User's Guide to the Pascal Tool Kit

Introduction to the Pascal User's Guide

Scope and Purpose of the Pascal User's Guide

Purpose

This document describes the Pascal Tool Kit 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 code, edit, compile, debug, and run Pascal programs under Symbolics Genera.

Scope

This document provides a conceptual overview of the Tool Kit, with just enough detail to clarify the concepts presented and to allow you to get started using Pascal in the Genera environment. It is not a step-by-step instruction guide or a traditional reference manual. We discuss the following topics:

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

Pascal Reference Manual

Reference information for the Pascal language is available from an on-line version of the *Pascal User Manual and Report*, third edition. This book was written by Kathleen Jensen and Niklaus Wirth and revised by Andrew B. Mickel and James F. Miner. It is reproduced with permission of Springer-Verlag and the authors.

It is listed in the Current Candidates pane of Document Examiner as "Jensen and Wirth -- Pascal User Manual and Report." Section titles of this work are identified by the introductory prefixes *Pascal*: for the user manual portion of the work and *Pascal Rpt*: for the Pascal Report portion.

Standard Pascal

Symbolics supports two dialects of the Pascal language: ISO Pascal and Pascal/VS. As the Tool Kit is based on the language standard, this document does not discuss Pascal itself, except to describe the extensions developed for this implementation.

ISO Pascal is documented in the *British Standards Institution Specification for the Computer Programming Language Pascal, BS 6192: 1982*, which defines the International Standards Organization (ISO) version of Pascal, hereafter called the ISO Standard.

Pascal/VS is documented in the *Pascal/VS Language Reference Manual, International Business Machines Corporation, Program Number 5796-PNQ*, hereafter referred to as the Pascal/VS Standard.

Those users who have little experience with the standard language should refer to one of these two documents. For a discussion of some of the differences between the dialects, and the mechanism which controls the choice of dialect: see the section "Pascal Dialects".

The User and Lisp
The Tool Kit depends heavily on the Genera software environment. Thus, while you can use the Tool Kit knowing relatively little about the Lisp language and Symbolics Genera, it is not possible to enjoy the full benefits of running Pascal under Symbolics Genera without eventually becoming familiar with the Lisp language. 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 the Pascal Tool Kit.

Symbolics Genera References
Users who know little about Symbolics Genera should consult the Genera Documentation Set. Pay particular attention to the following documents:

- 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 tutorial guide to programming in the Symbolics Genera environment: See the section "Program Development Tools and Techniques".
- For a guide to the Symbolics text editor: See the section "Zmacs Manual".
- For information on error handling: See the section "Conditions".

More Help
For a brief explanation of the Lisp syntax encountered in this manual: See the section "Lisp Syntax for Pascal Users".

For those who want classroom instruction, Symbolics offers courses on Lisp programming.

Conventions
This *User's Guide* uses the standard conventions as well as the following additional conventions. see the section "Notation Conventions".

Appearance in document	*Representing*
`lispobject`, `reset`	Pascal reserved words in running text. Pascal program, procedure, function, and package identifiers in running text. Pascal directives and source code in running text.

program quadratic;	Pascal reserved words in examples.

Lisp Syntax

Introduction

This document frequently presents Lisp expressions pertinent to using the Pascal Tool Kit. For those readers completely unfamiliar with Lisp, this section provides just enough explanation of Lisp syntax to enable you to use this manual. This section defines the data type *symbol* and the following special characters:

- Single quote ’
- Parentheses ( )
- Double quote "
- Colon :
- Backquote ‘ used with a comma ,

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 can be a sequence of letters, numbers, and some special characters, like hyphens. For a discussion of the characteristics of symbols: see the section "The Symbol Data Type".

Example: **fred**, **december-25**, and **cest-la-vie** are all symbols.

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**.

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))
```

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 callpascal (file)
  (pascal:execute ’pascal-user:iftest :input file))
```

Zmacs, the Symbolics text 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.

Double quote
Double quotes delimit character strings, such as file names.

Example: `("schedulex" "scheduley" "schedulez")` lists three file name strings.

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, then whatever follows the colon is said to belong to the keyword package. All keyword options to functions belong to the keyword package.

Example:

```
     (pascalcopy :input 'foo)
```

In this expression **:input** is the keyword option to a function called **pascalcopy**.

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 that follows 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.

Introduction to the Pascal Tool Kit

Summary of the Pascal Tool Kit

Components
The Pascal Tool Kit contains the following components.

- A compiler for the two dialects of the Pascal language, as described in the ISO Standard and the Pascal/VS Standard. The Tool Kit implements both the full ISO Pascal Standard and the full Pascal/VS specification, with a few exceptions: See the section "Pascal Dialect Restrictions".

- Several language extensions to Pascal, which are of particular use to Symbolics Genera programmers.

- 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.
- An online documentation set consisting of Symbolics *User's Guide to the Pascal Tool Kit* and the *Pascal User Manual and Report, third edition*, a Pascal reference manual.
- Extensions to Zmacs, the Symbolics Genera text editor, to support Pascal language editing, using the language-specific capability of Zmacs.
- A window-oriented symbolic Debugger, permitting debugging of Pascal code at the source level.

Comparison of the Pascal Tool Kit with Other Implementations

In most computing environments...	*Under Symbolics Genera...*
A file holds only one main program.	A file can hold as many main programs as you wish.
You must compile entire files even if only a small change is made to the code.	Compilation is incremental; you can compile only the function that has changed.
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 resolve problems manually.	The editor processes the compiler warnings and provides commands that help you resolve problems.
A link-and-load step is required for all programs.	Programs are immediately executable after compiling; no separate linking step is needed.
After loading and execution, a program disappears from memory; you must reload the program to rerun it.	Once loaded, you can rerun programs without reloading.
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 can be entered at any time during execution.

Using the Pascal Tool Kit for the First Time

Introduction

Purpose

This chapter enables you to run a simple Pascal program immediately — without having to read through the entire manual first or understand much of the Symbolics Genera environment.

Contents

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

Scope

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

Installing Pascal

Procedure

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

2. You need to indicate where the Pascal system definition is to reside; to do this, create the file SYS.SITE;PASCAL.SYSTEM and type the following attribute list and form in the file.

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

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

3. Load the contents of the Release 5.1 Pascal tape into your file system by typing the following form to a Lisp Listener:

   ```
   (dis:load-distribution-tape)
   ```

4. Once the contents of the tape have been loaded onto your sys host, type:

   ```
   Load System Pascal :query no
   ```

   The Pascal Tool Kit is now ready for use.

Notes

1. To avoid making the Pascal system every time you want to use it, save a world on disk with Pascal already loaded into it. See the Save World command. You can view the online documentation available via Document Examiner by pressing `SELECT D`. Online documentation consists of the *User's Guide to the Pascal Tool Kit* and the *Pascal User Manual and Report.* You might

want to become familiar with the overall structure of the documentation before viewing individual chapters. For example, to read in the *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 the pascal tool kit
```

The Current Candidates window should display the complete list of chapter titles, section titles, and so on. The indentation of the entries suggests the hierarchy of topics. For example, all section titles are aligned and are indented slightly more than chapter titles. Each entry displayed in the Current Candidates window is mouse-sensitive; clicking on any entry brings its associated documentation into the Viewer.

Loading Pascal

Procedure

1. Boot your machine, if necessary.

2. Log in.

3. Look at the *herald*, the multiline message that appears on the screen after booting. If the herald lists the Pascal software: See the section "Using the Editor to Enter a Pascal Example".

   If the herald does *not* list the Pascal software, use the Load System command to load the Pascal Tool Kit, which is called `Pascal`.

   To use Load System, type the following to the Command Processor and press `RETURN`:

   ```
   Load System Pascal :Automatic Answer Yes
   ```

Using the Editor for the First Time

Procedure

1. Invoke Zmacs by pressing `SELECT E`.

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

   Choose a name for the file. 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.

Example: Suppose Fred wants to create a new Pascal source file, quadratic.pascal, that he wants to store in his home directory on a Symbolics host called quabbin (q for short). Fred would type the following:

```
c-U c-X c-F q:>fred>quadratic.pascal
```

Make sure that the name includes the proper Pascal *file type* (extension) for your host; for example, `quadratic.pascal` is the correct name for a file residing on LMFS, the Genera file system. (see the section "Pascal File Types".) The mode line, located below the editor window, displays (`PASCAL`).

3. If the file type was not appropriate, then use `m-X` Pascal Mode to set the buffer mode manually. Press the `META` and `X` keys together. Then type `pascal mode` and press `RETURN`. The mode line displays (`PASCAL`).

   The Pascal Mode command prompts you about whether you wish to set the mode in the attribute list, which specifies the properties of the buffer. Type `y` to create an attribute list, like so:

   ```
   {-*- Mode: PASCAL -*- }
   ```

4. Turn on Electric Pascal mode using `m-X` Electric Pascal Mode. The mode line, located below the editor window, displays (`PASCAL Electric Mode`). *Electric Pascal mode* is a special facility that places input in the appropriate character style and case, depending on the syntactic context into which the input is inserted.

5. Use `m-X` Update Attribute List to specify the properties of the buffer, among them mode, the dialect of Pascal you are using, and the package into which your Pascal program will be compiled. The attribute list is the first line of a file and should look something like:

   ```
   {-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }
   ```

   Note that the dialect attribute is ISO Pascal, which is the default dialect of the Tool Kit.

6. Enter the sample program `quadratic`. Note that as you type the reserved words, they are rendered in an uppercase, boldface character style controlled by Electric Pascal mode. By default, body text is rendered in a lowercase, Roman style.

```
PROGRAM quadratic;
VAR a,b,c,discriminant,x1,x2 : real;
BEGIN
    writeln;
    write('Solves the equation: ');
    writeln('A*X**2 + B*X + C = 0 for X');
    write('Enter values for A, B, and C:  ');
    readln(a,b,c);
    discriminant := b*b - 4*a*c;
    x1 := (-b + sqrt(discriminant)) / (2*a);
    x2 := (-b - sqrt(discriminant)) / (2*a);
    writeln('The roots are:');
    writeln(x1,' and ',x2)
END.
```

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

Check the code in your buffer against the example.

7. If you want to save the source code in a disk file, use `c-X c-S`, that is, hold down the `CTRL` key as you press the `X` and then the `S` keys.

Compilation, Execution, and Error Recovery in the Example

Procedure

1. With the cursor positioned within the program text, use `c-sh-C` to compile the program. Ignore any compiler warnings about base and syntax attributes. If the compilation completes successfully, the minibuffer displays "program quadratic compiled". Go to step 3. If the compilation halts before completion, the typeout window at the top of the screen displays compiler warnings, errors, or both. Go to step 2.

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

   If the typeout window displays compiler warnings, press `ABORT`, 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:
   ```
   (pascal:execute 'quadratic)
   ```

Remember to type the parentheses.

5. The program prompts:
   ```
   Solves the equation: A*X**2 + B*X + C = 0 for X
   Enter values for A, B, and C:
   ```
   Type:
   ```
   1 0 -9 RETURN
   ```

   The window should display the following:
   ```
   The roots are:
    3.000000000000E+000 and -3.000000000000E+000
   ```
   nil

6. Rerun the program, this time causing a run-time error. When the program prompts for input, type:
   ```
   1 2 3 RETURN
   ```

   Genera automatically invokes the Debugger (identifiable by Error:), which displays a descriptive error message ("Attempt to take the square root of -8.0d0, which is negative"), an error message, and then a list of suggested actions and their outcomes. When executing a Pascal program, the first such action is always "Enter the display debugger". See figure ! .

   For example, pressing ABORT or s-D from the Debugger causes you to return to the editor breakpoint, from which you can rerun quadratic.

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

8. This step completes the edit-compile-error-recovery cycle for the sample program. It is recommended that you spend some time editing the code, recompiling, and rerunning the program until you feel comfortable with the process. Then read about the Tool Kit's extensions: See the section "Extensions to Pascal".

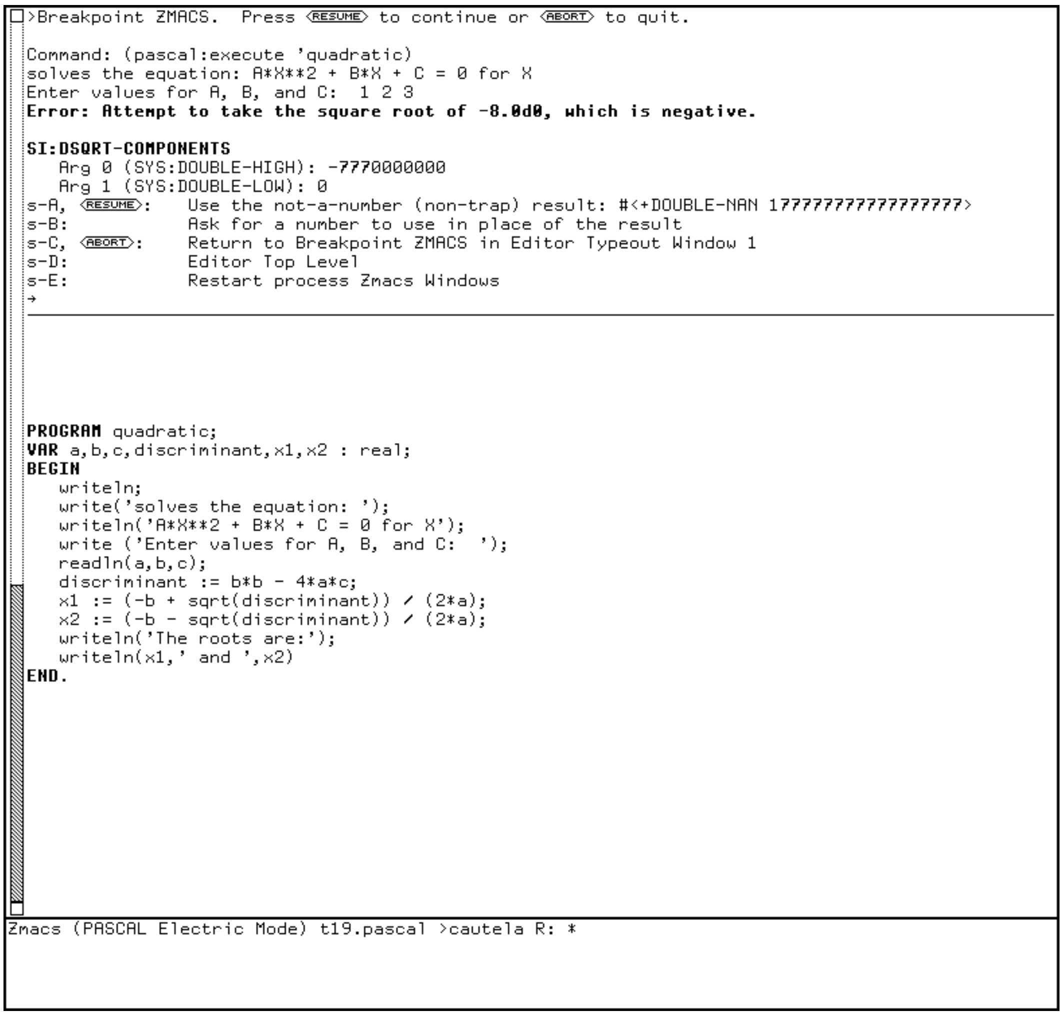
Pascal Dialects

Introduction

Purpose

The Tool Kit implements the full ISO Pascal Standard, with extensions specially designed for the Symbolics Genera environment. The full Pascal/VS specification has been implemented with a few exceptions. Several differences distinguish the dialects, and this chapter serves to:

- define the mechanism for selecting a dialect.



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

Command: (pascal:execute 'quadratic)
solves the equation: A*X**2 + B*X + C = 0 for X
Enter values for A, B, and C:  1 2 3
Error: Attempt to take the square root of -8.0d0, which is negative.

SI:DSQRT-COMPONENTS
   Arg 0 (SYS:DOUBLE-HIGH): -7770000000
   Arg 1 (SYS:DOUBLE-LOW): 0
s-A, RESUME:   Use the not-a-number (non-trap) result: #<+DOUBLE-NAN 17777777777777777>
s-B:           Ask for a number to use in place of the result
s-C, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window 1
s-D:           Editor Top Level
s-E:           Restart process Zmacs Windows
→
```

```
PROGRAM quadratic;
VAR a,b,c,discriminant,x1,x2 : real;
BEGIN
   writeln;
   write('solves the equation: ');
   writeln('A*X**2 + B*X + C = 0 for X');
   write ('Enter values for A, B, and C:  ');
   readln(a,b,c);
   discriminant := b*b - 4*a*c;
   x1 := (-b + sqrt(discriminant)) / (2*a);
   x2 := (-b - sqrt(discriminant)) / (2*a);
   writeln('The roots are:');
   writeln(x1,' and ',x2)
END.
```

```
Zmacs (PASCAL Electric Mode) t19.pascal >cautela R: *
```

