extensions
- Interaction with Lisp
- FORTRAN package system
- Integer to logical coercion
- Department of Defense extensions to the Standard
  - `do` statement
  - `do while` statement
  - `enddo` statement
  - `include` statement
  - `implicit` statement
  - Binary pattern processing
    - Logic operations
    - Shift operations
  - Bit substrings

Discussion of Extensions

Arbitrary-Precision Integers

Symbolics FORTRAN supports arbitrary-precision integers, called *bignums*; as a result, all integers are immune from overflow. Suppose you have a typical iterative factorial routine:

```
      integer function factorial(n)
      implicit none
      integer n,i
      factorial=1
      do 100 i=n,1,-1
         factorial=factorial*i
100   continue
      return
      end
```

In FORTRAN on conventional machines, this routine works properly until the product computed in the variable `factorial` overflows. Then either a hardware overflow is signalled, or the computation delivers the incorrect answer with no warning whatsoever. With Genera, however, the computation completes and returns the correct answer independent of the value of *n*.

Since the Standard does not restrict the range of precision of integers (see the Standard, section 1.3.2), the accommodation of bignums provides great flexibility developing code for machines with differing word sizes. It is also helpful in solving mathematical problems, since integers do not exhibit the anomalous overflow behavior of conventional machines.

In conjunction with arbitrary-precision integers, Symbolics FORTRAN supports formatted output of large integers. Hence, format specifications such as I200 are meaningful.

The only operation for which arbitrary-precision integers are *not* valid is unformatted input/output. In this case, integers must be between -2^{31} and 2^{31}-1.

Detection of Uninitialized Variables

Symbolics FORTRAN detects uninitialized data (other than character data), so that an error condition results if a variable is used before it is assigned a value.

FORTRAN implementations on conventional machines, which cannot easily check for uninitialized data, generally initialize all data to zero before beginning program execution.

If you do not wish to use this feature, a compiler option exists that causes all variables to initialize to zero. See the section "Compiler Option: Initializing FORTRAN Program Data".

Strong Data-Type Checking

Genera provides strong hardware data-type checking among logical, integer, and real data. Thus, the hardware prevents a program error due to falsely regarding data as equivalent, for example, in the case where the exponent and mantissa of a real number is interpreted as an integer.

This strong data-type checking does not extend to the complex and double-precision data types. For example, where data equivalence is used to store values of the complex and real type, the complex data type is represented by a pair of real numbers, one of which is confused with a single real number. The hardware representation for double-precision data is in fact a pair of integers. (See the Standard, sections 8.2-8.2.5, for information on equivalence.)

Syntactic Extensions

Symbolics FORTRAN supports five syntactic extensions:

- *Tab*. The ASCII horizontal tab character (HT) is permitted in source code and is treated as if it were replaced by sufficient spaces to pad out to the next multiple of 8 columns. For example, if the first column is 1, the tab is positioned at column 9.
- *Commas*. No comma is required between format specification items in a format statement, with the following exceptions:
  - Unless it is necessary to distinguish between Ew.d and Ew.dE.e (or Dw.d and Dw.dE.e), the edit descriptors for specifying the input/output (I/O) of real, double-precision, and complex data types.
  - In Hollerith items preceded by a digit; for example, without a comma, a41ha could be read as either a, 41ha, or a4, 1ha.

  See the Standard, chapter 13, particularly section 13.5.9.2.2. The motivation for this extension is that the ANSI Standard rules for omitting commas are relatively complicated to explain and remember.
- *Symbolic names*. Symbolic names, which identify FORTRAN program units, can be longer than six characters. For example, `program quadratic` is a valid statement. See the Standard, chapter 18.
- *Hollerith data*. Symbolics FORTRAN allows Hollerith data as recommended in the Standard, appendix C. You can use `Hollerith` values to initialize a real, logical integer, complex, or double-precision value. Accordingly, A format is legal for output of these quantities.
- *Case*. Symbolics FORTRAN does not distinguish between upper- and lowercase letters, except in character strings, Hollerith constants, and H edit descriptors. See the Standard, section 3.1.1.

Interaction with Lisp

Symbolics FORTRAN defines two extensions for interacting with Lisp. A new scalar data type called `lispobject` allows you to call Lisp routines from FORTRAN. By declaring a variable the type `lispobject`, you can represent any Lisp data object. See the section "`lispobject`: FORTRAN Data Type for Handling Lisp".

A new FORTRAN declaration statement, `lispfunction`, allows the declaration of an existing Lisp function that you can call from a FORTRAN routine. See the section "`lispfunction`: Type Declaration Statement".

FORTRAN Package System

Symbolics are large-scale virtual-memory, single-user computers; many programs — the editor, the compiler, and so on — coexist in the same environment (address space). Once FORTRAN routines (and Lisp functions, as well) are loaded into the environment, they remain there until they are explicitly removed, replaced by recompilation, or until you cold boot the machine (that is, until a fresh version of the Genera software is loaded).

Since you might have two large FORTRAN programs that both define a subroutine named `load`, how can Symbolics Genera distinguish between them? A similar conflict might arise between FORTRAN programs and Lisp functions. Symbolics Genera provides a mechanism for separating the like-named functions in different programs by assigning each its own distinct context, or *namespace*. The namespace is called the *package*.

The package name precedes and distinguishes two identically named functions.

Example: **ftn-user:cube** and **games:cube** define two different functions named **cube**, one in package **ftn-user**, the other in package **games**.

Packages in Fortran

The names of symbols you use within a FORTRAN package do not conflict with the names of symbols in any Lisp packages or in any other FORTRAN packages. Unlike FORTRAN, Lisp programs need to access basic Lisp functions and variables, such as **cond** or **nil**, from the **global** package, and such symbol names are reserved. In FORTRAN no symbols are reserved, and it is possible for one of your FORTRAN routines or variables to have the name of a predefined Lisp symbol like **nil**.

f77:package-declare-with-no-superiors is the special form used for declaring your own FORTRAN packages. Once you declare a FORTRAN package, you can use names for variables and functions without fear of conflict with the same names in other FORTRAN or Lisp packages. Symbolics FORTRAN also provides a built-in FORTRAN package called **ftn-user**. **ftn-user** is the default package of the FORTRAN system.

For instructions on defining your own packages, see the section "Declaring a FORTRAN Package".

For information on assigning package names to buffers, see the section "Editing Basics for Symbolics FORTRAN Programs".

For a general discussion of the Lisp package facility, see the section "Packages".

Integer to Logical Coercion

For compatibility with many other FORTRAN 77 implementations, Symbolics FORTRAN supports coercion from integer to logical types. A non-zero integer is treated as logical `.true.`, and integer zero is treated as logical `.false.`.

Example: Assuming that `l` is declared to be of type logical and `i` of type integer, the following statement is valid:

```
l = i
```

It is treated as if it were:

```
l = i .ne. 0
```

The compiler generates a semantic warning regarding the coercion. Note that such coercion applies only to values, and not to parameters. That is, any time an integer is passed to a logical formal parameter, no coercion occurs.

Binary Pattern Processing

In each of the defined functions it is assumed that integer numbers are represented in binary form.

Logic operations. The logic operations provided are the Boolean functions **or**, **and**, **eor**, and **not**. These operations are implemented as integer intrinsic functions. The implicit type for **or**, **and**, and **eor** is indicated by the use of I as the first letter of their function names. You can make the arguments *m* and *n* single variables, array elements, or expressions of type integer. After execution of the functions, the arguments remain unchanged. The operations are performed on all corresponding bits of the two operands.

`ior` (*m*,*n*) Inclusive **or**. The arguments, *m* and *n*, are combined according to the following truth table.

m	*n*	*Function value*
0	0	0
1	0	1
0	1	1
1	1	1

`iand` (*m*,*n*) Logical **and**. The arguments, *m* and *n*, are combined according to the following truth table.

m	*n*	*Function value*
0	0	0
1	0	0
0	1	0
1	1	1

`not` (*m*) Logical complement. The argument *m* is logically complemented according to the following truth table.

m	*Function value*
0	1
1	0

`ieor` (*m*,*n*) Exclusive **or**. The arguments, *m* and *n*, are combined according to the following truth table.

m	*n*	*Function value*
0	0	0
1	0	1
0	1	1
1	1	0

Shift operations. The shift operations provided are logical and circular. These operations are implemented as integer intrinsic functions and have two arguments, *m* and *n*. Simple variables, array elements, or integer expressions are permitted as arguments.

Arg	*Meaning*
m	Specifies the value (binary pattern) to be shifted.
n	Specifies the shift count.

Value	*Meaning*
$n > 0$	Indicates a left shift.
$n = 0$	Indicates no shift.
$n < 0$	Indicates a right shift.

The arguments are not changed by the shift operations.

`ishft` (*m*,*n*) Logical shift. All bits representing the argument *m* shift *n* places. Bits shifted out from the right end are lost. Zeros are shifted in from the right end on a left shift.

`ishftc` (*m*,*n*,*ic*) Circular shift. The right-most *ic* bits of the argument are shifted circularly *n* places; that is, the bits shifted out of one end are shifted into the opposite end. No bits are lost. The unshifted bits of the result are the same as the unshifted bits of the argument *m*. The argument *ic* must be greater than or equal to 1.

Bit Substrings

In each of the defined functions integer numbers are represented in binary form.

`ibits` (*m*,*i*,*len*) Bit extraction. *m*, *i*, and *len* are integer expressions. This intrinsic function extracts a field of *len* bits from string *m* starting with bit *i* (counted from right to left where the right-most bit is bit 0) and extending left for *len* bits. The resulting field is right-justified and the remaining bits are set to 0.

`call mvbits` (*m*,*i*,*len*,*n*,*j*)
Bit move intrinsic subroutine. *len* bits are moved from position *i* of argument *m* to position *j* of argument *n*. Arguments *m* and *n* are permitted to be the same storage unit. All arguments must be of type integer.

Bit processing. You can test and change individual bits of a storage unit with the following routines for bit processing. The functions have two arguments, *j* and *k*. For both *j* and *k*, simple variables and array elements are permitted; *k* can also be an expression. Be sure all arguments are of type integer.

Arg	*Meaning*
j	Specifies the corresponding binary pattern.
k	Specifies the selected bit (right-most bit is bit 0). If *k* is negative, the result of the function is undefined.

`btest` (*j*,*k*)	Bit testing. This routine is implemented as a logical intrinsic function. The *k*th bit of argument *j* is tested. If it is 1, the value of the function is `.true.`. If it is 0, the value of the function is `.false.`. Since the arguments are not changed by the function reference, *j* can also be an integer expression.
`ibset` (*j*,*k*)	Set bit. The result of this intrinsic function is *j* with the *k*th bit set to 1.
`ibclr` (*j*,*k*)	Clear bit. The result of this intrinsic function is *j* with the *k*th bit set to 0.

Using the Editor to Write Symbolics FORTRAN Programs

Editing Basics

0.0.169. Zmacs Commands

The Zmacs text editor provides a full range of general-purpose commands for writing and editing programs. These commands include reading and writing files, basic cursor-movement commands, and text-manipulation commands.

In addition, Symbolics FORTRAN includes editor extensions that understand FORTRAN syntax.

- See the section "Editor Extensions for FORTRAN".

- See the section "Summary of Standard Editing Features". See the section "Understanding Notation Conventions". See the section "Notation Conventions Quick Reference".

- For a discussion of the `HELP` key and command *completion*: See the section "Program Development Help Facilities".

0.0.170. Procedure

This procedure summarizes the steps for writing or editing FORTRAN source code.

1. Select Zmacs in one of the following ways:
   - Press `SELECT E`.
   - Click Left on [Edit] in the System menu.
   - Issue the Select Activity command at the Command Processor, supplying Zmacs or Editor as the activity name.

2. Use `c-X c-F` to read an existing file into the buffer, or `c-U c-X c-F` to create a buffer for a new file. When you add a FORTRAN file type to the file name, `c-U c-X c-F` automatically sets the mode of the buffer to FORTRAN.

   When prompted, type the full pathname of the file in the minibuffer (the small editing window at the bottom of the screen) and press `RETURN`. Use the pathname conventions appropriate for the host operating system.

   Example: Suppose you want to create a new FORTRAN source file, cube.fortran, that you want to store in your home directory on a Symbolics host called Quabbin (q for short). You type the following:

   ```
   c-U c-X c-F q:>fred>cube.fortran
   ```

   The correct file type for a FORTRAN source file depends on the host. See the section "FORTRAN File Types".

3. Use the command `m-X` Fortran Mode to set the buffer mode to FORTRAN if the mode is not currently in Fortran. This mode allows you to use the special editor extensions for FORTRAN. The mode line, situated below the editor window and near the bottom of the screen, displays `(Fortran)`.

   If you intend to write FORTRAN code most of the time, you might want to set the default major mode in your init file to FORTRAN. This sets the mode automatically to FORTRAN whenever you invoke an editor window. Type the following form in your init file:

   ```
   (setf zwei:*default-major-mode* :fortran)
   ```

4. Use `m-X` Update Attribute List to make the attribute list reflect FORTRAN mode. Alternatively, you can type the attribute list yourself.

   Note: The attribute list is optional, but if you choose to supply one, you must make it the first line of the source file. A sample attribute list might look like the following:

   ```
   {-*- Mode: FORTRAN -*- }
   ```

   For more information on the attribute list, see the section "File Attribute Lists".

5. To set or change the package name in the attribute list, use `m-X` Set Package and type the package name. The command offers to create the package if it does not yet exist. Alternatively, you can enter the package name in the at-

tribute list by typing; `Package:` and the name of a package that you have previously defined with the special form **f77:package-declare-with-no-superiors**. The FORTRAN system provides the predefined FORTRAN package, **ftn-user**.

Example:

```
{-*- Mode: FORTRAN; Package: FTN-USER -*- }
```

Once the package is set, FORTRAN routines in the file are defined as belonging to an existing package whenever the file is read into an editor buffer. Unless you explicitly override the package in the attribute list, the code in the buffer compiles into and loads to the indicated package.

The status line, the last line of the screen, reflects the updated package name.

6. Use `m-X` Reparse Attribute List whenever you make changes to the attribute list. Reparsing causes the changes to take effect.

   Note: Changing packages does not affect previously compiled code.

7. Use editor commands to create or alter file contents, compile the code (`c-sh-C`), save the source file (`c-X c-S` or `c-X c-W`), and so on.

FORTRAN Implementation-Dependent Values

Introduction

This chapter presents information specified by the Standard as implementation-dependent. The topics covered are:

- I/O processor-dependent values
- Data representation
- FORTRAN file types
- Record length of `open` statement
- Standard FORTRAN functions

I/O Processor-Dependent Values

Currently, the FORTRAN I/O system's capabilities reflect exactly those in the ANSI Standard.

The I/O Processor-Depenedent Values lists the processor-dependent values required for I/O statements (see the Standard, chapter 12) and particular I/O format specifications (see chapter 13).

For each value, the page and line in the ANSI Standard are provided.

Arbitrary-precision integers and a syntactic extension regarding the use of commas in format specifications are discussed elsewhere: See the section "Extensions to FORTRAN 77".

FORTRAN Data Representation

Symbolics FORTRAN adheres to the proposed IEEE format for floating-point numbers (See Jerome Coonen, et al., "A Proposed Standard for Binary Floating Point Arithmetic: Draft 8.0 of IEEE Task P754", Microprocessor Standards Committee, IEEE Computer Society, *Computer*, March 1981, pages 51-62.

Symbolics FORTRAN supports the ASCII character set, as extended for Symbolics Genera.

The table, Representation of Various Data Types, gives the range for each data type and the accuracy of the floating-point types.

FORTRAN File Types

Symbolics Genera determines whether a file contains FORTRAN source code by looking at its file type. The correct type for a FORTRAN source file varies with the operating system. The table below shows the file types for various systems.

FORTRAN source files are compiled to Lisp compiled-code (object) files. The correct file type of a compiled-code file is also system-dependent. For a list of the file types by host: See the section "File Types of Lisp Source and Compiled Code Files".

Standard FORTRAN Functions

Table 6. I/O Processor-Dependent Values, by Ref. to ANSI Standard.

Page	*Line*	*Section no., name*	*Description*
12-2	1	12.1.1 Formatted Record	The length of a formatted record is limited only by the size of virtual memory.
12-6	16	12.3.1 Unit Existence	Valid unit numbers are non-negative integers.
12-6	4	12.3.3 Unit Specifier and Identifier	An asterisk identifying an external unit refers to standard input, unit 5, and standard output, unit 6.
12-17	5	12.9.5.2.3 Printing of Formatted Records	Supports standard carriage control with the blank, 0, 1, and + characters.
13-7	29	13.5.7 P Editing	A scale factor, k, specified by a P edit descriptor in the form kP, must be in the range -38 to +38 for complex or reals, -308 to +308 for double-precision.
13-9	22	13.5.9.1 Integer Editing	In the Iw and Iw.m edit descriptors, w and m are unlimited.
13-10	5	13.5.9.2.1 F Editing	In the Fw.d edit descriptor, w and d are unlimited.
13-10	42	13.5.9.2.2 E and D Editing	In the Ew.d, Dw.d, and Ew.dEe edit descriptors, w.d. and e are unlimited.
13-11	35	13.9.5.2.3 G Editing	In the Gw.d and Ew.dEe edit descriptors, w.d. and e are unlimited.
13-12	33	13.5.10 L Editing	In the Lw edit descriptor, w is unlimited.
13-12	49	13.5.11 A Editing	In the Aw edit descriptor, w is unlimited.
13-15	50	13.6.2 List-Directed Output	Integer output constants are produced with the effect of an Iw edit descriptor, where w is chosen appropriately to the number being printed.

13-15	53	13.6.2 List-Directed Output	d_1 = -5; d_2 = 6; w, d, and e are chosen appropriately to the number being printed.

Table 7. Representation of Various Data Types.

Data type	*Range*
Single-precision	˜1.175 x 10^{-38} to 3.4 x 10^{38}, accurate to 24 bits (˜7 decimal digits)
Double-precision	˜2.2 x 10^{-308} to 1.8 x 10^{308}, accurate to 53 bits (˜16 decimal digits)
Integer	Arbitrary-precision *(bignums)*
Character	8 bits

Table 8. FORTRAN File Types on Different Systems.

System	*File type*
Genera	.fortran, .for, .ftn
UNIX4.2	.f, .for, .ftn, .fortran
UNIX4.1	.f
MULTICS	.fortran
All others	.for

0.0.171. Predicates

Press the following keys to format input from an interactive stream:

Function	*Implementation*
EOF	`<FUNCTION> <END>`
EOLN	`<RETURN>`, `<LINE>`, or `<END>`

open **Statement**

0.0.172. recl Field

The length of the `recl` field of the `open` statement is processor-dependent. The computation of this value in the Symbolics software environment is complicated by the requirement to store tag information in the record, as well as data.

Record lengths are measured in 32-bit words. Words written to the file are either *data* or *tag* words.

Data type	*No. of words*
Integer, real, logical	1 data word
Double-precision, complex	2 data words
Character block	**ceiling** of (length, 4) data words
Sequence of items of the same type	1 tag word

Example: For a file of records containing two items of type integer and one of type double-precision, the `recl` field = 6 words.

TAG	integer	integer	TAG	double-precision	
1	1	1	1	2	= 6 words

A file of records containing items all of the same type has the minimum `recl` field. Then only a single word of tag information is required. If *n* data words were written, the size is *n + 1*.

A file of records containing items of alternating types has the maximum `recl` field. Then the size is somewhere between *2n* and *3n*, depending on the mixture of single- and double-word types of the data words.

Compiling Symbolics FORTRAN Programs

Features of the Symbolics FORTRAN Compiler

0.0.173. Lisp Code Is Produced

The Symbolics FORTRAN compiler produces Lisp code, which is compiled into "machine code" by the standard Lisp compiler. Since Lisp is essentially the machine language of Symbolics computers, no loss of performance occurs.

The Lisp produced by the compiler is not intended to be examined or maintained as such; rather, it is just the compiler's intermediate object language, incidental to producing machine code. You are never required to think in terms of either Lisp or the machine code, since the Symbolic debugger works on the FORTRAN source code level.

0.0.174. Benefits of Lisp Code

You derive several important advantages from compiling into Lisp. One is that you can call Lisp numeric functions from FORTRAN; indeed, many of the FORTRAN library subroutines, such as `sin` and `cos`, are in fact those built into the Lisp system.

Another benefit is that you can choose to implement a particular routine in either Lisp or FORTRAN, whichever language is more suitable for your ends. (The procedure for calling Lisp functions from FORTRAN is described elsewhere: See the section "`lispfunction`: Type Declaration Statement".)