Figure 72. Producing a run-time error in the example program.

- describe some of the differences and similarities between the dialects.
- describe the restrictions on each dialect.
- define the constraints on interaction between routines compiled as ISO Pascal versus those compiled as Pascal/VS.

Selecting a Pascal Dialect

Dialect Entry in Attribute List

Each Pascal file or buffer to be compiled is associated with one of the two Pascal dialects, either ISO Pascal or Pascal/VS. The choice of dialect is shown in the *dialect entry* of the file attribute list.

The format of the Dialect entry is `Dialect:` *dialect-name*, as in the attribute list:

```
{-*- Mode: PASCAL; Dialect: ISO; ...  -*- }
```

Thereafter, whenever you read in this file, the buffer will be compiled in ISO Pascal.

Note that the dialect entry (indeed the entire attribute list) is not required; however, whether or not it is present, your buffer or file is still associated with one dialect of Pascal.

The Default Dialect
The default dialect is specified by the **pascal::*pascal-default-dialect*** compiler option. The default value of the option is ISO Pascal.

For example, if you create a new Pascal buffer the attribute list might look like the following:

```
{-*- Mode: PASCAL; Dialect: ISO; ... -*- }
```

Unless you explicitly change the dialect entry in the attribute list, the dialect associated with the buffer is the default dialect. If you change the default, then a file without an attribute list might be compiled in the new default dialect when read into a buffer in a subsequent work session. Files with attribute lists are always compiled in the dialect specified in the dialect entry, regardless of the current default. Changing the default dialect affects only future buffers and existing files that do not contain dialect entries in their attribute lists. For more information:

See the section "Compiler Option: Setting the Default Pascal Dialect".

Changing the Dialect Entry
To change the dialect entry in the attribute list, use `m-X` Set Dialect, or manually change the dialect yourself. If you manually change the dialect entry, remember to reparse the attribute list using the Reparse Attribute List command, `m-X` Reparse Attribute List. Reparsing causes the changes to take effect.

Differences and Similarities: ISO Pascal and Pascal/VS

Differences
In general, every facility in ISO Pascal exists in Pascal/VS, with the sole exception of conformant array parameters. Pascal/VS, on the other hand, contains many features not present in ISO Pascal.

- Only Pascal/VS contains true strings; their maximum length is determined at compile-time, their current length at run-time. A complete library of useful string-manipulation facilities are provided as intrinsic functions.

- Only Pascal/VS contains several loop exit facilities called `leave` and `continue`.

- Only Pascal/VS supports random access input/output, via the `seek` predeclared procedure, and reading and writing of the same file, via the `update` file opening procedure.

Similarities
The Tool Kit has extended ISO Pascal in several ways that make it more similar to Pascal/VS.

- Both Pascal/VS and ISO Pascal support both single-precision `shortreal` and double-precision `real` data types. Whereas ISO Pascal has always included the `real` double-precision data type, the Tool Kit has extended the dialect to include `shortreal` as well.

- Because Pascal/VS and ISO Pascal support `real` and `shortreal` data types, both dialects have also been extended to include two conversion functions, `single` and `double`, to convert from one data type to the other.

- The Tool Kit has extended ISO Pascal to include the `otherwise` clause for `case` statements and variant `record` type definitions. Now both dialects support the `otherwise` alternative.

- Pascal/VS allows file open options to be passed with any file-opening predeclared procedure. The Tool Kit extends ISO Pascal to support the same file open options as Pascal/VS. see the section "Second Parameter to File-Opening Procedures".

Pascal Dialect Restrictions

Pascal/VS Restrictions

In general, the Pascal/VS dialect is implemented as described in the Pascal/VS Standard, and compatibly extended in the Tool Kit. However, a few standard features have not been implemented.

The following Pascal/VS predefined routines have not been implemented.

```
parms
pdsin
pdsout
retcode
token
```

With the exception of `%include`, the `%` feature of Pascal/VS has not been implemented. `%include` provides for compiler directives.

Pascal/VS defines a separate compilation facility, which the Tool Kit has not implemented. It is therefore expected that you will use the Tool Kit's units facility for Pascal. Accordingly, the following Pascal/VS declarators have not been implemented.

```
def
ref
segment
space
static
```

The following Pascal/VS routine directives have not been implemented.

```
external
fortran
main
reentrant
```

Additionally, you cannot use a range as a `Case` selector.

Symbolics Genera supports a family of namespaces for routine and variables names; this is called the *package system*. Normally, you specify a namespace (package) that does not inherit symbols from the Lisp environment, so no conflict is possible between Pascal names and Lisp names. However, if you use the default Genera package, it is possible that you will assign a variable the name of a predeclared Lisp symbol like **t**, or a routine the name of a Lisp function like **print**. In this case your Pascal programs cannot use the names of predefined Lisp symbols.

ISO Pascal Restrictions

In general, the ISO Pascal dialect is implemented as described in the ISO Standard, and compatibly extended in the Tool Kit. However, a very few restrictions or incompatibilities exist in the implementation.

- The Tool Kit extends ISO Pascal to include the `otherwise` alternative for `case` statements and variant `record` type definitions. Note that this makes the ISO Pascal identifier `otherwise` into a reserved word, which might have visible effects on strict ISO Pascal programs.

- Symbolics Genera supports a family of namespaces for routine and variables names, which is called the *package system*. Normally, the Pascal programmer specifies a namespace (package) that does not inherit symbols from the Lisp environment, so no conflict is possible between Pascal names and Lisp names. However, if you use the default Genera package, **user::cl-user**, it is possible that you will assign a variable the name of a predeclared Lisp symbol like **t**, or a routine the name of a Lisp function like **print**. In this case your Pascal programs cannot use the names of predefined Lisp symbols.

Dialect Constraints

In general, programs compiled under Pascal/VS and ISO Pascal can freely call each other and share data. The only restriction is that you cannot pass a `string` to ISO Pascal routines, since ISO Pascal does not support the Pascal/VS `string` data type. In addition, since Pascal/VS does not support ISO Pascal conformant array parameters, it is not valid to pass a conformant parameter to a Pascal/VS routine.

Extensions to Pascal

Summary

Twelve Extensions

The Tool Kit defines several extensions to standard Pascal. Most of these result from the strong hardware data-type checking provided by Symbolics Genera. This feature offers a greatly increased ability to detect errors that on conventional systems would be discovered only by more laborious means, if at all.