0.0.175. Compilation Is Incremental

The software environment allows *incremental compilation* in the editor buffer; that is, you can compile selected routines rather than whole programs. The compiler maintains a description of the declaration scope of every routine ever compiled. This permits you to compile any routine whose calling routine is compiled, usually without recompiling the calling routine.

This feature encourages frequent and thorough debugging and greatly speeds up the program development process.

0.0.176. Compiler Options

Symbolics FORTRAN supports three compiler options. One initializes program data, one allows you to assign ordinal numbers to unnamed main programs, and one lets you control whether sequence numbers are recognized during program compilation.

You can exercise these options by setting the initialization variables that control their behavior. Set these variables either in your init file or at a Lisp Listener. For more information:

- See the section "Compiler Option: Assigning Names to Unnamed FORTRAN Main Programs".

- See the section "Compiler Option: Initializing FORTRAN Program Data".

- See the section "Compiler Option: Recognizing Card Sequence Numbers".

In addition, when you are compiling files intended to be part of a run-time system, you must set the variable **cts:*compile-for-run-time-only***. By filtering out information needed to support debugging and incremental compilation, this variable reduces the size of the binary file produced. For further information, see the section "FORTRAN Applications with Run-time Systems".

0.0.177. Compiler Warnings

The FORTRAN compiler produces diagnostics whenever a program violates the rules for a legal program, as specified in the ANSI Standard. In this case, the screen displays compiler warnings, which generally provide useful information regarding the cause and location of an error.

Sometimes compilation produces a great many compiler warnings, too many for you to remember. Fortunately, the warnings are stored in an internal database, whose contents you can inspect and manipulate through several editor commands.

For example, `m-X` Edit Compiler Warnings splits the editor window into two frames: The upper frame displays a warning message; in the lower frame the cursor is positioned at the instance of source code that produced the message displayed in the upper frame.

Recompiling the corrected code deletes the old warnings and inserts any new warnings. Correcting all errors and recompiling the code empties the database.

0.0.178. Compile File

Invoking Compile File at the Command Processor compiles files of FORTRAN routines.

0.0.179. Zmacs Compiler Commands

Several Zmacs commands are available for compiling FORTRAN routines and resolving compiler warnings:

`c-sh-C`
: Compiles to memory the currently defined *region*, a contiguous delimited section of text in the editor buffer. If none is defined, it compiles the routine nearest the cursor. This command does not take a numeric argument.

`m-X` Compile And Execute Fortran Program
: Checks to see that the cursor is positioned near a valid FORTRAN program and then compiles and executes the program, without run-time options and with predefined files input and output bound to the editor typeout window.

`m-X` Compile Buffer
: Compiles the entire buffer to memory. With a numeric argument, it compiles from *point* (the cursor position) to the end of the buffer. You can use this feature for resuming compilation when a previous attempt fails.

`m-X` Compile Changed Definitions of Buffer
: Compiles to memory any FORTRAN routines in the buffer that have changed. With a numeric argument, it queries about whether to compile each changed routine.

`m-X` Compile File
: Compiles a file, offering to save it first if the buffer is modified. It prompts for a file name in the minibuffer, using the file associated with the current

buffer as the default. The command writes a compiled-code file to disk but does not create object code in memory.

m-X Compile Region
: Compiles to memory the currently defined region. If none is defined, it compiles the routine nearest the cursor. Same as c-sh-C.

m-X Compiler Warnings
: Places all pending compiler warnings in a buffer and selects that buffer. It loads the compiler warnings database into a buffer called *Compiler-Warnings-n*, creating that buffer if it does not exist.

m-X Edit Compiler Warnings
: Edits some or all routines whose compilation caused a warning message. It queries you, for each file mentioned in the compiler warnings database, whether you want to edit the warnings for the routines in that file. It splits the screen, placing the warning message in the top window and the source code whose compilation caused the error in the bottom window. Use c-. to move to the next pair of warning and source code.

m-X Load Compiler Warnings
: Loads a file containing compiler warning messages into the warnings database. It prompts you for the name of a file containing the printed representation of the warnings.

Compiling Small and Large FORTRAN Programs

0.0.180. Small Programs

One method of compilation, appropriate for small programs, is to read the source into an editor buffer and use c-sh-C to compile the routines to memory.

c-sh-C compiles the current *region* (a contiguous region of text defined by the user) or the routine nearest the cursor, if no region is defined. For each FORTRAN routine, this creates a function in the machine's virtual memory but does *not* create a file version of the object code.

Note that a region can be as large as the entire buffer. After reading in the source file, press c-X H to mark the entire buffer as a region. Then use c-sh-C as usual.

After successful compilation, the routines are available for execution; no separate linking and loading are necessary. Typically, programmers use the editor compilation facility during the debugging cycle to compile code changes quickly, rerun the program, and do subsequent testing.

Another method appropriate for programs contained in one file is to compile the file to disk. The Zmacs command m-X Compile File compiles a file and places the output in a Lisp compiled-code (binary) file.

The most essential difference between compiling a source file and compiling the same code in an editor buffer is this: When you compile a file, none of the FOR-

TRAN routines is defined at compile time. Instead the compiler puts instructions into the compiled-code file causing definition to occur at load time. Load the compiled-code file into memory with `m-X` Load File.

0.0.181. Large Programs

The other method of compilation is appropriate for a large FORTRAN program, especially if it consists of several files. This method requires two steps.

1. Declare a group of FORTRAN files as a system. A *system* is a set of files and a set of rules defining the relations among these files; together these files and rules constitute a complete program.
2. Use the Load System or Compile System command to compile the program's routines and user libraries, load the object code into virtual memory, or both.

This method results in the creation of object files in the file system, which you can then load at your discretion.

The facilities for defining a FORTRAN system are described elsewhere: See the section "Large FORTRAN Programs".

Compiler Option: Assigning Names to Unnamed FORTRAN Main Programs

0.0.182. Problem

The Standard allows the use of unnamed main programs. However, in the current version of Symbolics FORTRAN, the compilation of several unnamed main programs during one work session might cause problems.

The compiler assigns an unnamed main program (that is, one without a `program` statement) the name `.main.`. You can run the most recently compiled main program with its assigned name:

```
(f77:execute .main.)
```

However, a problem arises when you compile another unnamed program, which the compiler also calls `.main.`; the software environment warns about what it thinks is a redefinition of a previously compiled function, in actuality another unnamed program called `.main.`.

0.0.183. Solution

To avoid this problem, you can either place the programs in different packages or set the initialization variable that controls this behavior — **f77:*use-main-sequence-numbers***.

f77:*use-main-sequence-numbers* *Variable*

Controls whether the compiler assigns a unique "name" to an unnamed main program; the "name" distinguishes one compiled unnamed program from another.

Value	*Meaning*
t	The compiler assigns each unnamed FORTRAN main program with a name in the form of `.main-`*n*`.`, where *n* is an ordinal number. Example: `.MAIN-7.` is the name of the unnamed main program to be compiled.
nil	The compiler does not assign a unique name to each unnamed FORTRAN main program. **nil** is the default.

To set this variable in your init file, type:

```
(login-forms
  (setq f77:*use-main-sequence-numbers* t))
```

However, unless (1) the FORTRAN system is stored on the disk and is therefore accessible when you log in or (2) your init file loads FORTRAN, do *not* add the above Lisp expression to your init file.

Alternatively, you can set the variable at a Lisp Listener at any time after loading the FORTRAN system into memory but before compiling the program. Type:

```
(setq f77:*use-main-sequence-numbers* t)
```

To turn off this option, reset the variable. Type to a Lisp Listener:

```
(setq f77:*use-main-sequence-numbers* nil)
```

0.0.184. Invoking the Last Program

If you want to execute the last unnamed main program but cannot remember its number, use **f77:run-last-main**.

f77:run-last-main *Function*

When you are using the ordinal number facility, **f77:run-last-main** invokes the last compiled unnamed main program.

```
(f77:run-last-main)
```

0.0.185. Disadvantage

One disadvantage of using the ordinal number facility is that it causes problems for the editor, which determines the name of the program at the time the file is read in. It has no knowledge of the ordinal numbers assigned as programs are compiled. As a result, certain commands, like `m-X` Edit Compiler Warnings, do not work correctly.

Compiler Option: Initializing FORTRAN Program Data

0.0.186. Problem

Normally, Symbolics FORTRAN sets variables to the Lisp object Undefined when they are not explicitly initialized by FORTRAN data statements. The hardware flags an error if any program attempts to manipulate this value as a number. However, a problem arises if some of your programs actually depend on the absence of checking for uninitialized values.

0.0.187. How to Set the Option

To avoid this potential problem, set the initialization variable that controls this behavior — **f77:*ftn-init-to-zero***.

f77:*ftn-init-to-zero* *Variable*

Controls whether local FORTRAN variables have an initial value when they are compiled.

Value	*Meaning*
t	Sets the initial value of all local variables to zero.
nil	Sets local variables not explicitly initialized by FORTRAN data statements to the Lisp object Undefined. **nil** is the default.

Example: To set the form in your init file, type:

```
(login-forms
  (setq f77:*ftn-init-to-zero* t))
```

However, unless (1) the FORTRAN system is stored on the disk and therefore accessible when you log in or (2) your init file loads FORTRAN, do *not* add the above Lisp expression to your init file.

Alternatively, you can set the variable at a Lisp Listener at any time after loading the FORTRAN system to memory but before compiling the program. Type:

```
(setq f77:*ftn-init-to-zero* t)
```

To turn off this option, reset the variable. Type to a Lisp Listener:

```
(setq f77:*ftn-init-to-zero* nil)
```

Note on Global Variables and Arrays

Note that the **f77:*ftn-init-to-zero*** option controls initialization of local variables by the compiler. To control initialization of arrays, common variables, and other global variables, use the **:init-to-zero** option to **f77:execute**.

Compiler Option: Recognizing Card Sequence Numbers

The following variable is available for use with the FORTRAN compiler.

f77:*ftn-card-sequence-numbers-legal* *Variable*

Controls whether it is valid for characters to exist beyond column 72. The default value is **t**. Set the value to **nil** to have parse errors returned if characters exist beyond column 72.

The generated code is not affected by this variable.

Large FORTRAN Programs

What is a FORTRAN System?

0.0.188. Introduction

You can split large programs into several files for the sake of modularity, organization, and ease of editing, searching, and compiling/recompiling small pieces of code. The main disadvantage of this approach is the extra time spent in loading individual files and keeping track of which files you have edited but not recompiled.

To overcome the drawbacks inherent in manipulating small chunks of a large program, Symbolics FORTRAN provides a way to define a collection of FORTRAN routines, possibly spanning several files, as a FORTRAN *system*.

0.0.189. Definition

In general, a system is a set of files and a set of characteristics that describes the files; together these files and characteristics constitute a complete program that you can manipulate as a unit. Use the Lisp special operator **f77:def-program** to declare a group of FORTRAN files (or FORTRAN and Lisp files) a FORTRAN system. The declaration allows you to specify the files and libraries composing the system as well as the desired properties of the system, such as the package into which the object code compiles.

f77:def-program is analogous to the **defsystem** function used in Lisp. In fact, the facilities provided for FORTRAN are simply Lisp macros that expand into **defsystem** invocations.

0.0.190. Benefits

The FORTRAN system offers several benefits:

- Compiled code is stored on disk.
- You can compile and load all the system files to your environment *at one time*, and in accordance with the properties specified in the system definition. More-

over, you can compile only those files that need compiling, that is, only those source files you have edited.

- You can call defined FORTRAN libraries.
- You can use certain system utilities, like patching, that operate only on defined systems.

0.0.191. Procedure

The procedure below summarizes all the steps necessary for declaring, compiling, and loading a FORTRAN system. You write all package, library and system declarations in the same file, called the *system declaration file*, in the order specified below. The system declaration file must have a Lisp file type and should be in the **cl-user** package.

1. Make a package declaration for the files composing the system, using **f77:package-declare-with-no-superiors**. Alternatively, you can use the predeclared package **ftn-user**, in which case you do not need to make a package declaration. Also add the name of the package to the attribute list of the individual files in the system. See the section "Declaring a FORTRAN Package".
2. Make a package declaration for the user library or libraries composing the system, if any, using **f77:package-declare-with-no-superiors**. You can declare different packages for each library. Alternatively, you can use the predeclared package **ftn-user**.
3. Make a package declaration for the FORTRAN system defined in step 5, if different from the package declared for the individual files. Use **f77:package-declare-with-no-superiors**. Typically, the system package is the same as the package of the individual files.
4. Define a set of routines as a FORTRAN user library, using **f77:def-library**. You have to previously define any libraries cited in the definition of another library as FORTRAN libraries. See the section "Declaring a FORTRAN Library".
5. Make a system declaration for all files and libraries, using **f77:def-program**. The system declaration includes the name of the files in the system, the package into which the files are compiled, and the order of compilation and loading of files. See the section "Declaring a FORTRAN System".
6. Load the system declaration file manually (via m-X Load File) or create a system site file to load the declaration. See the section "Loading the System Definition".
7. Compile and load the FORTRAN system defined in step 5, using the Command Processor. See the section "Compiling and Loading a FORTRAN System".

Declaring a FORTRAN Package

0.0.192. Introduction

Once you load FORTRAN and Lisp routines into the software environment, they remain there until they are replaced by recompilation, explicitly removed, or until you cold boot the machine. Since you might have two large FORTRAN programs having the same name, Genera must have a means of distinguishing between them. Genera provides a mechanism for separating like-named programs by assigning each its own distinct context, or *namespace*. The namespace is called the package.

The use of packages prevents naming conflicts; two different FORTRAN programs can have the same name only if each exists in its own package. For example, you must place two FORTRAN programs the name `primes` by the compiler in two different packages.

You can avail yourself of the built-in package provided by Symbolics Fortran, or you can create your own, using the special form **f77:package-declare-with-no-superiors**. The facility for declaring FORTRAN systems allows you to specify only an *existing* package in the definition; that is, you must previously define and compile the package. All files in the system are compiled in the package specified in the system declaration.

f77:package-declare-with-no-superiors *name* &body *defpackage-options*
Special Form

Creates packages appropriate for FORTRAN code.

name is a symbol that is the name of your package, for example, `matrix`.

```
(f77:package-declare-with-no-superiors matrix)
```

defpackage-options are optional keyword arguments. For most users, it is not necessary to include these keywords in the package declaration. However, if you decide to use them, please read the conceptual material on packages in the Genera documentation Set. See the section "Packages".

:nicknames *[name name]...*

(:nicknames *name name***...)** for **defpackage**
:nicknames '(*name name...***)** for **make-package**
The package is given these nicknames, in addition to its primary name.

(:prefix-name *name***)**

(:prefix-name *name***)** for **defpackage**

:prefix-name *name* for **make-package**

This name is used when printing a qualified name for a symbol in this package. You should make the specified name one of the nicknames of the package or its primary name. If you do not specify **:prefix-name**, it defaults to the shortest of the package's names (the primary name plus the nicknames).

(:size *number***)**

(:size *number***)** for **defpackage**

:size *number* for **make-package**

The number of symbols expected in the package. This controls the initial size of the package's hash table. You can make the **:size** specification an underestimate; the hash table is expanded as necessary.

How to Make a FORTRAN Package Declaration

You can make a package declaration in either of two ways:

- You can type the **package-declare-with-no-superiors** form to a Lisp Listener, in which case the declaration lasts only as long as you are logged in. You must create the package every time you log in.

- You can type the **package-declare-with-no-superiors** form in a Zmacs buffer whose mode is Lisp. (To have the editor set the mode automatically, create the file with the correct Lisp file type for your host system.)

  If you intend to specify the package name in a system declaration, place the package declaration in a file, specifically, the same file in which you enter your FORTRAN system declaration (called the *system declaration file*). The file name must have a Lisp file type and is typically compiled to the **cl-user** package.

  Compile the package declaration.

Predeclared FORTRAN Packages

Symbolics FORTRAN recognizes two packages that you do not need to explicitly define.

- **cl-user** is the default Symbolics Genera package.

  Caution: cl-user inherits all the symbols in the **global** package, which contains the basic symbols of the Lisp language. As a result, FORTRAN routines or variables in the **cl-user** package can conflict with existing Lisp functions.

- **ftn-user** is the default package of Symbolics FORTRAN. Since **ftn-user** was originally declared using **f77:package-declare-with-no-superiors**, no name conflicts can occur for programs in this package.

How to Assign a FORTRAN Package

You must add the package name to the file's attribute list. This permits editor-based compilation of routines in the file, without reference to a system declaration or the prevailing package.

To change the package name in the attribute list, use `m-X` Set Package and type the package name. The command offers to create the package if it does not exist. Alternatively, you can manually enter the package name in the attribute list by typing `; Package:` and the name of an existing package or a package you have previously defined with **f77:package-declare-with-no-superiors**. Type `m-X` Reparse Attribute List to have the change take effect.

If you plan on including the file as part of a system declaration, we recommend that you specify the package as the value of the **:package** option in the system declaration. The files you list in the definition become associated with the specified package. Note that you must compile all FORTRAN files of a system declaration into the same package or they will not correctly reference each other.

Declaring a FORTRAN Library

0.0.193. Libraries in Genera

FORTRAN user libraries under Genera have the following characteristics:

- Library routines must be declared as a FORTRAN library.
- Main programs can call routines in a defined library in several ways:
  - Use the **:libraries** keyword to **f77:execute** at run time.
  - Define the library as part of a FORTRAN system and use the **:system** keyword to **f77:execute** at run time.
  - Define the library as part of a FORTRAN system and use the **:libraries** keyword to **f77:execute** at run time.
- Only one copy of the library resides in memory. This contrasts with conventional computing environments, where many copies of a library live in memory if called by several main programs.
- Many FORTRAN routines can use the same library.
- You do not have to put user libraries in the same package as the FORTRAN routines calling them.

Why? Because at run time, during construction of the main program's call tree, a search is made for any routines not present in the package into which the main program is loaded. This search examines all libraries in the order in which they were declared in the system definition, until the desired library routine is found. The library is then made directly accessible from the package of the FORTRAN system.

- Whenever you recompile a library routine independent of recompilation of a FORTRAN system, the FORTRAN-system package always has access to the updated object code.

You must declare a group of routines as a FORTRAN library with **f77:def-library** before you can include that library in the declaration of a FORTRAN system.

f77:def-library *name* &rest *options* *Special Form*

Defines a FORTRAN library, where *name* is a symbol that is the name of the library. *options* refers to the valid keyword options.

Each of the following keywords to **f77:def-library** is allowed:

Keyword	*Meaning*
:package	Specifies the name of an existing package, if any. The package paired to the **:package** keyword overrides the packages associated with the individual files or libraries of the system.
:files	Specifies one or more file specification strings. The strings correspond to the names of files composing the library.
:libraries	Specifies one or more previously defined FORTRAN libraries, if any, that you have to include in this library declaration. Libraries can themselves depend on the existence of other libraries.

In addition to the above keywords, you can use any of the valid keywords for **defsystem**, the analogous function for Lisp. See the section "**defsystem** Options".

Example: Suppose you want to create a library of matrix routines (call it `matrix2`) in a file whose pathname is s:>libraries>matrix2.ftn, and have the code compiled into a package called `matrix-pkg`. In addition, `matrix2` depends on an existing library, `matrix1`. To define `matrix2` as a library:

```
(f77:def-library matrix2
  (:default-package "matrix-pkg")
  (:files "s:>libraries>matrix2")
  (:libraries matrix1))
```

For instructions on how to make a library declaration, see the section "Declaring a FORTRAN System".

Declaring a FORTRAN System

You should read background material before attempting to declare your own FORTRAN system. See the section "System Construction Tool".

f77:def-program *name options* &rest *modules-etc* *Special Form*

Declares a set of files as a FORTRAN system, where:

name is the name you have chosen for your FORTRAN system, not necessarily the name of a main program.

options are the valid keywords for **defsystem**, the analogous function for Lisp. See the section "Defining a System". Note that FORTRAN systems can consist of just FORTRAN code, or FORTRAN code and Lisp code; however, the Lisp and FORTRAN files must not depend on one another.

modules-etc are module specifications (Lisp, FORTRAN, or Pascal type modules) and/or the **:files** and **:libraries** keywords, described below.

Option	*Meaning*
:files	Specifies one or more FORTRAN files that composing the system.
:libraries	Specifies one or more previously defined FORTRAN libraries, if any, that you wish to include in the system declaration.