The Tool Kit supports the following language extensions:

- Arbitrary-precision integers
- Detection of uninitialized variables
- Strong data-type checking
- Floating-point data type called `shortreal` (extension to ISO Pascal)
- `otherwise` clause (extension to ISO Pascal)
- `%include` compiler directive (extension to ISO Pascal)
- Floating-point conversion functions called `single` and `double`
- Syntactic extensions
- Extensions for interacting with Lisp functions
- File-opening parameters
- Pascal package system
- Separate compilation units

Discussion of Extensions

Arbitrary-Precision Integers

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

```
PROGRAM factor;
VAR x : integer;
  function factorial (n : integer) : integer;
  VAR i : integer;
  BEGIN
      factorial := 1;
      FOR i := n DOWNTO 1 DO
        factorial := factorial * i
  END; { Function factorial }
BEGIN
  writeln;
  write ('Program calculates the factorial of a number. ');
  writeln ('Enter 0 to halt program.');
  REPEAT
    write ('Find the factorial of what number? ');
    readln(x);
    writeln('Factorial ',x:1,' is ',factorial(x):1)
  UNTIL x = 0;
  writeln ('Leaving factorial program.')
END.
```

In Pascal on conventional machines, this routine would work properly until the product computed in the variable factorial overflowed. Then either (1) a hardware overflow would be signalled or (2) the computation would deliver the incorrect answer with no warning whatsoever. In Symbolics Genera, however, the computation completes and returns the correct answer independent of the value of n.

Since the ISO Standard does not restrict the range of precision of integers (see the ISO Standard, section 1.3.2), the accommodation of bignums should provide great flexibility to the programmer who needs to develop code for machines with differing word sizes. It is also helpful to the programmer who is solving mathematical problems, since integers do not exhibit the anomalous overflow behavior of conventional machines.

In conjunction with arbitrary-precision integers, the Tool Kit supports formatted input/output of large integers. Hence, integer format widths such as 200 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

The Tool Kit detects uninitialized data, so that an error condition results if a variable is used before it is assigned a value.

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

For those who do not wish to use this feature, a compiler option exists that initializes all variables to zero. See the section "Compiler Option: Initializing Pascal Program Data".

Strong Data-Type Checking

Symbolics computers provide strong hardware data-type checking among `integer`, `shortreal`, and pointer. Thus, the hardware prevents a program error due to Pascal incorrect variant record selection, for example, in the case where the exponent and mantissa of a `real` number might be interpreted as an `integer`. You can subvert this data-type checking by calling Lisp functions to take the word apart.

This strong data-type checking does not extend to the `real` data type, which is internally represented by a Lisp double-precision floating-point number. The hardware representation for `real` data is in fact a pair of integers.

`shortreal` Data Type

ISO Pascal has been extended to include the single-precision floating-point type `shortreal`. As a result, both Pascal/VS and ISO Pascal support two floating-point data types: `real`, corresponding to double-precision (range ˜2.2 x 10^{-308} to 1.8 x 10^{308}, with accuracy to 53 bits), and `shortreal`, corresponding to single-precision (range ˜1.175 x 10^{-38} to 3.4 x 10^{38}, with accuracy to 24 bits).

Users who wish to have the default `real` data type be single-precision floating point can include the following type declarations in their programs:

type `double = real;`
type `real = shortreal;`

`otherwise` Clause

The Tool Kit has extended ISO Pascal to include the `otherwise` clause for `case` statements and variant `record` type definitions. Now both dialects support the `otherwise` alternative. Note that this makes the ISO Pascal identifier `otherwise` into a reserved word, which might have visible effects on strict ISO Pascal programs.

`%include` Directive

The Tool Kit has extended ISO Pascal to include the `%include` facility, standard in Pascal/VS. For both ISO Pascal and Pascal/VS the allowed formats are:

`%include` *identifier*
and
`%include` *character-string*

Anthing that follows *identifier* or *character-string* on the same line is ignored.

The included file is found by defaulting the string or identifier name against the name of the file containing the `%include`. Lines of compiler input are taken from the included file until EOF is reached, at which time compilation continues with the line following the `%include` directive.

Conversion Functions

Pascal/VS and ISO Pascal have been extended to include two floating-point types, `real`, corresponding to double-precision, and `shortreal`, corresponding to single-precision. To convert between these floating-point formats, the conversions `single` and `double` are provided. The function `single` takes a double-precision (`real`) value as its argument and returns the single-precision number closest to the double-precision input. Conversely, the function `double` takes a single-precision (`shortreal`) value as its argument and returns the double-precision number closest to the single-precision input.

Length of Symbolic Names

Pascal identifiers can have any length, but must fit on a single line, like all Pascal tokens. For example, the following is a valid statement:

```
program quadratic_solution_to_equations;
```

Interaction with Lisp

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

Analogous to `forward`, the new Pascal routine directive `lisp` allows the declaration of an existing Lisp function that you can call from a Pascal routine. See the section "`lisp`: Pascal Routine Directive".

Second Parameter to File-Opening Procedures

The Pascal Tool Kit permits a second parameter for each of the predefined procedures that open files, including `reset` and `rewrite` for ISO Pascal, and `reset`, `rewrite`, `update`, `termin`, `termout`, `pdsin`, and `pdsout` for Pascal/VS. The second parameter must be a string literal, a packed array of `char`, or a Pascal/VS `string`. The parameter consists of two options, one specifying the pathname of the file to be opened; the other, the type of file access. The format of the second parameter is a comma-separated sequence of options and values, like so:

option-name = *option-value*, ...

Spaces are ignored except within file names.

Option	*Value*

name or *ddname*	Specifies a string that corresponds to the pathname of the file to be opened. This pathname is merged with the current pathname default, possibly modified by the **:pathname-defaults** main program option, at file open time.
access	Specifies the type of file access. The option has two possible values: sequential and direct. The default is sequential, except when the Pascal/VS `update` procedure is used; in this case, the default is direct. Also note that the Pascal/VS predeclared procedure `seek` requires that you specify file access as direct.

Example 1: To open a file `f`, whose name is "s:>scald-ii>foo.data", for sequential access reading, type:

```
reset(f, 'name = s:>scald-ii>foo.data');
```

Example 2: To open a file `f`, whose name is "vixen:/ufs/comp/bar.direct", for direct access writing, type:

```
rewrite(f, 'name=vixen:/ufs/comp/bar.direct, access=direct');
```

Pascal Package System

Background. Symbolics computers are a large-scale virtual-memory, single-user computer; many programs — the editor, the compiler, and so on — coexist in the same environment (address space). Once Pascal routines (and Lisp functions, too) are loaded into the Genera environment, they remain there until replaced by recompilation or until the machine is *cold booted*, that is, until a fresh version of Genera is loaded.

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

The package name prefixes two identically named functions.

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

Packages in Pascal. The Pascal package facility differs from Lisp packages in one important way: Pascal packages do not conflict with symbols in any Lisp packages or in any other Pascal packages.

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

For instructions on defining your own Pascal packages: See the section "Declaring a Pascal Package". For information on assigning package names to buffers: See the section "Editing Basics for Pascal Programs". For a general discussion of the Lisp package facility: See the section "Packages".

Separate Compilation Units
The Pascal Tool Kit implements its own units facility in lieu of the separate compilation facility defined by Pascal/VS. The units facility provides a mechanism for any routine to call separately compilable units that reside in the same or different files. Hence, code needed by many programs is available for general use. The declaration scope of the calling routine "expands" to include the declarations made in the called unit.

See the section "Units Facility".

Using the Editor to Write Pascal Programs

Editing Basics

Zmacs and Pascal
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, Pascal editor mode incorporates many helpful editor extensions that understand Pascal syntax. For example, there are commands for moving from one Pascal language unit to another, and commands that locate syntax errors. An editor minor mode, called Electric Pascal mode, places typed input in the appropriate character style and case. Electric Pascal mode is accessible from Pascal editor mode.

For more information:

- See the section "Editor Extensions for Pascal".
- See the section "Summary of Standard Editing Features".
- For a discussion of the HELP key and command *completion*: See the section "Getting Help".

Procedure
This procedure summarizes the steps for preparing an editor buffer for writing or editing Pascal source code.

1. Invoke 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 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.

   Make sure that the name includes the proper Pascal *file type* (extension) for your host; for example, quadratic.pascal is the correct name for a file residing on LMFS, the Genera file system. See the section "Pascal File Types". Example: Suppose Fred wants to create a new Pascal source file, cube.pascal, that he wants to store in his home directory on a Symbolics Lisp Machine host called quabbin (q for short). Fred would type the following:

   ```
   c-U c-X c-F q:>fred>cube.pascal
   ```
   The mode line, situated below the editor window and near the bottom of the screen, displays (PASCAL).

3. Whenever you specify a file name with the correct Pascal file type (file extension), the editor automatically sets the mode of the buffer to Pascal and sets the buffer dialect to either ISO Pascal or Pascal/VS, whichever is your preferred default.

   If the editor did *not* set the mode of the buffer to Pascal, use m-X Pascal Mode to set the mode manually. This situation might occur if the buffer was created in such a way that the editing mode is not implicit, for example, if you entered a pathname with a non-Pascal file type or you created a buffer not associated with a file.

   The Pascal Mode command prompts you about whether you wish to set the mode in the attribute list as well as the mode line. The attribute list specifies the properties of the buffer; the Pascal Mode command specifies the mode property in particular. For example, if you type y, it creates an attribute list, like so:

   ```
   {-*- Mode: PASCAL -*- }
   ```

   When the editor sets the mode to Pascal automatically, most users create the attribute list with m-X Update Attribute List, rather than the Pascal Mode command. The Update Attribute List command sets many of the properties of the buffer at once, among them:

   - The editor mode
   - The dialect of Pascal you are using
   - The package into which your Pascal program will be compiled

   The attribute list is optional but if present must be the first line of a file. A sample attribute list might look like the following:

   ```
   {-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }
   ```

   Alternatively, you can type the entire attribute list or any part of it yourself.

   For more information on the attribute list: See the section "File Attribute Lists".

4. If the dialect listed in the attribute list is not the one you want, then use `m-X` Set Dialect to change it. Currently, the valid values of the Dialect attribute are iso and vs. Alternatively, you can manually enter the dialect name in the attribute list by typing `; Dialect:` and either `iso` or `vs`.

   (The Tool Kit default is ISO Pascal. For information on changing the default: See the section "Compiler Option: Setting the Default Pascal Dialect".)

5. When no package is specified in the attribute list, the default package is **pascal-user**, the standard Tool Kit 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 yet 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 the special form **pascal::package-declare-with-no-superiors**.

   Once the package is set in the attribute list, Pascal routines in the file are defined as belonging to an existing package whenever the file is read into an editor buffer. 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. Invoke `m-X` Electric Pascal Mode to turn on Electric Pascal mode, which is a special editor facility that places input in the appropriate character style and case, depending on the syntactic context into which the input is inserted.

   By default the Tool Kit renders Pascal reserved words in an uppercase boldface character style, body text in a lowercase Roman style, and comments in mixed-case italics. To change the defaults, invoke `m-X` Adjust Face and Case.

   To turn off the mode if it is on, reissue the command.

8. 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. Remember that you can have as many main programs in a file as you wish.

Setting Editor Defaults

Introduction

If you intend to write Pascal code most of the time, you might want to set editor defaults in your init file for

- Major mode

- Package

- Dialect

Major Mode

By setting the major mode to Pascal, you will not need to specify the file type when creating a new file buffer. Example: The following is sufficient for creating a Pascal source file:

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

To set the default major mode to Pascal, type the following form in your init file:

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

Package

If you plan to compile the majority of your Pascal mode in a particular package (other than the default **pascal-user**), you might want to set the default package in your init file.

To set the default package when the Pascal system is loaded and the package does *not* exist, type:

```
      (setf zwei:*default-package*
            (pascal:package-declare-with-no-superiors
              "package-name"))
```

and supply a value for *package-name*.

To set the default package when the Pascal system is loaded and the package does *not* exist, type:

```
      (setf zwei:*default-package*
            (find-package "package-name"))
```

and supply a value for the string *package-name*.

Dialect

The default dialect of the Tool Kit is ISO Pascal. If you write code in Pascal/VS, then you might want to set the default in your init file, as follows:

```
      (login-forms
        (setq pascal:*pascal-default-dialect* :vs))
```

Add this form to your init file only when (1) your Pascal system is stored on disk and is therefore accessible at login or (2) your init file loads Pascal.

Pascal Editor Mode

A Syntax-directed Editor

The Pascal editor mode extension to Zmacs is based on a syntax-directed editor. The syntax-directed editor understands the syntax of Pascal and makes use of this knowledge to provide language-specific commands and information while editing, for example, indicating the location of the next syntax error in the buffer.

The Pascal editing commands operate on Pascal language units (procedure or function definitions and statements) and on language tokens (for example, comments and identifiers) and expressions. This means, in effect, that the syntax-directed editor understands how to distinguish one unit from another.

In addition to Zmacs textual model of editing, the syntax-directed editor provides the features of a structure or template editor as well. Unlike many structure editors, the syntax-directed editor does not restrict the size or the legality of the contents of the buffer. However, the more syntactically correct a program, the more helpful the editor.

Relation to Zmacs

The syntax-directed editor provides the standard commands and capabilities of Zmacs that are applicable to Pascal. For example, `c-N` moves the cursor to the next line in both Lisp mode and Pascal mode as well as in text mode buffers.

One fine but crucial difference is that Zmacs commands that operate on Lisp forms in a Lisp mode buffer operate on statements and larger language-specific constructs (like procedures) in Pascal modes. Separate commands operate on language expressions; others exhibit even more refined behavior, such as deleting a Pascal language token or finding Pascal syntax errors.

Where possible, the Pascal editor mode commands are modelled on their analogous Lisp mode commands. For example, in Lisp mode `c-m-RUBOUT` deletes the previous Lisp form; in Pascal mode the same command deletes the previous Pascal language statement or definition.

Moving the Cursor

Introduction

The cursor movement commands operate on Pascal statements, definitions, and expressions. How the cursor moves is determined by its current position and the specific command you execute.

The examples in this section refer to the following program.

```
1       PROGRAM factor;
2       { program calculates the factorial of any number. }
3       { Halts when the user types 0. }
4       VAR x : integer;
5          FUNCTION factorial (n : integer) : integer;
6          VAR i : integer;
7          BEGIN
8             factorial := 1;
9             FOR i := n DOWNTO 1 DO
10               factorial := factorial * 1
11         END; { function factorial }
12
13      BEGIN { main program factor }
14         writeln;
15         writeln('Program finds factorial.  Type 0 to halt program.');
16         REPEAT
17             write('Find the factorial of what number? ');
18             readln(x);
19             writeln('Factorial ',x:1,' is ', factorial(x):1)
20         UNTIL x = 0;
21         writeln('Leaving factorial program')
22      END. { main program factor }
```

Moving by Statement

- `c-m-F`

  Moves the cursor forward to the end of the current or next unit.

  Example: Placing the cursor on line 19 and pressing `c-m-F` moves the cursor to line 20, at the semicolon.

- `c-m-B`

  Moves the cursor backward to the beginning of the current or previous unit.

  Example: Placing the cursor on line 20, at the semicolon, and pressing `c-m-B` moves the cursor to line 19, to the beginning of the statement `writeln('Factorial ',x:1,' is ', factorial(x):1)`.

  Example: Placing the cursor on line 20, *after* the semicolon, and pressing `c-m-B` moves the cursor to line 16, to `repeat`.

Moving by Definition

- `c-m-A`

  Moves the cursor backward to the beginning of the current or previous procedural or functional definition.

- `c-m-E`

Moves the cursor forward to the end of the current or next procedural or functional definition.