0.0.194. Example 1: A FORTRAN System

Suppose you have a FORTRAN program, `plot`, with three files — onedplot, twodplot, and axislabels — residing on directory f:>fred>plot>. In addition, you want (1) the object code compiled into an existing package called `plot` and (2) one user library, `graphics`, incorporated into the defined system.

```
(f77:def-program plot
        (:full-name "Plot System"
         :default-pathname "f:>fred>plot>"
         :default-package "plot")
   (:files "onedplot" "twodplot" "axislabels")
   (:libraries graphics))
```

0.0.195. Example 2: A FORTRAN and Lisp System

System declarations can mix FORTRAN and Lisp code. Note carefully that the Lisp code can in no way depend on the FORTRAN code, and vice versa. In the example below, the value of the **:default-package** keyword is a FORTRAN package. Assume that the Lisp code compiles into the **cl-user** package, which is specified in the attribute list of each Lisp file.

```
(f77:def-program lisp-and-fortran
        (:pretty-name "Registration System"
         :default-package "registrar"
         :default-pathname "f:>sr>registrar>")
   (:module definitions ("vars" "defs")
        (:package cl-user) (:type :Lisp))
   (:serial (:parallel definitions "macros")
            "display")
   (:files "dave" "bob" "fred")
   (:libraries foo))
```

The **:serial** and **:parallel** options provide an abbreviated syntax for defining what modules (files or sets of Lisp files) compose the system and how these modules depend on one another. In the example above, "display" depends on the prior compilation/loading of definitions and "macros", but in no particular order. ("Definitions" is a module containing two Lisp files in package **cl-user**, vars and defs). These options apply only to Lisp code within the **f77:def-program** form. (See the section "Short-form Module Specifications".)

0.0.196. Example 3: A FORTRAN, Lisp, and Pascal System

You can use **defsystem** to create a system combining FORTRAN, Pascal, and Lisp code. Note in the example below that you must specify the files of each language separately in their own **:module** declaration.

:module, considered the long-form module specifier, is used only when more complicated dependency relationships exist among modules or when your system contains modules and packages that are not of the default type for the system. (See the section "Long-form Module Specifications" in *Program Development Utilities*.)

For example, the **:module** specifications are required here because the modules are not of the default type, which is Lisp. Lisp is the default type when the options list does not specify one for the system; the options list below specifies only a default pathname.

When you do not specify any compile or load dependencies, each module compiles and loads in turn. Here, FORTRAN compiles and loads first, then Pascal, then Lisp.

```
(defsystem fortran-pascal-lisp-demo
  (:default-pathname "C:>sr>")
  ;; needs a module declaration because of non-default
  ;; type and package
  (:module fortran-part ("fpl-ftn")
           (:package ftn-user)
           (:type :fortran))
  (:module pascal-part ("fpl-pascal")
           (:package pascal-user)
           (:type :pascal))
  (:module lisp-part ("fpl-lisp")
           (:package cl-user)))      ; LISP is the default type
```

Compiling and Loading a FORTRAN System

0.0.197. Command Processor

You use the Compile System and Load System comands to compile and load your FORTRAN system. These commands load the system declaration file if:

1. you have created a system site file, or

2. you have a **si:set-system-source-file** form in your init file.

Otherwise, use the Load File command for loading the system declaration file:

```
Load File analysis:analysis;finite-element-analysis-sysdcl.lisp
```

To load and compile the system files of `finite-element-analysis`, type:

```
Compile System finite-element-analysis :Load :newly-compiled
```

FORTRAN Applications with Run-time Systems

Symbolics FORTRAN supports features enabling you to build and distribute minimally sized applications including the FORTRAN run-time system. Applications including this system can run in environments that are not running the FORTRAN development system. Customers who distribute an application with the FORTRAN run-time system **must** sign a *Sublicense Addendum to the Terms*. See the section "Sublicense Addendum for Symbolics FORTRAN77".

0.0.199. Components of the Run-time System

A run-time system (as opposed to the development system) is made up of the minimal subset of the FORTRAN development system software required to load and execute a program. From a user's perspective, it contains the library routines defined for the language, the loader, and the function that initiateing execution. The following functionality, normally present in the development system, is absent in the run-time system:

- Language-specific Zmacs Editor Mode

- Compiler

- Language-specific Source Level Debugger

The FORTRAN run-time system is called *Fortran-Runtime*.

0.0.200. Creating Applications

The normal procedure to develop an application and deliver it with the FORTRAN run-time system follows these steps:

1. You develop the application software in a development environment.

2. During program compilation you can set a global variable that filters out debugging and other information from binary files. Exercising this option creates smaller bin files.

3. When writing a system declaration, you include a run-time component system as part of the system declaration, or *sysdcl*.

Minimizing the Size of Compiled Files

During a normal compilation, the compiler produces information that supports debugging and incremental compilation. This information is normally written out to the bin file, a binary file identified by the file extension *bin*. You can exclude this information from the bin file by setting the special variable **cts:*compile-for-run-time-only*** to the Lisp boolean **t.** Doing so minimizes the size of the binary files and can therefore aid in producing applications of minimal size.

By convention, binary files produced in this manner are referred to as *rto* bins (but assigned the file extension .bin). Using *rto* binary files limits your ability to debug and compile source code, so use this facility judiciously. Use of this facility does not change the generated code. The section "Program Configurations: Development System and Run-time System Options for FORTRAN" specifies the capabilities of *rto* binary files.

Incorporating the Run-time System Into a FORTRAN Application

You should package the run-time system as a dependent component system of the application. When defining such a packaged system, the packaged system definition must cause the run-time system to load before any of the application program loads. For proper loading, specify the appropriate `:serial` dependency.

The following example illustrates how you can package an application named *a1*. Note that you must include *a1* as a component system (with accompanying separate *sysdcl* file) and not just as a separate subsystem.

```
(defsystem a1
    (:default-pathname "foo:bar;"
     :distribute-binaries t
     :default-module-type :FORTRAN)
  (:serial "f1.fortran" "f2.fortran"))
```

```
(defsystem packaged-a1
    (:default-pathname "packaged-foo:bar;"
     :distribute-binaries t)
  (:module Fortran-runtime "Fortran-runtime" (:type :system))
  (:module a1 "a1" (:type :system))
  (:serial Fortran-runtime a1))
```

You can use the distribution software to distribute the packaged software. See the sections:

"Distribute Systems Command"
"Distribute Systems Frame"
"Restore Distribution Command"
"Restore Distribution Frame"

Program Configurations: Development System and Run-time System Options

Given the capabilities of a run-time system, and the ability to produce *rto* bins, a program can be in one of the configurations obtained by the following cross product:

(normal bin, rto bin) X (development system, run-time system)

The (normal bin, development system) configuration is the usual configuration and the one that makes the full functionality of the development system available. Other configurations limit the functionality in various ways.

The following table describes the properties of each possible configuration.

	Development System		*Run-time System*	
Normal Bin	Incremental Compilation:	Yes	Incremental Compilation:	No
	Batch compilation:	Yes	Batch compilation:	No
	Language-specific debugging:	Yes	Language-specific debugging:	No
Rto Bin	Incremental Compilation:	*	Incremental Compilation:	No
	Batch compilation:	*	Batch compilation:	No
	Language-specific debugging:	No	Language-specific debugging:	No

*Incremental compilation is possible, after all references external to the unit you are incrementally compiling are compiled. For FORTRAN this means that you must compile a file or buffer before you can compile an individual function within it.

0.0.198. Purpose of Configurations

Normal bin, Development system
: This is the normal configuration for software development.

Normal bin, Run-time system
: This configuration is advantageous when you are actively developing software and simultaneously using it in a run-time system.

Rto bin, Run-time system
: This is the desired configuration for software that is released, and of minimal size.

Rto bin, Development system
: This is not a recommended configuration. You should re-create normal *bin* files if you plan to do any debugging or development work with these files.

Executing FORTRAN Programs

Background

0.0.201. No Link-and-Load Step

Symbolics computers are a large-scale virtual memory, single-user machines. Routines compiled by Symbolics FORTRAN remain in the environment until replaced by recompilation or until you cold boot the machine, that is, until you load a fresh version of Genera.

The *executable module*, as most FORTRAN programmers understand the term, does not exist. Rather, once routines are brought into virtual memory by compilation in an editor buffer or by "making" a FORTRAN system, they are immediately executable; thus, Symbolics FORTRAN requires no separate link-and-load step.

The Genera environment supports incremental compilation; consequently, the absence of the link step is of great significance when making small changes to large FORTRAN programs, since the link step in traditional computing environments is time-consuming.

0.0.202. Call Tree Is Set Up Before Run

When you first run a main program, it sets up a data structure representing the program's call tree (the set of all FORTRAN routines called); all variables initialized by FORTRAN data statements are set up *before* the program begins execution. This differs from most implementations, in which data initialization takes place in the link-and-load step.

0.0.203. Call Tree Exploration

The following example illustrates a concern when setting up a program's call tree before execution. It shows some FORTRAN code and a Lisp function calling a FORTRAN subroutine from this code.

The call tree is explored at execution time to determine the space requirements of a FORTRAN subprogram. In the following example, since `ftn-user::bar` is not called directly from a FORTRAN subprogram, it is not encountered during the call tree exploration of `ftn-user::test`. You can solve this problem by putting `bar` in an *external* statement in `ftn-user::test`. You can also use the more temporary solution of calling **f77:execute** or the CP command Execute FORTRAN, passing it `ftn-user::bar` in the `:additional-externals` list. Note that `ftn-user::boo` is found during the call tree exploration of `ftn-user::bar`.

FORTRAN code:

```
C  -*- Mode: FORTRAN; Package: FTN-USER -*-

         program test
         lispfunction foo 'cl-user::foo'
         call foo
         end

         subroutine bar
         implicit none
         integer i
         call boo (i)
         end

         subroutine boo (i)
         implicit none
         integer i
         print *,i
         end
```

Lisp code:

```
(defun cl-user::foo () (funcall 'ftn-user::bar))
```

How to Run a FORTRAN Main Program

0.0.204. Invoke Only a Main Program

Since only a FORTRAN main program can initialize input/output facilities and program data, it is not valid to invoke a FORTRAN routine *except* in the dynamic scope of a main program.

All Symbolics FORTRAN programs are compiled into Lisp object code. Invoke these programs — once they are in memory — in one of the following ways:

- Zmacs commands: `m-X` Execute Fortran Program or `m-X` Compile and Execute FORTRAN Program
- Command Processor command: Execute FORTRAN
- Lisp function: **f77:execute**

0.0.205. Zmacs Commands

To run your code in a Zmacs buffer, compile the program to virtual memory using `c-sh-C` or a related command. Placing the cursor near the program you want to run, issue `m-X` Execute Fortran Program. The command checks to see that the cursor is near a valid, compiled FORTRAN program and then executes the program. Execute FORTRAN Program does not accept any of the run-time options accepted by **f77:execute**. The predefined file input and output are bound to the editor typeout window.

`m-X` Compile and Execute FORTRAN Program performs identically to Execute Fortran Program, except that it first compiles the program to virtual memory before executing it.

0.0.206. Command Processor Command

The Execute FORTRAN command runs a valid, compiled FORTRAN program. The command takes a FORTRAN main program name and accepts the same set of keywords as **f77:execute**. See the section "FORTRAN Main Program Options".

Note, however, that command keywords use underscores, not hyphens. For example, the **:init-to-zero** option for **f77:execute** is rendered as the :Init_to_zero keyword to Execute Fortran.

Example: `(f77:execute ftn-user:mean :init-to-zero t)` is invoked from the Command Processor as:

```
Execute Fortran ftn-user:mean :Init to zero yes
```

0.0.207. Lisp Function

f77:execute *main-program-name* &rest *options* *Function*

Runs a FORTRAN program, where *name* is the name of the program. **f77:xqt** is a synonym for **f77:execute**.

The FORTRAN run-time system supports the options **:additional-externals,** **:blockdata**, **:extra-character-space**, **:extra-number-space**, **:init-to-zero**, **:libraries**, **:pathname-default**, **:reload**, **:save-environment**, **:system**, **:trap-underflow**, and **:units**.

See the section "FORTRAN Main Program Options".

0.0.208. Example 1

To run a main program `ftn-user:convert`, go to a Lisp Listener and type:

```
(f77:execute ftn-user:convert)
```

0.0.209. Example 2

To run a main program `ftn-user:convert`, which reads a single line of parameters from unit 5, go to a Lisp Listener and type:

```
(f77:execute ftn-user:convert)1 2 3 RETURN
```

0.0.210. Lisp Listener Package

Run a FORTRAN main program from a Lisp Listener. But be careful: An error results if you attempt to invoke a main program when its package differs from that of the current Lisp Listener and you do not specify the package of the main program. For example, assuming that the Lisp Listener package is **cl-user** and the main program package is **ftn-user**, the following invocation of `convert` signals an error:

```
    (f77:execute convert)
```

The **cl-user** package does not recognize `convert` as a FORTRAN main program.

The current package of the Lisp Listener always displays on the status line. In general the current package is determined by these guidelines:

- At a top-level Lisp Listener, the package is the default **cl-user**, unless you explicitly change it.

- In the editor, the Lisp Listener package corresponds to that of the Zmacs buffer.

  Example: If the cursor is in the middle window of a three-window screen and you invoke a Lisp Listener, the package of the Lisp Listener is the same as that of the middle window.

If your main program resides in a different package than that of the Lisp Listener, specify the package name at run time, for example:

```
    (f77:execute ftn-user:quadratic)
```

Note: Be careful when using the **zl:pkg-goto** function to change to the **ftn-user** package, or any package declared with no superiors; specifically note that even **nil** requires a package prefix.

FORTRAN Main Program Options

0.0.211. Introduction

All FORTRAN main program invocations accept several pairs of keywords and values. These keywords are recognized:

- **:additional-externals**
- **:blockdata**
- **:extra-character-space**
- **:extra-number-space**
- **:init-to-zero**
- **:libraries**
- **:pathname-default**
- **:reload**
- **:save-environment**
- **:system**
- **:trap-underflow**
- **:units**

Note that using the **:init-to-zero** option resets all real variables to the integer zero.

0.0.212. Format

The options are specified as keyword-value pairs:

(**F77:execute** *main-program-name option value* ...)

Example: Invoking the main program `mean` with multiple options looks like this.

```
(f77:execute ftn-user:mean :units ((3 "f:>tc>abc.ftn"))
                           :save-environment :copy)
```

:additional-externals

The **:additional-externals** option enables you to specify a list of FORTRAN subprograms you are adding to the call tree exploration, the program initialization step. It is especially useful if you intend to use a FORTRAN library from Lisp.

These FORTRAN subprograms are those called directly by a Lisp function and never referenced in a FORTRAN EXTERNAL statement or called directly or indirectly by the FORTRAN program you are executing.

For further information, see the section "Call Trees in Symbolics FORTRAN 77".

:blockdata

The **:blockdata** option enables you to specify all block data subprograms called by the main program. (See the Standard, chapter 16.) This contrasts with most systems, in which block data subprograms are specifically listed at link-and-load time.

Note: The compiler assigns the name `.blockdata.` to an unnamed block data subroutine.

Value	*Meaning*
block-name	Specifies a list of names of block data subroutines.

To run a main program `taxes` with block data subroutines `schedulex`, `scheduley`, and `schedulez`, type:

```
(f77:execute ftn-user:taxes blockdata (schedulex scheduley schedulez))
```

:extra-character-space

The **:extra-character-space** option applies only when you use the **f77:with-fortran-character-data** macro in your program. This option specifies the amount of extra space to reserve for allocation in the global character array (created by the macro), in case insufficient space remains. Note that without explicitly allocating more space, the array would grow automatically to the proper size; this operation, however, can be inefficient. The value of the **:extra-character-space** option is a non-negative integer number of characters.

To invoke the program `printarrays` with 3000 characters of dynamic character space, type:

```
(f77:execute ftn-user:printarrays :extra-character-space 3000.)
```

:extra-number-space

The **:extra-number-space** option is applicable only when you have used the **f77:with-fortran-number-data** macro in your program. This option enables you to specify the amount of extra space to reserve for allocation in the global number array (created by the macro) in case insufficient space remains. Note that without explicitly allocating more space, the array grows automatically to the proper size; this operation is not as efficient. The value of the **:extra-number-space** option is a non-negative integer number of words.

To invoke the program `printarrays` with 10,000 words of dynamic number space, type:

```
(f77:execute ftn-user:printarrays :extra-number-space 10000.)
```

:init-to-zero

Normally, Symbolics FORTRAN sets variables to the Lisp string "Undefined" when they are not explicitly initialized by FORTRAN data statements. The hardware flags an error if any program attempts to manipulate this value as a number. However, a problem arises if some of your programs actually depend on the absence of checking for uninitialized values. To avoid this potential problem, use the **:init-to-zero** option.

Value	*Meaning*
t	Initializes memory-resident FORTRAN variables, space for which allocates at run time, to zero.
nil	Sets FORTRAN variables to the Lisp string "Undefined". **nil** is the default.

Note that this option controls initialization of arrays, common variables, and other global variables. To control initialization of local variables by the compiler, use the initialization variable **f77:*ftn-init-to-zero***. See the section "Compiler Option: Initializing FORTRAN Program Data".

:libraries

A main program can call a defined FORTRAN library via the **:libraries** option. See the section "Declaring a FORTRAN Library".

Value	*Meaning*
library-name	Specifies a list of names of FORTRAN libraries.

To run a main program, `taxes`, that depends on the library `taxtable`, type:

```
(f77:execute ftn-user:taxes :libraries (taxtable))
```

:pathname-default

The **:pathname-default** option specifies the pathname used as the default by the FORTRAN I/O system when parsing filenames specified in the **:units** option, or in `open` or `inquire` statements.

The pathname specified as the default merges with the prevailing default, **fs:*default-pathname-defaults***. If you do not specify the **:pathname-default** option, the prevailing default is used.

Assume that the prevailing pathname default is "s:>tc>tc.lisp". You run `payroll`, supplying the **:pathname-default** option:

```
(f77:execute 'ftn-user:payroll :pathname-default ".fortran")
```

The pathname system constructs a new default "s:>tc>tc.fortran".

Assume also that the statement calling `open` reads:

```
open(unit = 3, file = '>tc>payroll-dir>', recl = 6)
```

When the program executes, the pathname of the opened file is constructed from the new prevailing default and the value of the file parameter:

```
"s:>tc>payroll-dir>tc.fortran"
```

For more information, see the section "Pathname Defaults and Merging".

:reload

The **:reload** option builds a new environment.

Value	*Meaning*
t	Discards any previously saved environments and builds a new one. Use this option when you radically change programs and need to recoup storage after a large number of incremental loads.
nil	Does not build a new environment; **:reload** has no effect.

Suppose, after running `mean` numerous times, you want to flush the environment and then rerun `mean`, performing all data initialization only once.

```
(f77:execute ftn-user:mean :save-environment :no-copy
                           :reload t)
```

:save-environment

For very large programs, especially those with many data statements, exploring the call tree and initializing the data is time consuming. For this reason, the FORTRAN run-time system supports the **:save-environment** keyword.

Note: An environment remains saved until you discard it using **:reload**.

The value of **:save-environment** indicates how to save the environment.

Value	*Meaning*
:copy	Preserves the data space for the main program between executions of the program. Suppose you want the data area allocated only once instead of every time you run `mean`, but you also want variables initialized at every program execution. Type: `(f77:execute ftn-user:mean :save-environment :copy)`
:no-copy	Performs the required data initialization only once instead of every time the program runs. The underlying assumption is that all data statements initialize variables, which do not change during execution of the program.
nil	Does not save the environment; **:save-environment** has no effect.

:system

The files of a FORTRAN system (given as values to the **:files** keyword) can contain more than one main program. To run a main program that depends on any of the elements of the system definition, for example, **:libraries**, use the **:system** keyword to access them. See the section "Declaring a FORTRAN System".

Note: If the program depends only on a system library, you can substitute the **:libraries** keyword.

Value	*Meaning*
system-name	Specifies the name of a FORTRAN system defined by **f77:def-program**.

To run a main program, axis, in the system plot, where axis calls a system library and a routine in another main program in plot, type:

```
(f77:execute ftn-user:axis :system plot)
```

:trap-underflow

By default, Genera signals an underflow if a result is too small for expression as a normalized single-precision or double-precision floating-point number — less than ˜1.175 x 10^{-38} and less than ˜2.2 x 10^{-308}, respectively.

To turn off trapping mode, use the **:trap-underflow** option, which sets the result in this case to 0 or to a denormalized number (see Jerome Coonen, et al., :A Proposed Standard for Binary Floating Point Arithmetic: Draft 8.0 of IEEE Task P754", Microprocessor Standards Committee, IEEE Computer Society, *Computer*, March 1981).

Bear in mind that your result might lose some accuracy.

Value	*Meaning*
nil	Turns on nontrapping mode; sets the result, if too small for expression as a normalized floating-point number, to 0 or a denormalized number.
t	Turns off nontrapping mode; underflow is detected if a result is too small for expression as a normalized floating-point number. Trapping mode is the system default.