Example: Placing the cursor on line 4 and pressing `c-m-E` moves the cursor to the end of line 22.

Example: Placing the cursor on line 5 and pressing `c-m-E` moves the cursor to line 11, after `end;`.

- `c-m-H`

  Marks the current language definition as a region and moves the cursor to the beginning of the definition. Used in conjunction with a command to compile, delete, or yank the region.

Moving by Expression

- `c-sh-F`

  Moves the cursor forward to the end of the current or next expression.

  Example: Placing the cursor at the token `for` on line 9 and pressing `c-sh-F` moves the cursor to the token `downto` on the same line.

- `c-sh-B`

  Moves the cursor backward to the beginning of the current or previous expression.

Blinking Reserved Words

`m-X` Blink Matching Construct enables you to check that block constructs are balanced. When the feature is turned on and the cursor is positioned at a reserved word that closes a block statement, the editor flashes the reserved word that opens the block statement. For example, positioning the cursor anywhere after the first letter in `end if` makes the matching `if` construct blink. Invoke the command again to turn this feature off (the default condition). Note: This command works only when the buffer is syntactically correct.

Deleting Language Units

Introduction

The deletion commands operate on Pascal contructs (statements and definitions), tokens, and expressions. What text is deleted is determined by the current position of the cursor and the specific command you execute. To retrieve just-deleted text, use the standard yanking commands, such as `c-Y`.

Deleting Statements and Definitions

The following three deletion commands operate only on language units and expressions. They understand the notion of nested statements.

Example: Consider lines 5-11 of the example.

```
3               .
4               .
5          FUNCTION factorial (n : integer) : integer;
6          VAR i : integer;
7          BEGIN
8             factorial := 1;
9             FOR i := n DOWNTO 1 DO
10               factorial := factorial * 1
11         END; { function factorial }
```

- `c-m-K`

  Deletes forward to the end of the current or next statement. When you attempt to delete a statement or definition that contains a syntax error, the editor marks the region in inverse video and queries you about whether or not to go ahead with the deletion.

Cursor is ...	*Effect*
Anywhere on line 8	Deletes `factorial := 1`, up to semicolon
On line 9, from left margin to end of `for`	Deletes entire `for` statement, lines 9-10
On line 7, from left margin to end of `begin`	Deletes lines 7-11, up to semicolon
On line 9, anywhere after `for`	Signals warning: No balanced language construct to delete

- `c-m-RUBOUT`

  Deletes backward to the beginning of the current or previous statement.

- `c-sh-K`

  Deletes the statement or definition around the cursor.

Cursor is ...	*Effect*
Anywhere on line 8	Deletes `factorial := 1`, up to semicolon
Anywhere on line 9	Deletes entire `for` statement, lines 9-10
On line 7, from left	Deletes lines 7-11, up to semicolon

margin to end of `begin`

Deleting Expressions
As a precaution, deletion commands do not operate on unbalanced expressions.

- `c-sh-X`

  Deletes forward to the end of the current or next expression.

  Example: Consider the expression `while (A + B +(B+ B ))` do. Invoking `c-sh-X` on this expression has different effects depending on the position of the cursor.

Cursor is at ...	*Effect*
First open parenthesis	Deletes expression: `(A + B +(B+ B ))`
Identifier A	Deletes everything but the outer balancing parentheses
Any plus sign	Signals warning: No balanced language construct to delete
Second open parenthesis	Deletes `(B+ B )`

- `m-sh-X`

  Deletes backward to the beginning of the current or previous expression.

Deleting Tokens

- `c-sh-T`

  Deletes forward to the end of the current or next token.

- `m-sh-T`

  Deletes backward to the beginning of the current or previous token.

Finding Syntax Errors

Introduction
Usually you become aware of syntax errors only when you try to compile a unit of source code. The syntax-directed editor, however, parses source as you type it and keeps track of all syntax errors. However, the editor notifies you of such errors only in certain circumstances:

- You explicitly query about syntax errors using either `c-sh-N` or `c-sh-P`.
- Some compilation commands, for example, `c-sh-C`, notify you when syntax errors are encountered.
- Deletion commands like `c-m-K` and `c-m-RUBOUT` mark the region containing a syntax error in inverse video and query you about whether or not to proceed with the deletion.
- You press `LINE` after entering a line of source code.

`c-sh-N` and `c-sh-P`

- `c-sh-N`

  Finds the nearest syntax error to the right of the cursor, if any, and moves the cursor there. With a numeric argument, it finds the last syntax error in the buffer.

- `c-sh-P`

  Finds the nearest syntax error to the left of the cursor and moves the cursor there. With a numeric argument, it finds the first syntax error in the buffer.

Sometimes a single error can result in a cascade of error messages from `c-sh-N` or `c-sh-P`. In such cases, correct the errors starting with the first error.

`LINE`
In addition to indenting the current line correctly with respect to the line above it, `LINE` also detects syntax errors within that line, indicating in the minibuffer the point of error, as in:

```
j := k + ;
         ^
```

Introduction to Pascal Mode Completion and Templates

Introduction

Pascal editor mode provides a general completion facility over the set of Pascal language constructs as well as over the set of predeclared identifiers and reserved words. For example, as soon as you type enough characters in a Pascal reserved word or predeclared identifier so the word is recognized as unique, you can ask for completion. The remaining characters of the identifier are inserted in the buffer. If the word is not recognized as unique, you can ask to see all possible completions to what you have typed.

The completion facility can also insert *templates*, which show patterns of the syntactic constructs of Pascal. For example, below is the template for the `for` construct.

```
FOR ⊂control variable⊃ := ⊂initial value⊃ ⊂to or downto⊃ ⊂final value⊃ DO
    ⊂statement⊃
```

Example
For the purposes of explaining completion and templates, assume that we are entering the following procedure:

```
PROGRAM factor;
{ Program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
   FUNCTION factorial (n : integer) : integer;
   VAR i : integer;
   BEGIN
         factorial := 1;
         FOR i := n DOWNTO 1 DO
            factorial := factorial * 1
   END; { function factorial }

BEGIN { main program factor }
   writeln;
   writeln('Program finds factorial.  Type 0 to halt program.');
   REPEAT
       write('Find the factorial of what number? ');
       readln(x);
       writeln('Factorial ',x:1,' is ', factorial(x):1)
   UNTIL x = 0;
   writeln('Leaving factorial program')
END. { main program factor }
```

Pascal Mode Templates

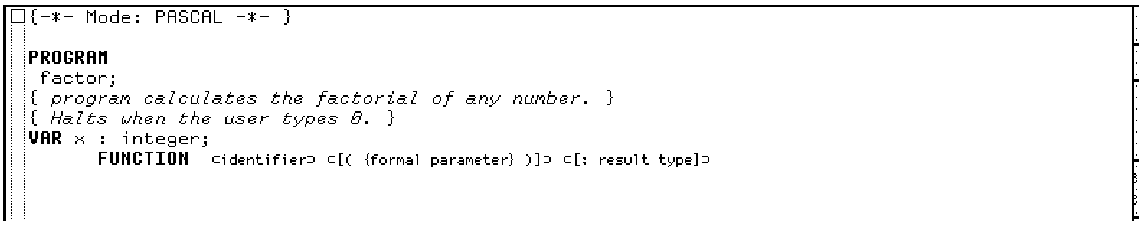
Inserting a Template with `END`
Assume that the current buffer looks as follows and that the cursor is positioned after the keyword `function`.

```
{ -*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*-}

PROGRAM factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
    FUNCTION
```

When you position the cursor after a reserved word, like `function`, pressing `END` inserts the syntactic pattern (template) of the appropriate language structure into the buffer. The template consists of some combination of descriptive items and Pascal keywords.

Where a single valid construct is possible, the `END` key inserts the template that matches the keyword. Where multiple possibilities exist, `END` pops up a menu of template items for all the constructs that are valid in the current context.