To prevent the detection of underflow while running a main program, add, type:

```
(f77:execute ftn-user:add :trap-underflow nil)
```

:units

In the call to the main program you can specify files associated with FORTRAN logical units. The value for the keyword **:units** is a Lisp list, each element of which is a pair of values enclosed in parentheses:

((unit-number file-source) (unit-number file-source))

Value	*Meaning*
unit-number	Specifies the FORTRAN logical unit number associated with the file-source. Valid unit numbers are non-negative integers.
file-source	Refers to the file associated with the FORTRAN logical unit and can be any of these:

- A file specification string, giving the pathname of a file.

  The file specification string and the pathname are merged with the system pathname default (the variable **fs:*default-pathname-defaults***), as modified by the **:pathname-default** main program option, if present.

- A pathname flavor instance.

- A stream.

During execution of the main program mean, unit 3 is associated with the file "f:>tc>abc.ftn" and unit 4 with "f:>sr>xyz.ftn".

```
(f77:execute mean :units ((3 "f:>tc>abc.ftn") (4 "f:>tc>xyz.ftn")))
```

The Standard specifies that you can backspace and rewind any file — operations not permitted on normal Genera files. Symbolics FORTRAN allows these normal FORTRAN operations, however, with some associated run-time cost. Thus, if your programs do not use the backspace and rewind statements, it is more efficient to pass in Genera-style streams, as in the following example.

When the main program mean executes, standard input associates with the file "f:>tc>abc.text" and standard output with the file "f:>tc>xyz.text". Output to logical unit 14 is discarded.

```
(with-open-file (input "f:>tc>abc.text"
                           :direction :input :characters t)
  (with-open-file (output "f:>tc>xyz.text"
                              :direction :output :characters t)
    (eval '(f77:execute mean :units ((5 ,input)
                                     (6 ,output)
                                     (14 ,#'si:null-stream))))))
```

Debugging Symbolics FORTRAN Programs

Introduction

0.0.213. Standard Debugger

The standard Debugger starts automatically when an error occurs, and provides information about the Lisp or FORTRAN routine causing the error.

The Debugger provides features enabling you to:

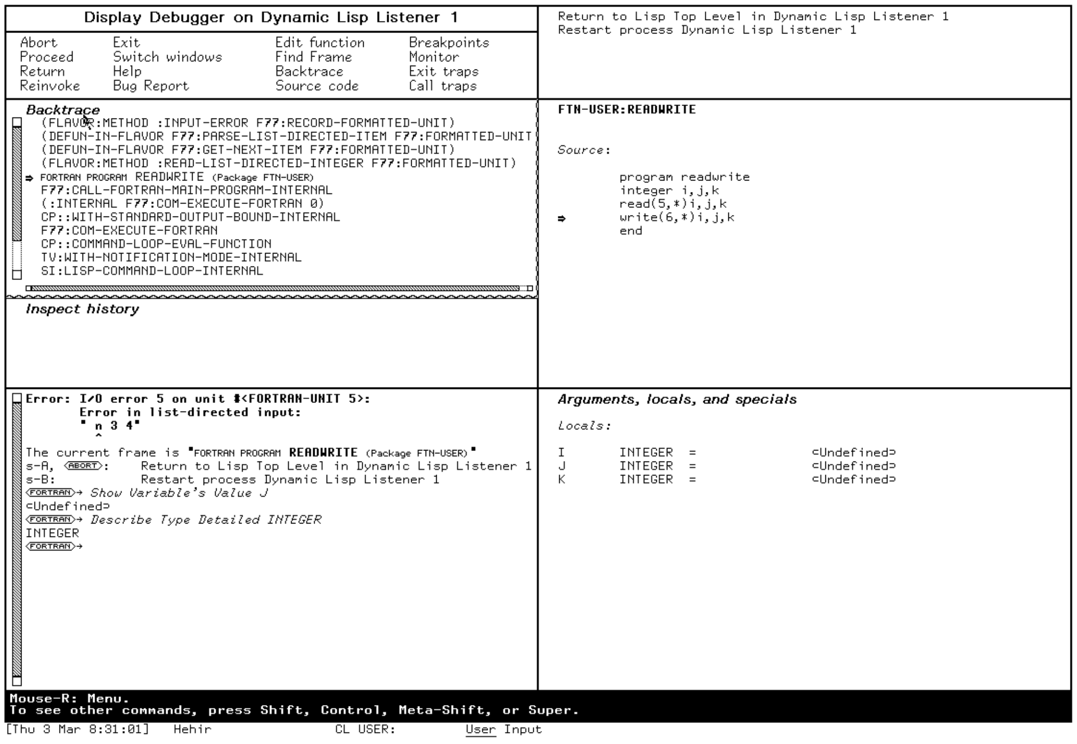
- Examine the backtrace of routines leading to the error at the source level. If the error occurred in a FORTRAN routine, a small arrow on the left side of the Inspect pane of the Debugger points at the location of the error in the source file. In any case, you can examine the sequence of calls, at the source level, from the FORTRAN program to the error site.

- Examine variables and arguments at the source level.
- Enter executable FORTRAN statements and have them evaluated at the debug level.
- Set and clear breakpoints.

The standard debugger presents these capabilities in the Listener window in which the program is executed.

0.0.214. Display Debugger

The standard Debugger invokes the Display Debugger. The Display Debugger has the same functionality as the standard Debugger, but provides a structured framework for you to work in in the form of a multipaned window. See the section "Using the Display Debugger". To invoke the Display Debugger, press `c-m-W` at the FORTRAN Debugger prompt. Figure 16 is an example of a FORTRAN program in the Display Debugger.



```
  (FLAVOR:METHOD :INPUT-ERROR F77:RECORD-FORMATTED-UNIT)
  (DEFUN-IN-FLAVOR F77:PARSE-LIST-DIRECTED-ITEM F77:FORMATTED-UNIT
  (DEFUN-IN-FLAVOR F77:GET-NEXT-ITEM F77:FORMATTED-UNIT)
  (FLAVOR:METHOD :READ-LIST-DIRECTED-INTEGER F77:FORMATTED-UNIT)
⇒ FORTRAN PROGRAM READWRITE (Package FTN-USER)
  F77:CALL-FORTRAN-MAIN-PROGRAM-INTERNAL
  (:INTERNAL F77:COM-EXECUTE-FORTRAN 0)
  CP::WITH-STANDARD-OUTPUT-BOUND-INTERNAL
  F77:COM-EXECUTE-FORTRAN
  CP::COMMAND-LOOP-EVAL-FUNCTION
  TV:WITH-NOTIFICATION-MODE-INTERNAL
  SI:LISP-COMMAND-LOOP-INTERNAL
```

```
FTN-USER:READWRITE

Source:

        program readwrite
        integer i,j,k
        read(5,*)i,j,k
⇒       write(6,*)i,j,k
        end
```

```
Error: I/O error 5 on unit #<FORTRAN-UNIT 5>:
       Error in list-directed input:
       " n 3 4"
         ^
The current frame is "FORTRAN PROGRAM READWRITE (Package FTN-USER)"
s-A, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
s-B:             Restart process Dynamic Lisp Listener 1
<FORTRAN>→ Show Variable's Value J
<Undefined>
<FORTRAN>→ Describe Type Detailed INTEGER
INTEGER
<FORTRAN>→
```

```
Locals:

I       INTEGER  =              <Undefined>
J       INTEGER  =              <Undefined>
K       INTEGER  =              <Undefined>
```

```
Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Thu 3 Mar 8:31:01]   Hehir              CL USER:         User Input
```

Figure 66. A FORTRAN program in the Display Debugger

0.0.215. Types of I/O Errors

Four classes of error conditions can occur:

End-of-File

> You can handle this error by placing an `end=` statement in your code. (See the ANSI Standard, section 12.7.2). If you choose *not* to handle these errors, the Debugger starts automatically.

Recoverable I/O error

> You can handle this error by placing an `err=` statement in your code. (See the ANSI Standard, section 12.7.1.) If you choose *not* to handle these errors, the Debugger starts automatically.
>
> An example of a recoverable I/O error is opening a nonexistent file.

Irrecoverable, or fatal, error

> Starts the Debugger.
>
> An example of an irrecoverable, or fatal error is using an illegal character constant.

System error

> External to FORTRAN. See your system maintainer.
>
> An example of a system error is network failure.

0.0.216. Error-Handling Mechanism

This section outlines the basic theory underlying error handling by the I/O system; this information has no functional application.

An error is detected by means appropriate to the type of error. For example, end-of-file errors are trapped by establishing a **condition-bind** for that specific condition. See the section "Conditions". Special-case code in the I/O subsystem handles data translation errors. The code that detected the error determines the internal error code; again, most of these errors are special cases.

The I/O subsystem then passes the error code to a sort of general-purpose I/O error handler, which determines whether or not the error is recoverable.

- If the error is recoverable *and* the calling program specifies that it wants control of the error (that is, you specified `err=` or `end=` so that control returns to the program without user intervention), the I/O system (1) returns the error code to the calling program's I/O status specifier (`iostat`) location, if any, and then (2) takes the appropriate return path, as specified in the calling program.

- If the error is not recoverable or if the calling program does not specify that it wants control, the I/O system notifies you of the error by displaying the error code and a message appropriate to the condition. At this point, the system is in a *debugger break*, and you can elect to abort, resume, or enter the Display Debugger.

Run-time Errors

0.0.217. FORTRAN I/O Status Specifiers

The Lisp function **(f77:print-fortran-errors)** displays on the screen all current FORTRAN I/O Status Specifiers, error code numbers, and messages. (See the ANSI Standard, section 12.9.7.) For this reason, the I/O Status Specifiers are not listed here.

f77:print-fortran-errors &optional *iostat* *Function*

Displays the message corresponding to *iostat*, the I/O Status Specifier error code, a digit from 1 onward. The top number depends on the software; as error messages are added with each new software revision, the range of error codes grow. When you omit the argument, the function displays all the I/O error codes and error messages.

`(f77:print-fortran-errors 4)` displays:

```
*FTN-ILLEGAL-FORMAT* is Not Recoverable; error return is 4
       Message is "Format statement contains too few data items"
       Documentation is CL:NIL
```

0.0.218. Example 1: A FORTRAN Error

Figure 17, page 198, shows the faulty execution of program `readwrite` in a Zmacs breakpoint window; the source code for this simple program is in the lower window. Note that `readwrite` is executed with the wrong type of arguments, a character string ('abc') instead of integers.

As a result, the FORTRAN I/O system generates an irrecoverable FORTRAN error 6:

```
Error in list-directed input ...
```

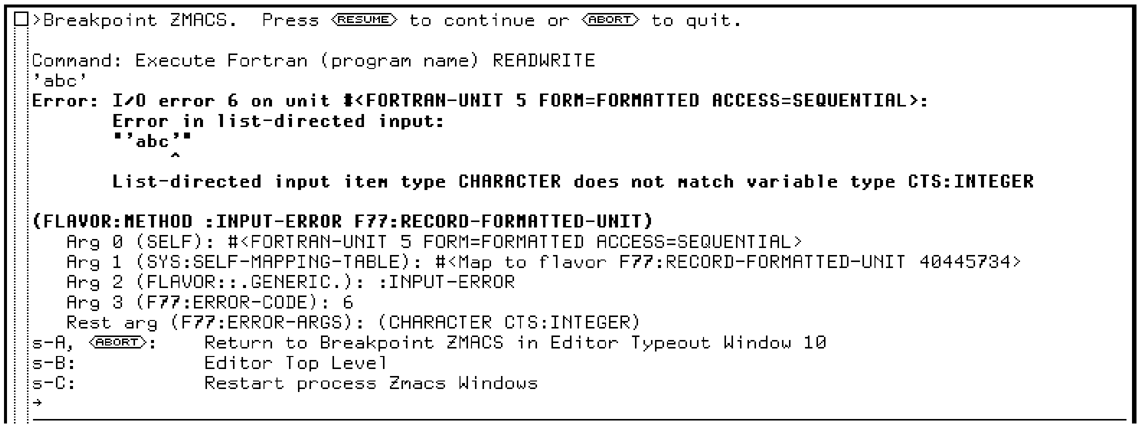
Then it isolates the error, marking the offending argument with a pointer that looks like an up-arrow (^).

The Lisp error message is internal to the Lisp run-time system.

Following the Lisp error message are the *proceed types*, a list of the possible actions you can take. For example, pressing `s-B` or `ABORT` returns you to the original Zmacs breakpoint window, where you can rerun `readwrite` with the correct argument type.

0.0.219. Example 2: A Lisp-Detected Error

The Lisp I/O system detects hundreds of its own errors and generates error messages. For example, figure 18, page 198, shows program `readwrite` executed without any arguments. This causes the program to enter a *debugger break*, indicated by "Error".



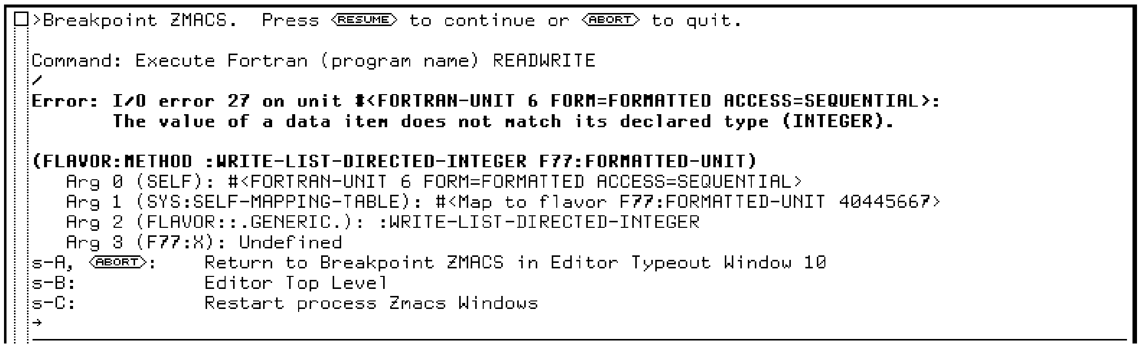
```
□>Breakpoint ZMACS.  Press RESUME to continue or ABORT to quit.

Command: Execute Fortran (program name) READWRITE
'abc'
Error: I/O error 6 on unit #<FORTRAN-UNIT 5 FORM=FORMATTED ACCESS=SEQUENTIAL>:
       Error in list-directed input:
       "'abc'"
            ^
       List-directed input item type CHARACTER does not match variable type CTS:INTEGER

(FLAVOR:METHOD :INPUT-ERROR F77:RECORD-FORMATTED-UNIT)
   Arg 0 (SELF): #<FORTRAN-UNIT 5 FORM=FORMATTED ACCESS=SEQUENTIAL>
   Arg 1 (SYS:SELF-MAPPING-TABLE): #<Map to flavor F77:RECORD-FORMATTED-UNIT 40445734>
   Arg 2 (FLAVOR::.GENERIC.): :INPUT-ERROR
   Arg 3 (F77:ERROR-CODE): 6
   Rest arg (F77:ERROR-ARGS): (CHARACTER CTS:INTEGER)
s-A, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window 10
s-B:           Editor Top Level
s-C:           Restart process Zmacs Windows
→
```

Figure 67. Example of an error produced by the FORTRAN I/O system.

In this example, where the cause of the error is obvious, you can press `ABORT` to return to an editor breakpoint and rerun the program.



```
□>Breakpoint ZMACS.  Press RESUME to continue or ABORT to quit.

Command: Execute Fortran (program name) READWRITE
/
Error: I/O error 27 on unit #<FORTRAN-UNIT 6 FORM=FORMATTED ACCESS=SEQUENTIAL>:
       The value of a data item does not match its declared type (INTEGER).

(FLAVOR:METHOD :WRITE-LIST-DIRECTED-INTEGER F77:FORMATTED-UNIT)
   Arg 0 (SELF): #<FORTRAN-UNIT 6 FORM=FORMATTED ACCESS=SEQUENTIAL>
   Arg 1 (SYS:SELF-MAPPING-TABLE): #<Map to flavor F77:FORMATTED-UNIT 40445667>
   Arg 2 (FLAVOR::.GENERIC.): :WRITE-LIST-DIRECTED-INTEGER
   Arg 3 (F77:X): Undefined
s-A, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window 10
s-B:           Editor Top Level
s-C:           Restart process Zmacs Windows
→
```

Figure 68. Example of an error detected by the Lisp I/O system.

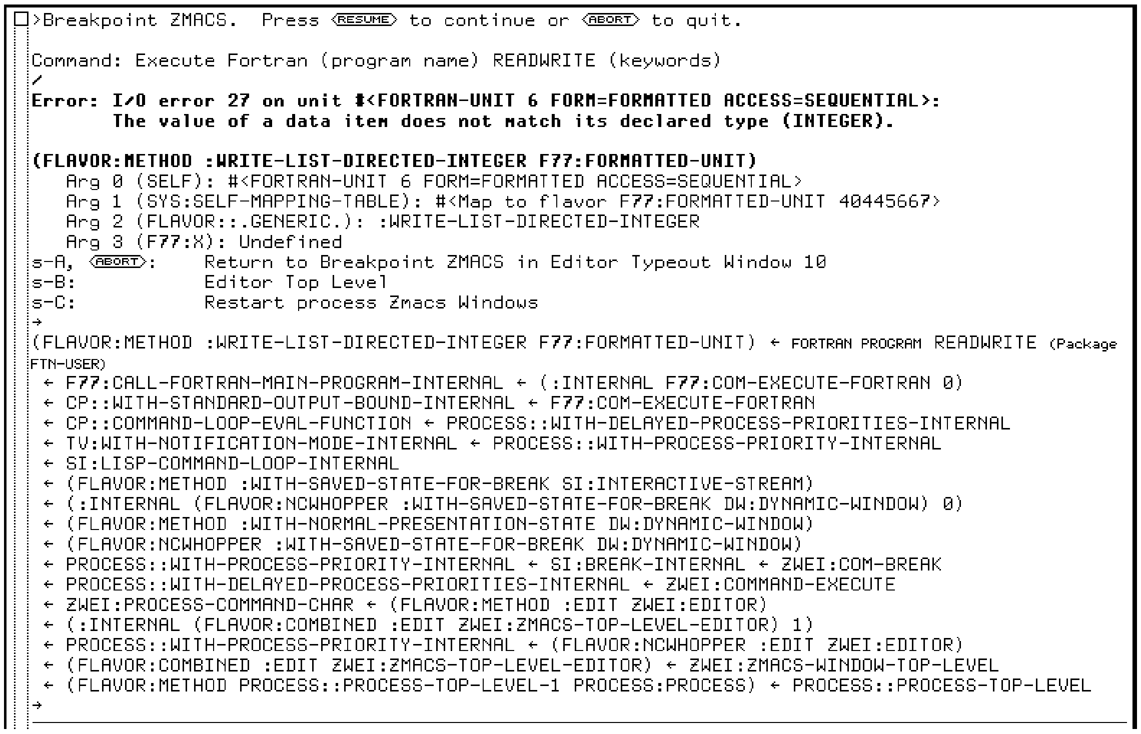
0.0.220. Example 3: A Backtrace

In cases where the error message is more puzzling, you can print a *backtrace* (a backward trace) of the program stack via `c-B`. Figure 19, page 199, shows a mouse-sensitive backtrace of all active functions for the execution of `readwrite`. Note that the arrows indicate the direction of calling, and that all hyphenated expressions are Lisp functions. `readwrite` is a very simple program, but, if the routines in which the error occurred are deeply nested, the backtrace marks the entire path from the onset of program execution to the error.

A similar command, `m-B`, prints the names of the arguments to each function and their current values, in addition to the backtrace.

For more information:

- See the section "Debugger".
- See the section "Conditions".



```
□>Breakpoint ZMACS.  Press RESUME to continue or ABORT to quit.

Command: Execute Fortran (program name) READWRITE (keywords)
⁄
Error: I/O error 27 on unit #<FORTRAN-UNIT 6 FORM=FORMATTED ACCESS=SEQUENTIAL>:
       The value of a data item does not match its declared type (INTEGER).

(FLAVOR:METHOD :WRITE-LIST-DIRECTED-INTEGER F77:FORMATTED-UNIT)
   Arg 0 (SELF): #<FORTRAN-UNIT 6 FORM=FORMATTED ACCESS=SEQUENTIAL>
   Arg 1 (SYS:SELF-MAPPING-TABLE): #<Map to flavor F77:FORMATTED-UNIT 40445667>
   Arg 2 (FLAVOR::.GENERIC.): :WRITE-LIST-DIRECTED-INTEGER
   Arg 3 (F77:X): Undefined
s-A, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window 10
s-B:           Editor Top Level
s-C:           Restart process Zmacs Windows
→
(FLAVOR:METHOD :WRITE-LIST-DIRECTED-INTEGER F77:FORMATTED-UNIT) ← FORTRAN PROGRAM READWRITE (Package
FTN-USER)
 ← F77:CALL-FORTRAN-MAIN-PROGRAM-INTERNAL ← (:INTERNAL F77:COM-EXECUTE-FORTRAN 0)
 ← CP::WITH-STANDARD-OUTPUT-BOUND-INTERNAL ← F77:COM-EXECUTE-FORTRAN
 ← CP::COMMAND-LOOP-EVAL-FUNCTION ← PROCESS::WITH-DELAYED-PROCESS-PRIORITIES-INTERNAL
 ← TV:WITH-NOTIFICATION-MODE-INTERNAL ← PROCESS::WITH-PROCESS-PRIORITY-INTERNAL
 ← SI:LISP-COMMAND-LOOP-INTERNAL
 ← (FLAVOR:METHOD :WITH-SAVED-STATE-FOR-BREAK SI:INTERACTIVE-STREAM)
 ← (:INTERNAL (FLAVOR:NCWHOPPER :WITH-SAVED-STATE-FOR-BREAK DW:DYNAMIC-WINDOW) 0)
 ← (FLAVOR:METHOD :WITH-NORMAL-PRESENTATION-STATE DW:DYNAMIC-WINDOW)
 ← (FLAVOR:NCWHOPPER :WITH-SAVED-STATE-FOR-BREAK DW:DYNAMIC-WINDOW)
 ← PROCESS::WITH-PROCESS-PRIORITY-INTERNAL ← SI:BREAK-INTERNAL ← ZWEI:COM-BREAK
 ← PROCESS::WITH-DELAYED-PROCESS-PRIORITIES-INTERNAL ← ZWEI:COMMAND-EXECUTE
 ← ZWEI:PROCESS-COMMAND-CHAR ← (FLAVOR:METHOD :EDIT ZWEI:EDITOR)
 ← (:INTERNAL (FLAVOR:COMBINED :EDIT ZWEI:ZMACS-TOP-LEVEL-EDITOR) 1)
 ← PROCESS::WITH-PROCESS-PRIORITY-INTERNAL ← (FLAVOR:NCWHOPPER :EDIT ZWEI:EDITOR)
 ← (FLAVOR:COMBINED :EDIT ZWEI:ZMACS-TOP-LEVEL-EDITOR) ← ZWEI:ZMACS-WINDOW-TOP-LEVEL
 ← (FLAVOR:METHOD PROCESS::PROCESS-TOP-LEVEL-1 PROCESS:PROCESS) ← PROCESS::PROCESS-TOP-LEVEL
→
```

Figure 69. A backtrace of the control stack.

Multilanguage Debugging with FORTRAN Programs

You can debug FORTRAN programs from the FORTRAN source level. The Debugger lets you examine the following language objects: variables, values, and types. In addition, you can evaluate expressions and statements from the Debugger.

This discussion assumes you have some knowledge of Debugger concepts and capabilities. In particular, it refers to:

stack frame
: A frame from the control stack that holds the local variables for the routine.

current stack frame

The context within which debugger commands operate. The debugger uses the current frame environment for performing operations according to the suspended state of your program. It evaluates forms in the lexical context of the function suspended in the current frame.

Initially, the current stack frame is the one that signalled the error.

Invoking the Debugger

You use the Debugger when you encounter a run-time error and are automatically thrown into the Debugger, when you use m-SUSPEND or c-m-SUSPEND to deliberately use the Debugger context, or by setting a breakpoint from the editor.

0.0.222. Exiting the Debugger

To exit the Debugger, use the ABORT key, the :Abort command, or invoke a restart option.

If you are in the middle of a series of recursive Debugger invocations, pressing ABORT returns you to the previous invocation. Keep pressing ABORT until you leave the Debugger and return to top level. Pressing m-ABORT from a recursive Debugger invocation brings you back to top level immediately.

0.0.223. Using Help

The Debugger offers you online help. Pressing the HELP key inside the Debugger displays several help options for you to choose:

- c-HELP displays documentation about all Debugger commands. This documentation consists of brief command descriptions and available key-binding accelerators.

- The ABORT key takes you out of the Debugger. (You can enter the :Abort command or press c-Z instead of pressing ABORT.)

- c-m-W brings you into the Window Debugger. (You can enter the :Window Debugger command instead of pressing c-m-W.)

The REFRESH key, the :Show Frame command, or the :Show Frame command accelerator c-L clears the screen, then redisplays the error message for the current stack frame.

You can also ask for help with keywords. If you do not remember what keywords are available for the command you are entering, press the HELP key after you receive the keywords prompt. The Debugger displays a list of keywords for that command. For example:

```
→ :Previous Frame (keywords) HELP
You are being asked to enter a keyword argument

These are the possible keyword arguments:
:Detailed             Show locals and disassembled code
:Internal             Show internal interpreter frames
:Nframes              Move this many frames
:To Interesting       Move out to an interesting frame
```

0.0.221. FORTRAN Frames

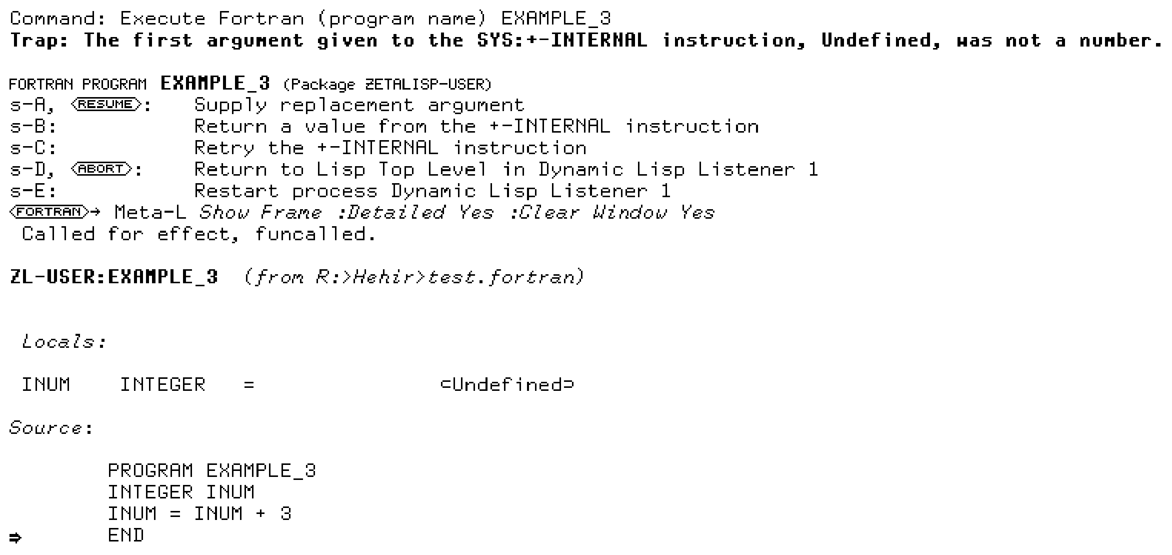
When you use the Debugger on a frame compiled in FORTRAN, you can get information about local and global variables and about the type and value of variables at various points in the source. You can also evaluate expressions and statements.

The Debugger prompt identifies the language used in compiling the current frame. For example, if you are debugging from a stack frame compiled in FORTRAN, you see the prompt:

```
<FORTRAN>→
```

Frames compiled in Lisp display the arrow prompt.

The next example shows the Debugger operating in the context of a FORTRAN frame. The program is EXAMPLE_3 from "Porting FORTRAN 77 Programs" which increments `INUM`, an uninitialized integer, thereby producing an error. The example shows the initial debugging information and the result of using the `m-L` Debugger command to show local variables.



```
Command: Execute Fortran (program name) EXAMPLE_3
Trap: The first argument given to the SYS:+-INTERNAL instruction, Undefined, was not a number.

FORTRAN PROGRAM EXAMPLE_3 (Package ZETALISP-USER)
s-A, <RESUME>:   Supply replacement argument
s-B:             Return a value from the +-INTERNAL instruction
s-C:             Retry the +-INTERNAL instruction
s-D, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
s-E:             Restart process Dynamic Lisp Listener 1
<FORTRAN>→ Meta-L Show Frame :Detailed Yes :Clear Window Yes
 Called for effect, funcalled.

ZL-USER:EXAMPLE_3  (from R:>Hehir>test.fortran)


 Locals:

 INUM    INTEGER   =              ⊂Undefined⊃

Source:

        PROGRAM EXAMPLE_3
        INTEGER INUM
        INUM = INUM + 3
⇒       END
```

You can see that a list of local variables and their values follow the `m-L` command as well as source information. The items under local are all mouse-sensitive. For further information, see the section "Looking At Variables, Values, and Types in FORTRAN".

Looking At Variables, Values, and Types in FORTRAN

Local variables and their values are mouse-sensitive. The mouse documentation line displays what action occurs with each mouse click. The following table summarizes this information for variables, values, and types.

Language Object		*Mouse click*	
	Left	*Middle*	*Right*
variable	:Show value	:Show type	menu
value	Returns the value	Describes the value	menu
type	:Show type	:Show type :detailed	menu

Using the mouse offers a quick and efficient way for you to inspect a variable, value, or type. In particular, you can use the mouse in getting a complete description of a complex type.

How FORTRAN Values Are Displayed in the Debugger

Uninitialized Values

An uninitialized value prints out as the symbol

undefined

Values that Exist out of Range of An Object

When you try to access a value beyond the range of the *underlying* allocated Lisp object, the value prints out as the symbol

unallocated_memory

Unfortunately, there is no one-to-one correspondence between a Lisp object and a language object, since many language objects are allocated within a Lisp object. Thus, violating the bounds of a language object does not always yield the symbol *unallocated_memory.*

Summarized Values

Summarized values (objects that are too large to print out by default, such as: arrays, structures unless requested) are printed out in summary form between the characters ≤≥. The summary form contains an abbreviated type indication followed by a unique number that helps distinguish two different values.

For example,

≤struct {...} 97541456≥

Values not of the Declared Type

Values are printed between horseshoes when the value as indicated by the tags in the hardware does not correspond to the declared type for the value. An example of this is attempting to obtain the value of a variable declared as an integer but actually containing a real.

For example,

```
⊂1.3⊃
```

FORTRAN Language Debugger Commands

0.0.224. Variables, Values, and Types

The following table summarizes the Debugger commands that work in FORTRAN frames and give specific information for FORTRAN variables, values, and types. The left column represents command processor commands and accelerators, and the right column shows corresponding menu choices.

Commands for variables:

:Show Local (`m-L`)

Commands for values:

:Show Variable's Value	Examine the value associated with this variable
:Show Detailed Value	Examine this value in greater detail

Commands for types:

:Show Variable's Type	Examine the type associated with this variable
:Show Value's Type	
:Describe Type Detailed	Describe the type in greater detail
:Show Type Name	Show the type name

Other:

Edit Viewspecs (menu only)

Expressions and Statements

You can evaluate expressions and statements from the Debugger. Type the expression or statement and press `END` to evaluate it.

Commands
Definition

Commands for stepping

:Statement Step For Function	Program execution stops in the debugger before the execution of each statement
:Clear Statement Step For Function	Clears the :Statement Step For Function enabling the program to execute normally

The following debugging commands are useful when using the stepping feature. In order to see a complete list of all debugging commands, specify :language help from the debugger.

:Show Source Code

`c-X c-D`	Show the source code for the function in the current frame.

:Next Frame

`c-N,`	Move down a frame (takes numeric argument), skipping invisible frames.
`m-sh-N`	Move down a frame, not skipping invisible frames.
`m-N`	Move down a frame, displaying detailed information about it.
`c-m-N`	Move down a frame, not hiding internal interpreter frames.

:Previous Frame

`m-P`	Move up a frame (takes numeric argument), skipping invisible frames.
`m-sh-P`	Move up a frame, not skipping invisible frames.
`m-P`	Move up a frame, displaying detailed information about it.
`c-m-P`	Move up a frame, not hiding internal interpreter frames.
`c-m-U`	Move to the next frame that is not an internal interpreter frame.

:Show Backtrace

`c-B`	Displays a brief backtrace, hiding invisible frames, but not censoring continuation frames.
`c-sh-B`	Displays a brief backtrace of the stack, censoring invisible internal (continuation) frames. Use a numeric argument to indicate how many frames to display.
`m-B`	Displays a detailed backtrace of the stack.
`m-sh-B`	Displays a brief backtrace, without censoring invisible or continuation frames.
`c-m-B`	Displays a detailed backtrace of the stack, including internal frames.

Genera Debugger Commands

This section summarizes the Genera Debugger commands you can use in debugging FORTRAN programs. They are listed according to the following areas of functionality:

- Commands for viewing a stack frame
- Commands for stack motion
- Command for general information display
- Commands to continue execution
- Trap commands
- Commands for breakpoints and single stepping
- Commands for system transfer

Most of these commands are described fully in "Debugger Command Descriptions".

Commands for Viewing a Stack Frame

:Show Arglist (`c-X c-A`)
:Show Argument (`c-m-A`)
:Show Compiled Code (`c-X D`)
:Show Frame (`REFRESH, c-L, m-L`)
:Show Function (`c-m-F`)
:Show Local (`c-m-L`)
:Show Source Code (`c-X c-D`)
:Show Stack
:Show Value (`c-m-V`)

Commands for Stack Motion

:Bottom Of Stack (`m->`)
:Find Frame (`c-S`)
:Next Frame (`LINE, c-N, m-N, c-m-N`)
:Previous Frame (`RETURN, c-P, m-P, c-m-P, c-m-U`)
:Set Current Frame
:Top Of Stack (`m-<`)

Commands for General Information Display

:Describe Last (c-m-D)
:Show Backtrace (c-B, m-B, c-m-B)
:Show Instruction (c-m-I)

Commands to Continue Execution

:Abort (ABORT, c-Z)
:Disable Aborts
:Enable Aborts
:Proceed (RESUME)
:Reinvoke (c-m-R)
:Return (c-R)

Trap Commands

:Clear Trap On Call (c-X c-C)
:Clear Trap On Exit (c-X c-E)
:Disable Condition Tracing (c-X T)
:Enable Condition Tracing (c-X T)
:Show Monitored Locations

Commands for Breakpoints and Single Stepping

:Clear All Breakpoints
:Clear Breakpoint
:Set Breakpoint
:Show Breakpoints
:Single Step (c-sh-S)

Commands for System Transfer

:Edit Function (c-E)
:Mail Bug Report (c-M)

Lisp Frame Debugging

If at any point in a debugging session you want to return to to Lisp mode debugging, use the :Use Lisp Mode command to toggle to that mode.

FORTRAN-Lisp Interaction

Overview of FORTRAN-Lisp Interaction

0.0.225. Contents

This chapter discusses the interface between Lisp and FORTRAN, including the following issues:

- `lispobject`, an additional data type for representing Lisp objects in FORTRAN programs.
- `lispfunction`, an additional FORTRAN declaration statement that allows the declaration of an existing Lisp function, which you can subsequently call from a FORTRAN routine.
- How to call FORTRAN from a Lisp function.
- How to call a Lisp function from FORTRAN.
- Macros for the dynamic allocation of FORTRAN data.
- FORTRAN coercion of Lisp data types.

`lispobject`: FORTRAN Data Type for Handling Lisp

In addition to the data types defined in the Standard, Symbolics FORTRAN supports a scalar data type called `lispobject`, facilitating interaction with Lisp. By declaring a variable a `lispobject`, you can represent any lisp data object. You can form arrays of `lispobjects` and declare `lispobject` functions.

However, the only valid operations on objects of the `lispobject` type are assignment and parameter passing; they cannot be read, written, or even compared against each other. You cannot coerce any other type into a `lispobject`, and vice versa.

`lispfunction`: Type Declaration Statement

Symbolics FORTRAN supports a FORTRAN declaration statement, known as `lispfunction`, allowing you to declare a Lisp function you can call during a FORTRAN routine.

0.0.226. Format

The format is:
`lispfunction` *fortran-name-for-lisp-function 'lisp-function-name'*
        (*input-type-1, ..., input-type-*n) *output-type*

where the following conventions are used:

- Single quotes enclose the Lisp function name.
- Commas separate multiple input types.
- Parentheses enclose the required argument list.

0.0.227. Example

```
Declaration  fortran-name-for-lisp-function          input-type-n
name         |                           input-type-1    |
|            |     lisp-function-name    |               |  output-type
|            |        |                  |               |      |
↓            ↓        ↓                  ↓               ↓      ↓
lispfunction get time 'cl-user::get-time' (character(10), integer) integer
```

0.0.228. Description

`lispfunction` specifies the name of the declaration statement for calling a Lisp function. It takes the following arguments.

fortran-name-for-lisp-function

Specifies a valid FORTRAN name that FORTRAN uses to call the Lisp function. *fortran-name-for-lisp-function* can be the name of an existing, valid Lisp function (predefined or user-defined) as long as that name is valid in FORTRAN; for example, `length` is a valid FORTRAN identifier that is also the name of an actual Lisp function, **length**. When the FORTRAN name for the Lisp function is identical to the Lisp function name, it is not necessary to specify the *lisp-function-name* argument.

fortran-name-for-lisp-function is also a valid FORTRAN name that you can select to represent a Lisp function. For example, FORTRAN recognizes `get time` as the FORTRAN name of the user-defined Lisp function **cl-user:get-time**. In this case you must specify *lisp-function-name*.

lisp-function-name

Optional if *fortran-name-for-lisp-function* is identical to *lisp-function-name* and both are in the same package. Required if the Lisp function is in a different package than the FORTRAN program, or if the Lisp function name contains characters that are invalid in a FORTRAN identifier, such as "-" and "*". If present, *lisp-function-name* is specified as a FORTRAN string literal, whose contents are the print name of a Lisp function.

In this example, **cl-user:get-time** contains a hyphen and a colon, which are invalid characters in FORTRAN.

```
lispfunction get time 'cl-user:get-time' ....
```

input-type-1, ... input-type-n

Specifies the input data type of the argument to *lisp-function-name*.

Supply any of the standard FORTRAN scalar data types (integer, real, logical, double-precision, complex, and character) or `lispobject`, the extension to the Standard.

To specify an array of the indicated data type, follow the type by a single constant dimension in parentheses. If specified, the dimension must be an integer constant expression indicating the size of an array. The lower bound is 1.

When you pass arrayed parameters, the FORTRAN compiler creates Lisp indirect arrays, pointing into the FORTRAN data array, for each arrayed parameter. Thus, when the Lisp routine sets elements of the arrays passed into it, the values are modified in the FORTRAN data space as well. Nonarrayed parameters (scalar data) are passed by value, not reference, so changing the value in the Lisp routine has no effect on the FORTRAN data address space.

Enclose all the input data types that make up the parameter list in parentheses. Separate multiple data types with commas.

Example: Two input data types are given: a 10-slot, single-dimension character array and a scalar of type integer.

```
lispfunction time 'cl-user:get-time'
        (character(10),integer) ...
```

output-type Optional. Specifies the data type of the output, or value, returned from `lisp-function-name`.

If present, *output-type* must be one of the standard FORTRAN scalar data types (integer, real, logical, double-precision, complex, and character) or `lispobject`, the extension to the Standard. You can invoke the routine as a FORTRAN function. The Lisp routine is then expected to return an item of the indicated data type.

If *output-type* is absent, the routine returns no value and is invoked as a subroutine rather than as a function.

Symbolics FORTRAN does not handle character arrays as an *output-type*. See the section "Output-Type Restriction of `lispfunction`".

0.0.229. Example 1

Declare the Lisp function **length** in a FORTRAN routine; it returns an integer corresponding to the length of a list.

```
        .
        .
lispfunction length (lispobject) integer
lispobject a
        .
```

Use the variable **length** in the routine.

```
w=2*length(a)
```

0.0.230. Example 2

The Lisp function **screen-clear** refreshes the screen and **ball** draws a filled-in circle.

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 8 -*-
```

```
(defun screen-clear ()
  "Clears the screen"
  (send *standard-output* :clear-history))

(defun ball (center-x center-y radius)
  "Draws a ball whose center is (center-x, center-y)"
  (send *terminal-io* :draw-filled-in-circle
        center-x center-y radius tv:alu-xor))
```

The FORTRAN program `throwball` declares the Lisp function **ball** as a `lispfunction` that has three integer arguments.