Example: Pressing END after **FUNCTION** inserts a template, as shown in the lower portion of figure !.



```
{-*- Mode: PASCAL -*- }

PROGRAM
 factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
      FUNCTION  ⊂identifier⊃ ⊂[( {formal parameter} )]⊃ ⊂[: result type]⊃
```

Figure 73. After pressing END and selecting a template.

Template Items

In addition to Pascal reserved words, the template contains *template items*, which are syntactic constructs surrounded by horseshoes.

These template items contain constructs that are either required, optional and repeating, or repeating. The delimiters of the template item indicate the type of construct it describes, in accordance with extended Backus-Naur form.

Type of template item	*Delimiter*
⊂{ optional repeating }⊃	curly brackets, or braces
⊂[optional]⊃	square brackets
⊂ required ⊃	horseshoes only

Example: ⊂{, identifier}⊃ represents an optional, repeating identifier. You can omit or expand it into the following: , identifier {, identifier }.

Moving Among Template Items

You can move from one template item to another by using the commands c-m-N, which moves to the beginning of the next item, and c-m-P, which moves to the beginning of the previous item.

Filling in a Template Item

A template item is just text, in the sense that you can write out or read in a file containing template items. The editor treats (parses) a template item as what it represents; for example, ⊂identifier⊃ would act as though an identifier were present. However, unlike regular text, the template item disappears when you position the cursor at the item and type a character into the buffer.

Example: Type the name of the procedure, such as factorial. Note that typing the first character causes the template item ⊂identifier⊃ to disappear. Now type c-m-N to position the cursor in front of the next template item, ⊂[formal part]⊃. When you press SPACE at any template item, that item disappears, and the blank is *not* inserted. c-sh-T and m-sh-T also provide the same capability. These commands provide a convenient way of handling optional syntax.

The c-HELP Command

Getting Help Information

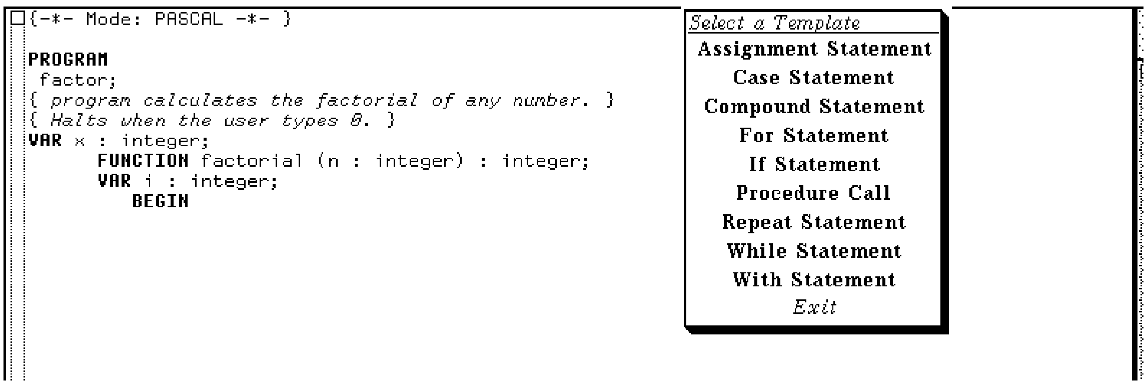
The c-HELP command displays a mouse-sensitive menu of all language constructs valid at the cursor position. The c-HELP mechanism understands the syntax of Pascal and works by comparing its understanding with the relative position of the cursor in the buffer. If the cursor is positioned between the begin and end reserved words, the c-HELP command displays a menu of all valid executable statements rather than declarative statements.

Because the c-HELP command relies heavily on language context, its behavior might is altered by incompleteness or syntax errors in your program. In general, the more accurate and complete your program, the more accurate and useful the help information.

In some contexts help information is not available. In such cases, pressing c-HELP yields only the message No help available here.

In general the c-HELP command is useful for finding out about which language constructs are legal at the cursor position, whether at a template item or a reserved word.

Example: Pressing c-HELP at BEGIN, displays a menu of all executable statements. See figure !.



```
{-*- Mode: PASCAL -*- }

PROGRAM
 factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
      FUNCTION factorial (n : integer) : integer;
      VAR i : integer;
         BEGIN
```

Figure 74. After pressing c-HELP at BEGIN.

Select a menu item by clicking the Left, Middle, or Right button on the mouse. Click Left, Middle, or Right on Exit to return to editing.

Mouse click	*Meaning*
Left	Inserts the selected template into the buffer at the cursor position.
Middle	Displays the selected template in a temporary window. Move off the window to make the template disappear.

Right	Displays documentation from the *Pascal User Manual and Report* for the selected topic. The documentation is displayed in a Document Examiner window. Press SELECT E to return to the editor.

Example: Clicking left on For Statement inserts a general template for a for statement.

```
FOR ⊂control variable⊃ := ⊂initial value⊃ ⊂to or downto⊃ ⊂final value⊃ DO
    ⊂statement⊃
```

The COMPLETE Command

Completing Reserved Words

When the cursor is at the end of your typein (as long as it is not inside a comment or string), pressing the COMPLETE key compares what you have typed with the set of all reserved words. If your typein completes to a unique string, it inserts the remaining characters and adjusts the font and case as appropriate.

Example: Typing proc and pressing COMPLETE inserts the remaining characters in "procedure" and places the word in a lowercase boldface font.

COMPLETE prints a message if more than one completion is possible. If no unique completion is available, press HELP to display all possible completions in the editor typeout window.

Example: Type pro and press HELP. The typeout window displays the reserved words program and procedure as the possible completions to your input. You can either press SPACE to make the typeout window disappear or click Left on one of the choices to have it inserted in your buffer.

Completing Language Constructs

In addition to completing reserved words, Pascal mode also completes language constructs. Once a template is inserted into a buffer, pressing the COMPLETE key at a template item further expands the item, overlaying its present contents with a more specific template. The COMPLETE command on a template works specific to the particular template; used after keywords, COMPLETE is specific to the keyword. The operation of c-HELP, on the other hand, is dependent on the syntactic context of the cursor.

When a unique completion exists, the editor inserts the new template directly into the buffer. (This behavior differs from c-HELP in that the latter always displays a menu rather than directly inserting a template.) When more than one completion exists, the editor displays a menu of completion choices.

Select a menu item by clicking the Left, Middle, or Right button on the mouse. Click Left, Middle, or Right on Exit to return to editing.

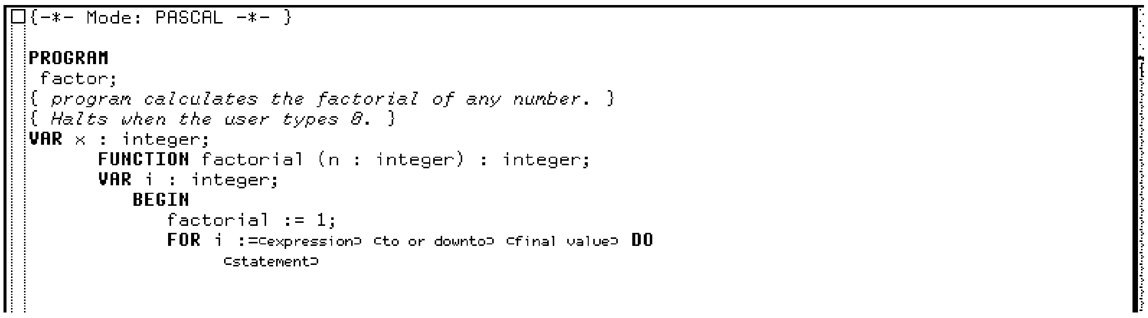
Mouse click	*Meaning*
Left	Inserts the selected template into the buffer at the cursor position.

Middle	Displays the selected template in a temporary window. Move off the window to make the template disappear.
Right	Not presently implemented.