```
C  -*- Mode: FORTRAN; Package: FTN-USER -*-

        program throwball
        implicit none
        integer i
        lispfunction ball 'cl-user::ball' (integer, integer,
                                           integer) integer
        lispfunction beep 'tv:beep' () lispobject
        lispfunction screenclear 'cl-user::screen-clear' ()
                                                lispobject

        call screenclear
        do i = 50, 300, 20
           call ball(i, i, 10)
        enddo
c       get the user's attention
        call beep
        end
```

Restriction on `doubleprecision` Arrays

`doubleprecision` numbers are represented as *boxed* numbers in the Lisp world, and as *unboxed*, or unpacked, numbers in FORTRAN. This difference stems from the fact that each manipulation of a boxed number creates storage to return a result. This is unnecessary overhead for FORTRAN.

This difference requires that you use caution in `lispfunction` declarations when passing arrays of type `doubleprecision` between Lisp and FORTRAN. This caution does not apply to scalars specified in `lispfunction` declarations; you can pass a double-precision scalar that Lisp can manipulate directly.

To pass an array of `doubleprecision` numbers to FORTRAN, unbox the Lisp number by calling **si:dfloat-components non-complex-number** on the Lisp double. **si:dfloat-components non-complex-number** returns two "integers", *x* and *y*, representing the high and low portions of the number. These can be placed separately in a Lisp array for manipulation by FORTRAN. Conversely, you can call

si:%make-double on *x* and *y* to produce a boxed double for Lisp to manipulate.

Example 1:

`add one to double` calls the `lispfunction` `addem`, passing in `a`, a double-precision scalar. The Lisp interface boxes `a` before passing it to Lisp. The macro **f77:with-fortran-number-data** creates the array **lisp-double-array**. The body code adds 1 to the boxed number and then unboxes it so FORTRAN can handle it, placing the high and low portions (*x* and *y*) in **lisp-double-array**. The FORTRAN subroutine `print double` is called with the FORTRAN address of the unboxed double-precision number as its argument.

```
c  -*- Mode: FORTRAN; Package: FTN-USER -*-

        program add one to double
        implicit none
        doubleprecision external print double
        doubleprecision a
        lispfunction addem
     x     'cl-user::add-one-in-lisp' (doubleprecision) lispobject
        external print double
        a = 23.d0
        call addem(a)
        end

        doubleprecision function print double(a)
        implicit none
        doubleprecision a
        print *,a
        print double = a
        end



;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 8 -*-

(defun add-one-in-lisp (double)
  (f77:with-fortran-number-data (lisp-double-array
                                  fortran-double-address 2)
     (multiple-value-bind (x y)
         (si:double-components (+ double 1.0d0))
       (setf (aref lisp-double-array 0) x)
       (setf (aref lisp-double-array 1) y)
       (ftn-user:print double fortran-double-address))))
```

Example 2:

`add one to double array element` calls the `lispfunction addem to array`, passing in `a`, a 3-element double-precision array. Each element is unboxed. The Lisp function **add-one-to-array-in-lisp** pulls out the high and low portions of the first element and calls **si:%make-double** to box the number. Lisp adds 1 to the number, unboxes it, and places the high and low portion back into double-array.

```
c  -*- Mode: FORTRAN; Package: FTN-USER -*-

        program add one to double array element
        implicit none
        lispfunction addem to array
     x     'cl-user::add-one-to-array-in-lisp' (doubleprecision(3))
     x     lispobject
        doubleprecision a(3)
        a(1) = 23.0
c    perform in Lisp the effect of a(1) = a(1) + 1.0
        call addem to array (a)
        print *,a(1)
        end
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 8 -*-

(defun add-one-to-array-in-lisp (double-array)
  (let* ((x (aref double-array 0))    ;get x of double-array(1)
         (y (aref double-array 1))    ;get y of double-array(1)
         (double (si:%make-double x y)))
    (multiple-value-bind (new-x new-y)
        (si:dfloat-components (+ double 1.0d0))
      (setf (aref double-array 0) new-x)
      (setf (aref double-array 1) new-y))))
```

Output-Type Restriction of `lispfunction`

Symbolics FORTRAN cannot handle a character array as an *output-type*. The workaround is to pass an additional parameter to the Lisp routine declared as a character array.

A FORTRAN routine calls a Lisp routine to get the time into a 20-character array, passing a time-zone indication:

```
C  -*- Mode: FORTRAN; Package: FTN-USER -*-
```

```
        program print time
        implicit none
        character*20 time
        integer zone
        lispfunction xtime 'cl-user::get-the-time' (character(20),
        integer)
      + integer
c     Let time-zone be something ridiculous, like -100
        zone = -100
        time = '0123456789'
        call xtime(time, zone)
c     Note that the arrayed parameter time is changed, but zone, an
c     integer passed by value, is not changed
        print *,time, zone
        end
```

And the Lisp code:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 8 -*-

;;; string is an indirect array, whereas timezone is a Lisp integer
(defun get-the-time (string timezone)
  ;; setq time-zone, to show that it has NO effect on the value
  ;; in the Ftn data space
  (setq timezone time:*timezone*)               ;get the timezone
  (let* ((time-string (string-time timezone))   ;call string-time
         (copy-length (min 20 (string-length time-string))))
    ;; copy as much of the time string as will fit in the Ftn string
    (loop for i below copy-length
          do (setf (aref string i) (aref time-string i)))
    ;; pad the rest of the Ftn string (if any) with spaces
    (loop for i from copy-length below 20
          do (setf (aref string i) #\SPACE))))

;;; given a time-zone, returns a string of date and univeral time
(defun string-time (timezone)
  (with-output-to-string (stream)
    (time:print-universal-time (time:get-universal-time)
                               stream timezone)))
```

Alternative to `lispfunction`

0.0.231. Introduction

Symbolics FORTRAN provides a means for calling Lisp without having to declare the Lisp routine from FORTRAN. In this case the following conditions obtain:

1. All parameters are represented by integers from Lisp's point of view. These integers are indices into the FORTRAN data space, and hence are a kind of "address" of the data.

   All FORTRAN data lives in two address spaces, one for character data and one for numeric data. These address spaces are represented by two Lisp arrays, one that stores general objects, and one that holds just characters. At run time, addresses in each of the arrays are represented by integer indices into the arrays.

2. A return value from a function is the same as the return value from Lisp, with the exception that return values of type `doubleprecision` consist of two integers, representing the high and low portions of the number. See the section "Restriction on `doubleprecision` Arrays".

0.0.232. Referencing Values in Lisp

Within the Lisp code, you can reference any `character` argument value by invoking `(f77:global-char` *parameter*`)`, and any non-`character` argument value by invoking `(f77:global-word` *parameter*`)`. These forms expand into array references (**aref**s) of the FORTRAN data area arrays, indexed by the integer passed in as an address parameter.

If FORTRAN passed a(1), then `(f77:global-word` *parameter*`)` directly accesses a(1). To set the *n*th element, invoke **f77:global-word** with an argument of (*n-1*).

Example:

```
subroutine test
implicit none
integer a(20)
call bar (a(1))
end

lispobject function print lispobject (what)
implicit none
lispobject what
print *,what
end

program test it
implicit none
lipsobject external print lispobject
call test end
```

```
(defun ftn-user:bar (a-addr)
  ;; set a(1) to 1
  (setf (f77:global-word a-addr) 1)
  ;; to set a(9) to 9, setf a-addr(n-1) to 9
  (setf (f77:global-word (+ a-addr 8)) 9)
  ;; set a(3) to 'FOO
  (setf (f77:global-word (+ a-addr 2)) 'FOO)
  ;; pass a(3) to FORTRAN routine print lispobject
  (ftn-user:print lispobject (+ a-addr 2))
  ;; pass a(9) to FORTRAN routine print lispobject
  (ftn-user:print lispobject (+ a-addr 8)))
```

Note that you can pass parameters passed from FORTRAN to undeclared Lisp functions, for example, **ftn-user:bar**, directly back to FORTRAN code as parameters of the same type, as in the hypothetical **ftn-user:print_lispobject**. This technique is, in fact, the only available means for calling FORTRAN subroutines and functions from Lisp.

0.0.233. Example

`printtime` is a variant of the `print time` example shown in another section. See the section "Output-Type Restriction of `lispfunction`". Compare the two examples.

Both programs call a Lisp routine to return the time in a 20-character array, but unlike `print time`, `printtime` does not declare a `lispfunction` to call Lisp. `printtime` thinks that it is calling another FORTRAN program called `xtime`.

```
C  -*- Mode: FORTRAN; Package: FTN-USER -*-

        program printtime
        implicit none
        character*20 time
        integer zone
c     Let time-zone be something ridiculous, like -100
        zone = -100
        time = '0123456789'
        call xtime(time, zone)
c     Note that both time (an arrayed parameter) and zone
c     (an integer passed by value) are changed
        print *,time, zone
        end
```

Here is the Lisp routine **xtime**. Note that **f77:global-word** and **f77:global-char** replace the **aref**s used in **get-the-time**, the Lisp routine called by `print time`.

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 8 -*-

;;; Note that the Lisp routine name must be in the same Lisp
;;; package as the FORTRAN routine calling it.
;;; string is an indirect array, whereas time-zone is a Lisp integer.

;;; xtime was called from printtime with 2 args, but Ftn passes
;;; in an additional arg -- the maximum length of string-index.

(defun ftn-user:xtime (string-index string-max-length timezone-index)
  ;; setq time-zone, to show that it DOES affect the value
  ;; in the Ftn data space
  (setf (f77:global-word timezone-index) time:*timezone*)
  (let* ((time-string (string-time (f77:global-word timezone-index)))
         (copy-length (min string-max-length
                              (string-length time-string))))
    ;; copy as much of the time string as will fit in the Ftn string
    (loop for i below copy-length
          do (setf (f77:global-char (+ string-index i))
                   (aref time-string i)))
    ;; pad the rest of the Ftn string (if any) with spaces
    (loop for i from copy-length below string-max-length
          do (setf (f77:global-char (+ string-index i))
                   #\SPACE))))
```

Calling FORTRAN from Lisp

Prerequisite

You can read the following sections for background information:

- See the section "Executing FORTRAN Programs".
- See the section "Lisp Syntax for FORTRAN Users".

0.0.234. Description

A Lisp function can call a FORTRAN main program directly.

In figure 20, page 217, the Lisp function **callfortran**, shown in the top pane, calls the FORTRAN program `iftest`, associating logical unit 7 with the output file "s:>tc>examples-output" at run time. The bottom pane displays the FORTRAN source code.

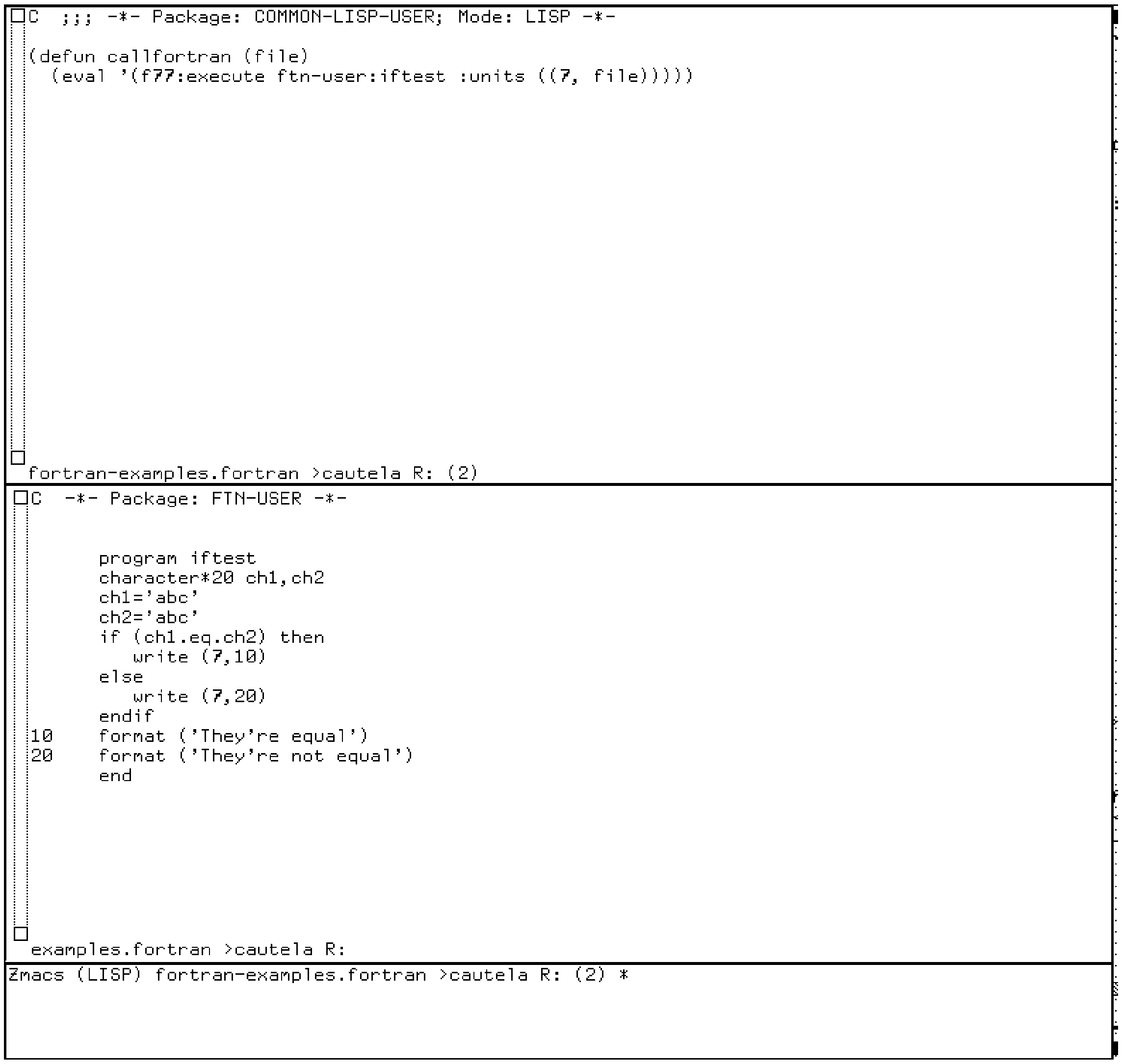
To invoke a FORTRAN program from Lisp while allowing the substitution of a run-time unit in the units list, you must use the Lisp function **eval**. Note in the following example that the **f77:execute** function requires a backquote and that the file source requires a comma.

To call the FORTRAN main program `iftest` from the Lisp function **callfortran** and specify the logical unit at run time, type:

```
(defun callfortran (file)
  (eval `(f77:execute ftn-user:iftest :units ((7 ,file)))))
```

To run **callfortran** designating unit 7 as "s:>tc>examples-output", type:

```
(callfortran "s:>tc>examples-output")
```



```
C  ;;; -*- Package: COMMON-LISP-USER; Mode: LISP -*-

(defun callfortran (file)
  (eval '(f77:execute ftn-user:iftest :units ((7, file)))))

fortran-examples.fortran >cautela R: (2)
```

```
C  -*- Package: FTN-USER -*-


      program iftest
      character*20 ch1,ch2
      ch1='abc'
      ch2='abc'
      if (ch1.eq.ch2) then
         write (7,10)
      else
         write (7,20)
      endif
10    format ('They're equal')
20    format ('They're not equal')
      end

examples.fortran >cautela R:
```

```
Zmacs (LISP) fortran-examples.fortran >cautela R: (2) *
```

Figure 70. A Lisp function calls a FORTRAN main program.

Calling Lisp from FORTRAN

A FORTRAN main program can call a Lisp function, which in turn calls a FORTRAN routine or subroutine.

The crucial point is that you have to place a FORTRAN routine or subroutine on the call tree of a main program if the routine has local variables or data statements needing initialization. Data initialization occurs when you execute the main program. See the section "Executing FORTRAN Programs".

In order to call a FORTRAN routine from a Lisp function, one of the following conditions must be true:

- The FORTRAN routine appears as a function in the main program or in some routine called by the main program.
- The FORTRAN routine is declared external in the main program. (See the ANSI Standard, section 8.7.)

0.0.235. Example

Figure 21, page 219, illustrates how program `runinter` (shown in the middle pane) declares and calls a Lisp function **runinteract**. In turn, **runinteract** (shown in the top pane) calls a FORTRAN subroutine `interact`, which has a data statement to be initialized at run time. The external statement in the main program makes it valid for **runinteract** to call `interact`. The bottom pane shows execution of the program and output to the terminal.

Macros for Dynamic Allocation of FORTRAN Data

All FORTRAN data lives in two address spaces, one for character data and one for numeric data. These address spaces are represented by two Lisp arrays, one that stores general objects, and one that holds just characters. At run time, addresses in each of the arrays are represented by integer indices into the arrays.

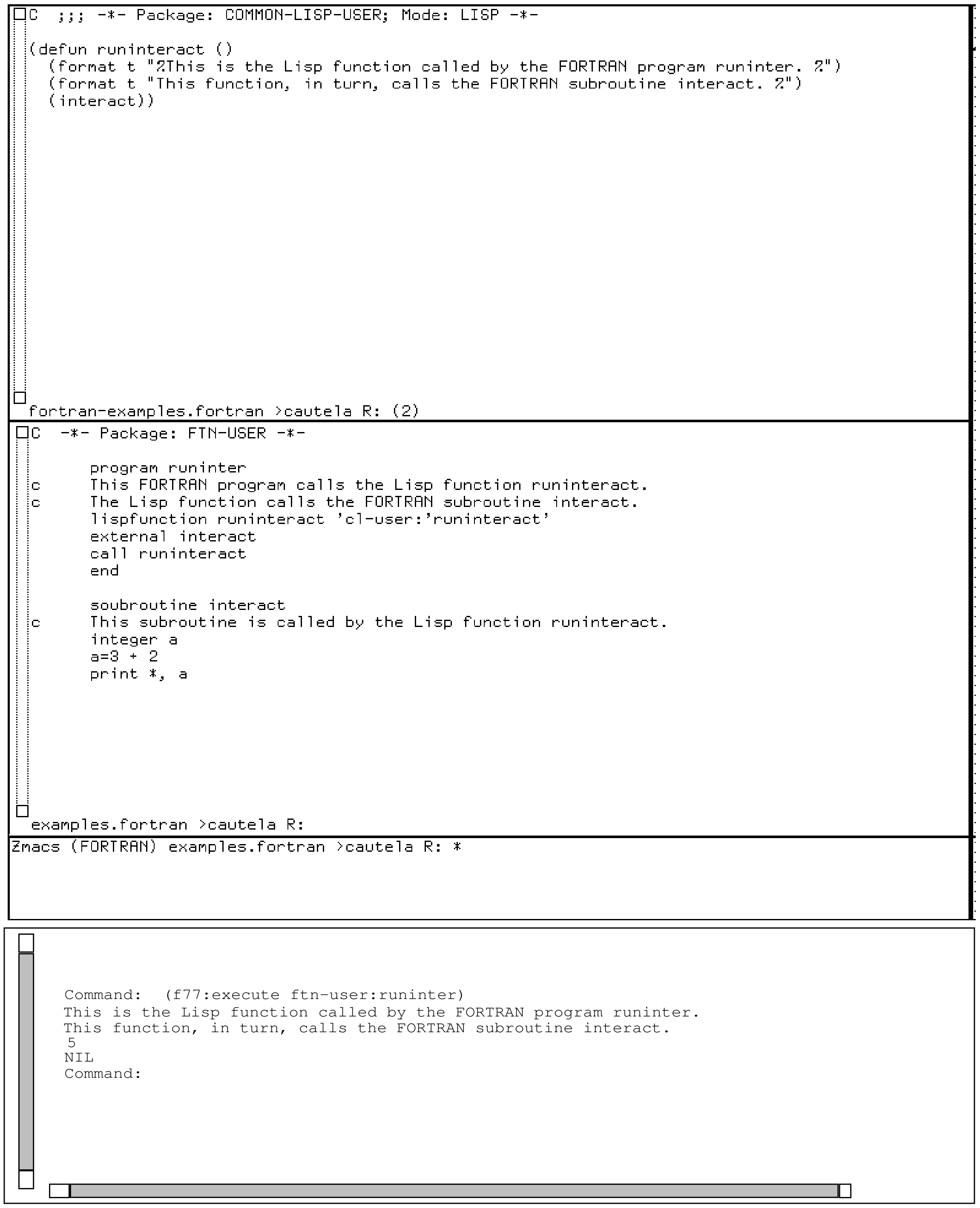
The process of invoking a FORTRAN main program performs a dynamic linking operation by exploring the call tree, starting from the main program. In the process, the offsets of the numeric and character data are accumulated. Afterwards, the two arrays — one for each kind of data — are allocated.

This process works well for data allocated by FORTRAN, but it does not provide an easy mechanism for creating possibly size-unresolved data in Lisp code and then passing it into FORTRAN. The macros **f77:with-fortran-number-data** and **f77:with-fortran-character-data** provide a mechanism for this purpose.

0.0.236. Numeric Data

f77:with-fortran-number-data *((lisp-array fortran-data-address dimension-1 ...dimension-n) ...)* &body *body* *Macro*

Provides a mechanism for creating dynamic data in Lisp and allowing FORTRAN (and Lisp) to manipulate that data. No previous declaration of the data by FORTRAN is necessary. The following situation illustrates a typical use of the macro:



```
C  ;;; -*- Package: COMMON-LISP-USER; Mode: LISP -*-

(defun runinteract ()
  (format t "%This is the Lisp function called by the FORTRAN program runinter. %")
  (format t "This function, in turn, calls the FORTRAN subroutine interact. %")
  (interact))

fortran-examples.fortran >cautela R: (2)
```

```
C  -*- Package: FTN-USER -*-

      program runinter
c     This FORTRAN program calls the Lisp function runinteract.
c     The Lisp function calls the FORTRAN subroutine interact.
      lispfunction runinteract 'cl-user:'runinteract'
      external interact
      call runinteract
      end

      soubroutine interact
c     This subroutine is called by the Lisp function runinteract.
      integer a
      a=3 + 2
      print *, a

examples.fortran >cautela R:
Zmacs (FORTRAN) examples.fortran >cautela R: *
```

```
Command:  (f77:execute ftn-user:runinter)
This is the Lisp function called by the FORTRAN program runinter.
This function, in turn, calls the FORTRAN subroutine interact.
 5
NIL
Command:
```

Figure 71. A FORTRAN main program calls a Lisp function, which calls a FORTRAN routine.

Suppose you want to call a Lisp user interface from a FORTRAN main program, whose purpose is gathering data to pass back to FORTRAN subroutines. The Lisp code allocates data using the **f77:with-fortran-number-data** macro, fills in the values in the Lisp version of the array, and then passes the values back to FORTRAN by providing the FORTRAN data address.

The first subform is a list of lists; each list consisting of these elements: *(lisp-array fortran-data-address dimension-1 ... dimension-n)*. *lisp-array* and *fortran-data-address* must be symbols, and are bound to the indirected Lisp array created and the FORTRAN data address, respectively. If the array has more than one dimension, a multidimensional Lisp array returns. To allocate a scalar to pass back to FORTRAN, allocate an array whose single dimension has the value 1.

The body of the macro contains Lisp code to manipulate *lisp-array*. The lifetime of the Lisp array so produced is the scope of the body of the macro invocation. You can nest invocations either in a single routine, or dynamically at run time, in several routines.

The macro is used only by a Lisp routine called directly or indirectly from a FORTRAN main program. Note that Symbolics Common Lisp stores arrays in row-major order and that Lisp arrays are addressed relative to 0. Thus, any references to multidimensional Lisp arrays created by **f77:with-fortran-number-data** must conform to this scheme.

The array created by

```
(with-fortran-number-data ((lisp-array ftn-address i j)) ...
```

is referenced as

```
     (aref lisp-array (- j 1) (- i 1))
```

For information on doubleprecision arrays, see the section "Restriction on `double-precision` Arrays". If the global number array is not large enough, the array automatically grows; however, since this operation is expensive, Symbolics FORTRAN provides the run-time option **:extra-number-space**. The option enables you to specify the amount of extra space needing allocation when the main program initializes.

The FORTRAN program `spline draw` draws two splines. It calls `get and plot`, prompting you for the number of points in each spline. `get and plot` calls the declared `lispfunction get splines and plot`, passing in the number of points. The actual Lisp function **get-splines-and-plot** creates five arrays and fills four of them with the coordinates of each point (entered via mouse clicks). The splines are drawn. The Lisp function calls the FORTRAN subroutine `draw lines`, which calls two Lisp functions to draw dashed lines between the corresponding points of each spline, and a solid line through the midpoints of each dashed line.

The FORTRAN source code file:

```
C  -*- Mode: FORTRAN; Package: FTN-USER -*-

        program spline draw
        external draw lines
```

```
      call get and plot
      end

      subroutine get and plot
      integer points
      external draw lines
      lispfunction get splines and plot
     x      'cl-user::get-splines-and-plot' (integer)
      lispfunction clrscreen 'cl-user::clrscreen'
      call clrscreen
      points = 1
      do while (points .ne. 0)
         print *
         print *,
     x     'Enter a number > 2 of points in each line. Enter 0 to exit'
         read *,points
         call clrscreen
         if (points .eq. 0) then
            stop
         elseif (points .le. 2) then
            print *, 'Need three or more points'
         else
            call get splines and plot(points)
         endif
      enddo
      end

      subroutine draw lines(points,fx1,fy1,fx2,fy2)
      lispfunction drawdashedline 'cl-user::drawdashedline'
c             from-x  from-y  to-x    to-y
     x       (integer,integer,integer,integer)
      lispfunction drawline 'cl-user::drawline'
c             from-x  from-y  to-x    to-y
     x       (integer,integer,integer,integer)
      integer points
      integer fx1(points),fy1(points),fx2(points),fy2(points)
      integer midx, midy, newmidx, newmidy
c    initialize midx and midy to the first line segment midpoint
      midx=(fx1 (1) + fx2(1))/2
      midy=(fy1 (1) + fy2(1))/2
      do i=1,points
c    call a Lisp function to draw a dashed line between the
c    corresponding points of each spline.
         call drawdashedline(fx1(i),fy1(i),fx2(i),fy2(i))
         newmidx = (fx1(i) + fx2(i))/2
         newmidy = (fy1(i) + fy2(i))/2
         If (i .gt. 1) then
c    call a Lisp function to draw a solid line through the
```

```
c     midpoints of each dashed line.
              call drawline(midx,midy,newmidx,newmidy)
           endif
           midx = newmidx
           midy = newmidy
        enddo
        end
```

The Lisp source code file:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 8 -*-

(defun get-splines-and-plot (size)
  ;; create five arrays: x1, y1, x2, y2, and points (scalar)
  (f77:with-fortran-number-data ((x1 fx1 size)
                                 (y1 fy1 size)
                                 (x2 fx2 size)
                                 (y2 fy2 size)
                                 (points fpoints 1))
                        ;fill points w/ # of pts requested
    (setf (aref points 0) size)
                        ;fill x1 & y1 w/ coords for spline 1
    (get-spline-values x1 y1 1 size)
                        ;fill x2 & y2 w/ coords for spline 2
    (get-spline-values x2 y2 2 size)
                        ;call Ftn subroutine w/ Ftn addrs
    (ftn-user:draw lines fpoints fx1 fy1 fx2 fy2)))

(defun get-spline-values (x y curve-num size
                          &optional (window *terminal-io*) (width 4)
                          (alu tv:alu-ior) (precision 20.))
  (format window "~&Drawing curve number ~D, containing ~D points"
          curve-num size)
  (tv:with-mouse-and-buttons-grabbed
    (mouse-draw-spline
      size
      window
      x
      y
      (format nil "Draw curve ~D containing ~D points. Left: Set Point."
              curve-num size))
    (dotimes (n size)
      (funcall window :draw-rectangle 3 3 (1- (aref x n))
               (1- (aref y n)) tv:alu-xor))))

(defun mouse-draw-spline (size window px py
                          documentation-string &aux dx dy)
  (multiple-value-setq (dx dy)
    (tv:sheet-calculate-offsets window tv:mouse-sheet))
  (setq dx (+ dx (tv:sheet-inside-left window))
        dy (+ dy (tv:sheet-inside-top window)))
  (setq tv:who-line-mouse-grabbed-documentation
        documentation-string)
  (loop with i = 0
        with old-x = nil with old-y = nil doing
    (multiple-value-bind (buttons x y)
        (tv:wait-for-mouse-button-down)
      (if (≠ buttons 1)
          (tv:beep)                ;Didn't click left
          (decf x dx)              ;Convert to window inside coords
          (decf y dy)
          ;;If same point is put in twice, err with 0-divide later
          (unless (and (eql x old-x) (eql y old-y))
            (send window :draw-rectangle 3 3 (1- x) (1- y) tv:alu-xor)
            (setf (aref px i) x)   ;fill arrays with coords
            (setf (aref py i) y)
            (incf i))
          (setq old-x x old-y y)))
        until (eq i size)))

(defun drawline (x1 y1 x2 y2)
  (send *terminal-io* :draw-line x1 y1 x2 y2))

(defun drawdashedline (x1 y1 x2 y2)
  (send *terminal-io* :draw-dashed-line x1 y1 x2 y2))

(defun clrscreen ()
  (send *standard-output* :clear-history))
```

0.0.237. Character Data

f77:with-fortran-character-data *((lisp-array fortran-data-address dimension-1 ...dimension-n) ...)* &body *body* *Macro*

Provides a mechanism for creating dynamic data in Lisp and allowing FORTRAN (and Lisp) to manipulate that data. No previous declaration of the data by FORTRAN is necessary. The following situation illustrates a typical use of the macro:

Suppose you wanted a FORTRAN main program to call a Lisp user interface, whose purpose was to gather data to pass to FORTRAN subroutines. The Lisp code allocates data using the **f77:with-fortran-character-data** macro, fills in the character values in the Lisp version of the array, and passes the character values back to FORTRAN by providing the FORTRAN data address.

The first subform is a list of lists; each list consisting of these elements: *(lisp-array fortran-data-address dimension-1 ... dimension-n)*. *lisp-array* and *fortran-data-address* must be symbols and are bound to the indirected Lisp array created and the FORTRAN data address, respectively. You must pass back to FORTRAN both the data address and, for any character parameter, the total size, which is the product of the dimensions, as in:
(ftn-pkg:ftn-routine ftn-data-address (dimension-1 .. dimension-n))*
If the array has more than one dimension, a multidimensional Lisp array is returned. To allocate a scalar to pass back to FORTRAN, allocate an array whose single dimension has the value 1.

The body of the macro contains Lisp code to manipulate *lisp-array*. The lifetime of the Lisp array so produced is the scope of the body of the macro invocation. Invocations can be nested either in a single routine, or dynamically at run time, in several routines.

The macro can be used only by a Lisp routine called directly or indirectly from a FORTRAN main program. Note that Symbolics Common Lisp stores arrays in row-major order and that Lisp arrays are addressed relative to 0. Thus, any references to multidimensional Lisp arrays created by **f77:with-fortran-character-data** must conform to this scheme.

The array created by
```
(with-fortran-character-data ((lisp-array ftn-address i j)) ...)
```
is referenced as
```
    (aref lisp-array (- j 1) (- i 1)).
```

Note: If the global number array is not large enough, the array automatically grows; however, since this operation is expensive, Symbolics FORTRAN provides the run-time option **:extra-character-space**. This option enables you to specify the amount of extra space allocated when the main program initializes.

Example: `make and print arrays` calls **chardostuff** to fill a single-dimension array with character values. It passes back the FORTRAN data address, as well as the product of the array dimensions (10 x 1). `charprintstuff` prints the characters.

The FORTRAN source code file:
```
c -*- Mode: FORTRAN; Package: FTN-USER -*-

      program make and print arrays
      implicit none
      lispfunction chardostuff 'cl-user::chardostuff' ()
      lispobject
      external charprintstuff
      call chardostuff
      end
```

```
        subroutine charprintstuff(ca)
        implicit none
        character*(*) ca
        print *,ca
        end
```

The Lisp source code file:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp;
;;; Package: USER; Base: 8 -*-

(defun chardostuff ()
  (f77:with-fortran-character-data
   ((lisp-data fortran-address 10))
  ;; fill the array
  (loop for i below 10 do
    (setf (aref lisp-data i)
          (code-char (+ (char-code #\a) i))))
  (ftn-user:charprintstuff fortran-address 10)
  ;; fill the array another way
  (zl:fillarray
   lisp-data
   '(#\a #\b #\c #\d #\e #\f #\g #\h #\i #\j))
  (ftn-user:charprintstuff fortran-address 10)))
```

FORTRAN Coercion of Lisp Data Types

This table describes the FORTRAN to Lisp coercions done by the FORTRAN compiler:

FORTRAN	*Lisp*
real	real
integer	integer
double precision	double
complex	complex
array	*
character*1	character
character*n where n >1	string**

+Arrays are displaced into character arrays or numeric arrays

++character*n return values are not supported.

Porting FORTRAN 77 Programs

Considerations in Porting FORTRAN Programs

This section discusses considerations in porting programs developed on other compilers to Symbolics machines for use in the Genera environment. In particular, it describes some effects of run-time data type-checking and the treatment of uninitialized variables in the Genera environment. It also presents a table showing the size of FORTRAN 77 data types in this implementation.

Data Type-checking At Run-time

Run-time data type-checking is perhaps the most noticeable difference for programmers used to conventional untagged architectures. Programming errors, where such errors cause meaningless operations go undetected in conventional hardware, but are trapped by Symbolics machines.

For example, the following FORTRAN program attempts to use an integer value as a real value, thus producing a run-time error at the PRINT statement:

```
PROGRAM EXAMPLE 1
INTEGER INUM
REAL RNUM
EQUIVALENCE (RNUM,INUM)
INUM = 5
PRINT *,RNUM,INUM
END
```

The following error is produced:

Error: I/O error 27 on unit #<FORTRAN-UNIT 6>:
The value of a data item does not match its declared type (REAL).

Uninitialized Variables in FORTRAN

All variables start out with the distinguished value "undefined" unless they are explicitly initialized or assigned. You cannot coerce or write an undefined value. Data is also initialized to values that are undefined. The execution of the next FORTRAN example, which attempts to write a value for the uninitialized variable `INUM`, produces an error.

```
PROGRAM EXAMPLE 2
INTEGER INUM
PRINT *,INUM
END
```

Error: I/O error 27 on unit #<FORTRAN-UNIT 6>:
The value of a data item does not match its declared type (INTEGER).

This is the same error message as in the previous example, but notice the telltale value of Undefined for Arg 3.

Using an undefined value in the non-I/O case yields a different result that stems from the same basic problem. The following example attempts to increment the uninitialized variable `INUM`.

```
PROGRAM EXAMPLE 3
INTEGER INUM
INUM = INUM + 3
END
```

Trap: The first argument given to the SYS:+-INTERNAL instruction, Undefined, was not a number.

Although this is unexpected behavior to those new to a Symbolics machine, signalling an error, in such a case, is preferable to picking up a random machine-dependent value, and actually eases the porting process.

FORTRAN Data Type Sizes

Data Type	*Size*
character	8 bits
integer	1 word
real	1 word
double precision	2 words
complex	2 words
logical	1 word

Notes on Symbolics FORTRAN

This chapter describes implementation-specific information, machine-dependent behavior, and common user errors resulting from unexpected aspects of FORTRAN.

It presents information concerning

- Data type-checking
- List-directed I/O
- The SAVE statement

FORTRAN Floating-Point Numbers

0.0.238. Floating-Point Numbers

The following are the IEEE standard single and double-precision formats:

Single-float
Single-precision floating-point numbers have a precision of 24 bits, or about 7 decimal digits. They use 8 bits to represent the exponent. Their range is from 1.0e-45, the smallest positive denormalized single-precision number, to 3.4028235e38, the largest positive normalized single-precision number.

Double-float
Double-precision floating-point numbers have a precision of 53 bits, or about 16 decimal digits. They use 11 bits to represent the exponent. Their range is from 5.0d-324, the smallest positive denormalized double-precision floating-point number, to 1.7976931348623157d308, the largest positive normalized double-precision floating-point number.

For more information, see IEEE Floating-Point Representation for Fortran.

0.0.239. IEEE Floating-Point Representation for FORTRAN

The Symbolics computer uses IEEE-standard formats for single-precision and double-precision floating-point numbers. Number objects exist that are outside the upper and lower limits of the ranges for single and double precision. Larger than the largest number is +1e∞ (or +1d∞ for doubles). Smaller than the smallest number is -1e∞ (or -1d∞ for doubles). Smaller than the smallest normalized positive number but larger than zero are the "denormalized" numbers. Some floating-point objects are Not-a-Number (NaN); they are the result of (/ 0.0 0.0) (with trapping disabled) and like operations.

IEEE numbers are symmetric about zero, so the negative of every representable number is also a representable number. Zeros are signed in IEEE format, but +0.0 and -0.0 act the same arithmetically as 0.0. However, they are distinguishable to non-numeric functions. For example:

```
+0.0 .eq. -0.0
```

See "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, *An American National Standard*, August 12, 1985.

0.0.240. Constants Indicating the Range of Floating-Point Numbers

Constant

least-positive-single-float 1.4e-45

least-positive-normalized-single-float 1.4e-45

most-positive-single-float 3.4028235e38

least-negative-single-float -1.4e-45

least-negative-normalized-single-float -1.4e-45

most-negative-single-float -3.4028235e38

least-positive-double-float 2.2250738585072014d-308

least-positive-normalized-double-float 2.2250738585072014d-308

most-positive-double-float 1.7976931348623157d308

least-negative-double-float -2.2250738585072014d-308

least-negative-normalized-double-float -2.2250738585072014d-308

most-negative-double-float -1.7976931348623157d308

Since the exponent in floating-point representation has a fixed length, some numbers cannot be represented. Thus floating-point computations can get exponent overflow or underflow, if the result is too large or small to be represented. Exponent overflow always signals an error. Exponent underflow normally signals an error, unless the computation is inside the body of a **without-floating-underflow-traps**. Any time a floating-point error occurs, you are offered a way to proceed from it, by substituting the IEEE floating-point standard result for the mathematical result.

Example:

```
(* 4e-20 4e-20)                ;evalutating this signals an error
(without-floating-underflow-traps  (* 4e-20 4e-20)) => 1.6e-39
```

0.0.241. Non-mathematical Behavior of Floating-point Numbers

The restricted representation of floating-point numbers leads to much behavior which can be confusing to users unfamiliar with the concept. This behavior is characteristic of floating-point numbers in general, and not of any particular language, machine, or implementation.

Floating-point operations don't always follow normal mathematical laws. For example, floating-point addition is not associative:

```
(+ (+ 1.0e10 -1.0e10) 1.0) => 1.0
(+ 1.0e10 (+ -1.0e10 1.0)) => 0.0
```

This follows from the restricted representation of floating-point, since 1.0 is insignificant relative to 1.0e10.

Much of the confusion surrounding floating-point comes from the problem of converting from decimal to binary and *vice versa.*

Consider that the binary representation of 1/10 repeats infinitely:

```
.000110011001100110011001100110011001100110011001100 ...
```

Since we can't represent this exact value of 1/10, we would like to find the mathematically closest number which is representable. We do that by rounding to the appropriate number of binary places:

```
Single precision:  (24 significant bits)
.000110011001100110011001101

(describe (float 1/10 0.0)) =>
0.1 is a single-precision floating-point number.
 Sign 0, exponent 173, 23-bit fraction 23146315
(not including hidden bit)
 Its exact decimal value is 0.100000001490116119384765625
0.1

Double precision:  (53 significant bits)
.000110011001100110011001100110011001100110011001100110011010

(describe (float 1/10 0.0d0)) =>
0.1d0 is a double-precision floating-point number.
 Sign 0, exponent 1773, 52-bit fraction 114631463146314632
(not including hidden bit)
 Its exact decimal value is 0.10000000000000000555111512312578270211
8158340454101562 5d00.1d0
```

Already we see some anomalies. The single-precision number closest to 1/10 has a different mathematical value from the double-precision one. So a decimal number, when represented in different floating-point precisions, can have different values. Yet the printer prints both as "0.1".

Why do the printed representations hide the difference in values? Every binary number has an exact, finite, decimal representation, which can be printed. The **describe** function does that, as shown in the example above. From that example, you can see that printing exact values would be cumbersome without giving useful information. So the printer prints the shortest decimal number that is properly rounded (from the actual decimal value), and whose rounded binary value (in that precision) is identical to the original. Note that Fortran list directed I/O is thje same as the LISP floating-point printing rule.

Here is an example of the rule used to derive the shortest decimal number:

```
(describe 1.17) =>
1.17 is a single-precision floating-point number.
Sign 0, exponent 177, 23-bit fraction 05341217
(not including hidden bit)
Its exact decimal value is 1.16999995708465576171875
1.17
```

The correctly rounded decimal values for this single-precision number are:

```
1, 1.2, 1.17, 1.16999996, 1.169999957, 1.1699999571, 1.16999995708, etc.
```

Rounded to single-precision (binary), the first three printed representations are all different, but after 1.17, they are all the same. Thus, 1.17 is the "best" representation to print.

Since the printing rule is sensitive to floating-point precision, it hides the difference between the exact mathematical values of 1.17 and 1.17d0.

Consider the following example:

```
program floating
real r1
doubleprecision d1, d2
logical r1EQd1, r1EQd2
d1 = 1.17d0
d2 = 1.17e0
r1 = 1.17e0
r1EQd1 = (r1.eq.d1)
r1EQd2 = (r1.eq.d2)
print *,'d1 = ',d1,', d2 = ',d2,', r1 = ', r1,
$  ', r1.eq.d1 = ',r1EQd1,', r1.eq.d2 =',r1EQd2
end
```

Executing this program generates the following:

- d1 = 1.17D0
- d2 = 1.1699999570846558D0
- r1 = 1.17

- r1.eq.d1 = F
- r1.eq.d2 = T

An anomaly exists in that d1 and r1 print the same, but are not actually the same. Additionally, d2 and r1 print differently, but are actually the same.

Formatting Floating-point Output

You can specify (via the `format` statement) how many digits of a floating-point number are printed. Floating-point numbers have limited precision. If you ask for too many digits, some of the digits are meaningless. Therefore, the system (by default) prints trailing zeros rather than these insignificant digits. Setting the variable to non-**nil** makes the system print all of the digits that `format` specifies.

In order to control FORTRAN formatted output of floating-point numbers, set the Lisp variable **f77:*print-insignificant-digits*** to **t**.

f77:*print-insignificant-digits* *Variable*

Controls whether insignificant digits in floating-point numbers are printed. The default for this variable is **nil**.

For example, consider these statements:

```
        print 10, 0.1
10      format (d30.20)
```

They yield the following results, depending on whether you set **f77:*print-insignificant-digits*** to **t** or accept the default value of **nil**.

Value	*Result*
nil (default)	0.10000000000000000000E+00
t	0.10000000149011611938E+00

Note that both printed representations correspond to the same internal (binary) number. If the number of digits specified in the format is large enough so that ***print-insignificant-digits*** affects the printed representation, reading either printed number back gives the original number.

Editor Extensions for FORTRAN

This section contains a brief summary of the commands available for editing FORTRAN programs in the Zmacs editor, as well as for finding out about and interacting with your FORTRAN program.

Note: These commands are available only after you have set the buffer mode to FORTRAN.

It also describes variables that control FORTRAN editing mode features.

These commands represent modifications of the standard language-sensitive editing commands. They perform roughly the equivalent function of the editing commands for Lisp.

Most of these commands allow a preceding argument in the form c-*n*.

With the cursor is positioned in the middle of a routine, move forward from the current routine to the end of the next routine by typing:

```
c-2 c-m-E
```

For more general editing commands: See the section "Summary of Standard Editing Features".

FORTRAN Mode Editor Commands

`TAB` (Insert FORTRAN Tab)

In FORTRAN mode, if the cursor is in column 0 through 6, pressing `TAB` inserts the number of spaces to reach column 7. Otherwise, it inserts the number of spaces needed to reach every eighth column.

`LINE`

In FORTRAN mode, pressing `LINE` moves the line of text beginning with the cursor to column 7 of the next line. In effect, it combines using the `RETURN` and `TAB` keys in sequence.

`c-LINE` (Insert FORTRAN Continuation Line)

Moves the line of text beginning with the cursor to column 7 of the next line and inserts a continuation character at column 6.

`c-m-A`

Moves the cursor backward to the beginning of a FORTRAN routine.

`c-m-E`

Moves the cursor forward to the end of a FORTRAN routine.

`c-m-F`

Moves the cursor forward to the end of a FORTRAN statement.

`c-m-B`

Moves the cursor backward to the beginning of a FORTRAN statement.

`c-m-H` (Mark FORTRAN Definition)

Marks the language definition surrounding the cursor and moves the cursor to the beginning of the definition. (A definition is a subprogram.)

`c-sh-A`

Displays the name of the current FORTRAN routine, as well as a brief list of the argument types. Position the cursor after the chosen routine. `c-sh-A` applies only to functions that have already been compiled and loaded.

`c-sh-C`
Compiles the currently defined *region*, a contiguous delimited section of text in the editor buffer. See the section "Summary of Standard Editing Features". If no region is defined, it compiles the routine nearest the cursor. This command does not take a numeric argument.

An unsuccessful compilation results in the display of compiler warnings and suggested actions in a typeout window at the top of the screen. Pressing any character causes the window to disappear.

`c-^` (Merge FORTRAN Lines)
Deletes characters in columns 1 through 6 in the current line and and then merges the current line with the previous line.

`m-X` Clear All Breakpoints
Clears all breakpoints.

`m-X` Compile And Execute Fortran Program
Checks to see that the cursor is positioned near a valid FORTRAN program and then compiles and executes the program, without run-time options and with predefined file input and output bound to the editor typeout window.

`m-X` Compile Buffer
Compiles the entire buffer. With a numeric argument, it compiles from the cursor position to the end of the buffer. You can use Compile Buffer for resuming compilation after a prior compilation (initiated via this command) fails.

`m-X` Compile Changed Definitions of Buffer
Compiles any changed definitions of subroutines, programs, and functions in the buffer. With a numeric argument, it prompts individually about whether to compile each changed definition.

`m-X` Compile File
Compiles a file. It prompts for a file name, but defaults to the file associated with the current buffer. It offers to save the buffer if it is modified.

`m-X` Compile Region
See `c-sh-C`.

`m-X` Compiler Warnings
Puts all pending compiler warnings in a buffer called `*Compiler-Warnings-`*n*`*` (creating the buffer if it does not exist) and selects that buffer.

`m-X` Edit Changed Definitions of Buffer
Determines which definitions in the current buffer have changed and positions the cursor at the first one. Use `c-.` to move to the next changed definition. A numeric argument controls the starting point for determining what has changed.

1 = since the file was last read (default)
2 = since the buffer was last saved
3 = since the definition was last compiled

`m-X` Edit Compiler Warnings

Edits some or all routines whose compilation caused a warning message. It queries you, for each file mentioned in the compiler warnings database, whether you want to edit the warnings for the routines in that file. It splits the screen, placing the warning message in the top window and the source code whose compilation caused the error in the bottom window. Use `c-.` to move to the next pair of warning and source code.

`m-X` Edit Definition `(m-.)`

Edits definition of compiled FORTRAN routine. When it prompts for the name of the routine, subroutine, and so on, you can either (1) type the name in the minibuffer at the bottom of the screen or (2) click on a name appearing in the current buffer. The command finds the routine, places it in an editor buffer, and positions the cursor there.

The echo area displays a message indicating multiple occurrences of the definition, if any. Use `c-.` to move to the next occurrence.

This command is especially useful because it can find any definition in a loaded FORTRAN system, whether or not the file that contains it is currently in a buffer.

`m-X` Electric FORTRAN Mode

Places a FORTRAN buffer into electric mode. Electric FORTRAN mode is a facility that automatically wraps text when you type characters beyond column 72. The facility takes delimiters and whitespace into account when determining where to break a line, and adds continuation characters as needed.

Note that you must put the buffer in FORTRAN mode before using this command. To switch off electric FORTRAN mode, reissue the `m-X` Electric FORTRAN Mode command.

`m-X` Execute Fortran Program

Checks to see that the cursor is positioned near a valid, compiled FORTRAN program and then executes the program, without run-time options and with predefined file input and output bound to the editor typeout window.

`m-X` Find Next FORTRAN Sequence Number

Moves the cursor to the beginning of the next sequence number. This command takes numeric arguments, using negative numbers to search for previous sequence numbers and positive numbers to search for following sequence numbers.

`m-X` Fortran Mode

Sets the mode in an editor buffer to FORTRAN, enabling you to use the special FORTRAN-mode commands described in this appendix. The mode line at the bottom of the screen changes to `Zmacs (FORTRAN)` .... The command also offers to set the mode in the attribute list. If you respond `y`, it creates the following: `{-*- Mode: FORTRAN-*-`. Once the attribute list mode is set to FORTRAN, you need not set the mode again upon reinvoking the file; the mode is set to FORTRAN automatically.

m-X List Breakpoints
Lists all currently active breakpoints.

m-X List Changed Definitions of Buffer
Displays a list of any definitions that changed in the current buffer. Use a numeric argument to choose the condition that determines the search:

1 = since the file was last read (default)
2 = since the file was last saved (written)
3 = since each definition was last compiled

m-X List Definitions
Displays the definitions from a specified buffer. Type the buffer name of choice in the minibuffer at the bottom of the screen, or press RETURN to select the default (the current buffer).

It displays the list of names of subroutines, programs, and functions in a typeout window. You can (1) press SPACE to make the typeout window disappear or (2) use the mouse to select individual names; upon selection, the cursor is positioned at the name in the editor buffer.

m-X Load Compiler Warnings
Loads a file containing compiler warning messages into the warnings database. You are prompted for the name of a file containing the printed representation of the compiler warnings.

m-X Remove Sequence Numbers
Deletes all characters from column 73 onward. Operates on a buffer by default or on a marked region.

m-X Reparse Attribute List
Causes all changes made to the attribute list to take effect.

m-X Replace Tabs
Removes TAB characters and replaces them with the equivalent number of spaces.

(**Note:** This command is not specific to FORTRAN mode. It is described here because it is particularly useful with FORTRAN, for which <TAB> is not defined as a legal character. Its treatment varies widely from one FORTRAN implementation to another.)

m-X Set Package
Sets the package for the buffer. When it prompts for a name, enter the name of an existing package. It offers to set the package for the attribute list as well as for the buffer.

m-X Update Attribute List
Creates or updates the attribute list for the file. Executing the command after entering FORTRAN mode causes the attribute list to set the mode to FORTRAN.

FORTRAN Mode Editor Variables

The following variables have been added for use with FORTRAN editing mode:

f77:*wraparound-column* *value*
Sets the value for the minimum line length resulting from automatic wrapping in Electric FORTRAN mode. The default value is 40.

f77:*continuation-character* *value*
Defines the FORTRAN continuation character. The character is placed in the sixth column with `c-LINE` and in Electric FORTRAN mode. The default value is the dollar sign character ($).

Summary of Standard Editing Features

Use `SELECT E` to select Zmacs. The standard Zmacs commands are very similar to those of the EMACS editor. This section summarizes some categories of Zmacs commands. All editor commands can take a preceding numeric argument in the form `c-` or `m-` to modify their behavior in some way.

See the section "Zmacs".

Zmacs Help Facilities

`c-ABORT`	Aborts the function currently executing.
`c-G`	Aborts a command when entered, unselects the region, or unmerges a kill.
`HELP A` *string*	Shows every command containing *string* (try `HELP A Paragr` or `HELP A Buffer`).
`HELP C` *x*	Explains the action of any command (try `HELP C c-K` as an example).
`HELP D` *string*	Describes a command (try `HELP D Query Rep`).
`HELP L`	Displays the last 60 keys pressed.
`HELP U`	Offers to undo the last change to the buffer.
`HELP V` *string*	Shows all Zmacs variables containing *string.*
`HELP W`	Prompts for an extended command and shows its keybinding.
`HELP HELP`	Displays these `HELP` key functions.
`HELP SPACE`	Repeats the last `HELP` command.
`SUSPEND`	Starts a Lisp Listener (return from it with `RESUME`).

Zmacs Recovery Facilities

`m-X` Undo	Undoes the last command.
`c-sh-U`	Undo.
`m-X` Redo	Undoes the last undo.
`c-sh-R`	Redo.
`c-Y`	Yanks back the last thing killed.
`m-Y`	After a `c-Y`, successively yanks back older things killed.
`c-sh-Y`	Prompts for a string to yank.

m-sh-Y	After c-sh-Y, successively yanks back older things containing string.

Extended Commands

Extended commands (the m-X commands) put you in a small area of the screen with full editing capabilities (a *minibuffer*) for entering names and arguments. Several kinds of help are available in a minibuffer.

COMPLETE	Completes as much of the current command as possible.
HELP	Gives information about special characters and possible completions.
c-?	Shows possible completions for the command currently being entered.
END or RETURN	Completes the command, and then executes it.
c-/	Does an apropos on what has been typed so far.

Writing Files

c-X c-S	Writes the current buffer into a new version of the current file name.
c-X c-W	Writes the current buffer into a file with a different name.
m-X Save File Buffers	Offers to save each file whose buffer has been modified.

Buffer Operations

c-X c-F	Gets a file into a buffer for editing.
c-X B	Selects a different buffer (prompts; default is the last one).
c-X c-B	Displays a menu of available buffers; lines are mouse-sensitive.
c-X K	Kills a buffer (prompts; default is current buffer).
m-<	Moves to the beginning of the current buffer.
m->	Moves to the end of the current buffer.
c-m-L	Selects the most recently selected buffer in this window.

Character Operations

c-B	Moves left (back) a character.
c-F	Moves right (forward) a character.
RUBOUT	Deletes a character left.
c-D	Deletes a character right.
c-T	Transposes the two characters around point; if at the end of a line, transposes the two characters before point, ht -> th.

Word Operations

m-B	Moves left (back) a word.
m-F	Moves right (forward) a word.

m-RUBOUT	Kills a word left (c-Y yanks it back at point).
m-D	Kills a word right (c-Y yanks it back at point).
m-T	Transposes the two words around point (if only -> only if).
m-C	Capitalizes the word following point.
m-L	Lowercases the word following point.
m-U	Uppercases the word following point.

Line Operations

c-A	Moves to the beginning of the line.
c-E	Moves to the end of the line.
c-N	Moves down (next) a line.
c-O	Opens up a line for typing.
c-P	Moves up (previous) a line.
c-X c-O	Closes up any blank lines around point.
CLEAR INPUT	Kills from the beginning of the line to point (c-Y yanks it back at point).
c-K	Kills from point to the end of the line (c-Y yanks it back at point).

Sentence Operations

m-A	Moves to the beginning of the sentence.
m-E	Moves to the end of the sentence.
c-X RUBOUT	Kills from the beginning of the sentence to point (c-Y yanks it back at point).
m-K	Kills from point to the end of the sentence (c-Y yanks it back at point).

Paragraph Operations

m-[	Moves to the beginning of the paragraph.
m-]	Moves to the end of the paragraph.
m-Q	Fills the current paragraph (see HELP A Auto fill).
n c-X F	Sets the fill column to *n* (example: c-6 c-5 c-X F).

Screen Operations

SCROLL or c-V	Shows next screen.
m-SCROLL or m-V	Shows previous screen.
c-0 c-L	Moves the line where point is to the top of the screen.
c-m-R	Repositions the window to display all of the current definition, if possible.

Search and Replace

c-S *string*	"Incremental" search; searches while you are entering the string; terminate search with END.

c-R *string*	"Incremental" backward search; terminate search with END.
c-S END	Enter String Search. See the section "String Search".
c-% *string1* RETURN *string2* RETURN	Replaces *string1* with *string2* throughout.
m-% *string1* RETURN *string2* RETURN	Replaces *string1* with *string2* throughout, querying for each occurrence of *string1*; press SPACE meaning "do it", RUBOUT meaning "skip", or HELP to see all options; (see HELP C m-%).

Region Operations

c-SPACE	Sets the mark, a delimiter of a region. Move the cursor from mark to create a region. The region is highlighted. Use with c-W, m-W, c-Y and region commands, for example, m-X Hardcopy Region.
c-W	Kills region (c-Y yanks it back at point).
m-W	Copies region onto kill ring without deleting it from buffer (c-Y yanks it back at point).
c-Y	Yanks back the last thing killed.

Window Operations

c-X 2	Splits the screen into two windows, using the current buffer and the previously selected buffer (the one that c-m-L would select).
c-X 1	Resumes single window, using the current buffer.
c-X O	Moves cursor to other window.
c-m-V	Shows next screen of the buffer in the other window; with a numeric argument, scrolls that number of lines — positive for the forward direction, negative for the reverse direction.
c-X 4	Splits the screen into two windows and asks what to show in the other window.

Lisp Syntax for Users of Symbolics FORTRAN

0.0.242. Introduction

Throughout the manual we present Lisp expressions relevant to using Symbolics FORTRA. If you are completely unfamiliar with Lisp, this section explains the Lisp syntax you need to enable you to use this manual. We define the data type *symbol* and the following special characters:

- Parentheses ( )
- Double quote "
- Single quote '
- Backquote ‘ used with a comma ,
- Colon :

0.0.243. Symbol

In addition to the traditional data, numbers, and character strings of other programming languages, Lisp also manipulates symbols. A symbol is a data object with a name and possibly a value. The name of a symbol is either a sequence of letters, numbers, and some special characters, like hyphens. For a discussion of the characteristics of symbols, see the section "Data Types".

Example: **fred**, **december-25**, and **cest-la-vie** are all symbols.

0.0.244. Single quote

A single quote prevents Lisp from evaluating (finding the value of) what follows the quote.

Example: **(print 'fred)** causes the Lisp function **print** to print and return the symbol **fred**, whereas **(print fred)** causes **print** to print and return the *value* of **fred**.

0.0.245. Parentheses

Parentheses enclose the elements of a *list*, as in the following list of three elements.

```
      (red yellow blue)
```

Lists can contain lists, and so the parentheses multiply:

```
      ((8 sourcef) (9 destf))
```

This example shows one list that consists of two lists:

```
      (8 sourcef) and (9 destf)
```

An empty list, one with no elements, is denoted by ( ).

Parentheses must balance — one right parenthesis for every left one. Thus, the following example is balanced.

Example:

```
(defun callfortran (file)
  (eval `(f77:execute ftn-user:iftest :units ((7 ,file)))))
```

The Zmacs editor understands Lisp syntax and helps you to balance parentheses.

In Lisp, an expression is complete as soon as you type the last balancing parenthesis.

0.0.246. Double quote

Double quotes delimit character strings, like file names.

Example: `(:files "schedulex" "scheduley" "schedulez")` lists three file name strings.

0.0.247. Backquote Used with a Comma

Similar to the single quote, the backquote-comma combination tells Lisp that it should not evaluate what follows the backquote until it reaches the comma; then it should evaluate the expression following the comma. The comma is not used as punctuation but instead inhibits the effect of quoting.

Example:

```
(defun test (file)
  (apply 'f77:execute `(iftest :units ((3 ,file)))))
```

Except for **file**, all the code following the backquote (‘) is not evaluated; however, **file** is "unquoted" (evaluated) rather than treated as a literal symbol **file**.

0.0.248. Colon

A colon after a word indicates that the word is a package name.

Example: In **si:fred**, **si** is the name of a package containing a Lisp symbol **fred**.

If no package name precedes the colon, whatever follows the colon belongs to the keyword package. All keyword options to functions belong to the keyword package.

Example:

```
     (ftncopy :units ((8 sourcef) (9 destf)))
```

In this expression **:units** is the keyword option to a function called **ftncopy**.

Sublicense Addendum for Symbolics FORTRAN77

Your purchase of Symbolics FORTRAN77 under the *Terms and Conditions of Sale, License, and Service* (3/89), allows you to use Firewall on a designated processor. Customers who distribute an application that includes Firewall **must** sign a *Sublicense Addendum to the Terms and Conditions of Sale, License and Service* (3/89). This agreement spells out the terms and conditions under which you can sublicense any application that contains FORTRAN77 code. The Sublicense Addendum appears on the next page. If you have not done so already, read the Sublicense Addendum carefully, sign it, detach it, and return it to your Symbolics sales representative.

Note: *You are required to sign the sublicense agreement even if you only distribute your Firewalled application internally — to end users who work for your company.*

Sublicense Addendum to Symbolics Inc. Terms and Conditionsof Sale, License and Service (dated 3/89)

Addendum made this _____ day of __________, 199__, ("this Addendum") to Symbolics, Inc. Standard Terms and Conditions of Sale, License and Service (dated 3/89) dated, 199__, (the "Agreement"), both of which are by and between Symbolics Inc. and Customer. All capitalized terms used in this Addendum, if not defined in this Addendum shall have the meanings assigned to them in the Agreement.

1. **The Software**. The Software to which this Addendum applies is defined as follows:

Model	Description	Price	Discount Category
	SLAN-FORT	Symbolics Fortran	6.1

2. **Right of Sublicense**.

2.1 Section 7 *Software License Terms and Conditions* of the Agreement is hereby incorporated by reference.

2.2 Customer may sublicense all or any portion of the binary code of the Software to Customer's end users provided that:

(i) such Software is part of Customer's application software program sublicensed to such end users;

(ii) the Software and Customer's application software program are licensed by Customer to Customer's end users to run on a Symbolics computer system; and

(iii) Symbolics' copyright and trademark notices required by Symbolics shall not be removed from the Software.

3. **End User**.

3.1 The term "end user" for the purposes of this Addendum shall mean Customer's commercial customers and includes Customer's own internal end users of its application software programs.

CUSTOMER	SYMBOLICS, INC.
______________________________	______________________________
Name	Name
______________________________	______________________________
Title	Title
______________________________	______________________________
(Address)	(Address)