Example: Assume that the following lines of source have been entered:

```
PROGRAM factor;
{ Program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
   FUNCTION factorial (n : integer) : integer;
   VAR i : integer;
   BEGIN
        factorial := 1;
        FOR ⊂control variable⊃ := ⊂initial value⊃ ⊂to or downto⊃
            ⊂statement⊃
```

You are ready to enter a `for` loop. Pressing `COMPLETE` at ⊂initial value⊃ replaces the template with ⊂expression⊃. See figure !.



```
{-*- Mode: PASCAL -*- }

PROGRAM
 factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
      FUNCTION factorial (n : integer) : integer;
      VAR i : integer;
         BEGIN
            factorial := 1;
            FOR i :=⊂expression⊃ ⊂to or downto⊃ ⊂final value⊃ DO
                 ⊂statement⊃
```

Figure 75. After pressing complete at ⊂initial value⊃.

Note that pressing `COMPLETE` at ⊂control variable⊃ does not further refine the template; no completion is possible.

Template and Completion Commands

Summary

`COMPLETE`	Completes a keyword to the left of the cursor or further fills in the current template.
`c-HELP`	Provides a menu of templates of valid language constructs inserted at the cursor position. Note: use `HELP` to get the standard Help capability.

`END`	Inserts a template that matches the keyword to the right of point.
`c-END`	Inserts whatever uniquely closes a language construct to the left of the cursor. For example, `c-END` inserts a close bracket ("]") to match a ("["), or and `then` to match an `if`.
`c-?`	Lists in an editor typeout window the possible completions of predeclared identifiers for the name immediately to the left of the cursor. This usage is specific to Pascal editor mode.
`SPACE`	Deletes the template item to the right of the cursor. The cursor must be positioned at the opening horseshoe.
`m-X` Remove Template Item	Deletes the next template to the right of the cursor. Same as `c-sh-T` and m `m-sh-T`.
`c-m-N`	Moves the cursor to the next template in the buffer.
`c-m-P`	Moves the cursor to the previous template in the buffer.
`c-sh-T`, `m-sh-T`	Deletes the next template to the right of the cursor. Same as Remove Template Item.

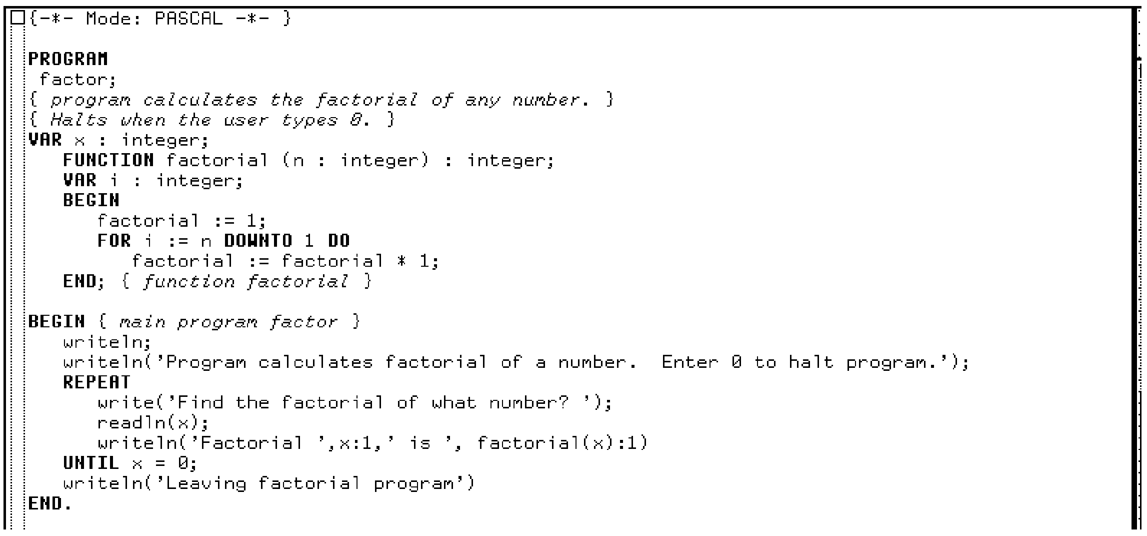
Electric Pascal Mode

Electric Pascal mode is an editor facility available in Pascal-mode buffers. As you type, input is placed in the appropriate font and case, depending on the syntactic context into which you insert the input. For example, by default, the word "procedure" typed within a comment is rendered in an italic face. On the other hand, "procedure" typed as a reserved word is placed in a lowercase and boldface face.

By default the Tool Kit renders Pascal reserved words in an uppercase boldface character style, body text in a lower-case Roman style, and comments in mixed-case italics. To change the defaults, use `m-X` Adjust Face and Case.

Example

Figure ! shows the effect of Electric Pascal mode on a complete program.



```
□{-*- Mode: PASCAL -*- }

PROGRAM
 factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
   FUNCTION factorial (n : integer) : integer;
   VAR i : integer;
   BEGIN
      factorial := 1;
      FOR i := n DOWNTO 1 DO
         factorial := factorial * 1;
   END; { function factorial }

BEGIN { main program factor }
   writeln;
   writeln('Program calculates factorial of a number.  Enter 0 to halt program.');
   REPEAT
      write('Find the factorial of what number? ');
      readln(x);
      writeln('Factorial ',x:1,' is ', factorial(x):1)
   UNTIL x = 0;
   writeln('Leaving factorial program')
END.
```

Figure 76. Example of a program typed in Electric Pascal mode.

Turning the Mode On and Off
Electric mode is available only in Pascal-mode buffers. If you created the buffer in such a way that the editing mode is not implicit (for example, via `c-U c-X B`), set the buffer mode to Pascal via `m-X` Pascal Mode. Then turn on electric mode using `m-X` Electric Pascal Mode. The mode line displays (`PASCAL Electric Mode`).

To turn off the mode if it is on, reissue the command.

Converting Code to Electric Pascal Mode Format
The Pascal editor mode provides a facility for applying the character style and capitalization rules of Electric Pascal mode to code that was not originally written using Electric Pascal mode, for example, code not developed under Symbolics Genera. This facility changes the face and case of reserved words to uppercase bold, and the face of comments to italics. It does not affect indenting.

This facility is invoked by `m-X` Format Language Region. You can apply the command to an editor region or to the current Pascal routine (default). Supplying a numeric argument reverses the effect: all formatting is removed from the specified editor region or routine.

Customizing Electric Pascal Mode

Introduction
When Electric Pascal mode is switched on, typed input is displayed in the appropriate face and case, depending on the *syntactic context* into which the input is inserted. Syntactic context refers to the following types of text:

- Body (plain) text

- Reserved words
- Comments
- Template items

Electric Pascal mode supplies a default case and face setting for each of these contexts. This section describes the default settings provided by Electric Pascal mode and explains how to change them, where possible.

Note: Face is only one element of character style. The other two elements are family and size. To change the screen defaults for family or size: see the section "Using Character Styles".

Default Case and Face
The table below shows the default case and face for each syntactic context.

Context	*Case*	*Face*
Body text	Lower	Roman
Reserved words	Upper	Bold
Comments	Leave alone	Italic
Template items	Leave alone	Roman

Leave alone means that input is displayed exactly as typed.

Changing the Global Defaults
The Tool Kit provides the special form **zwei:change-syntax-editor-defaults** to change Pascal editor mode's defaults for case, face, and indentation.

zwei:change-syntax-editor-defaults *Special Form*

No documentation available for special form ZWEI:CHANGE-SYNTAX-EDITOR-DEFAULTS.

Changing Indentation Globally
An easy procedure allows you to change the global indentation of Pascal source code. Select a Pascal mode buffer and position the cursor at the beginning of the construct whose indentation you wish to change. Press `SPACE` or `RUBOUT` for as many characters as you wish to indent or outdent the construct, respectively. Then press `c-I`. When the change is successful the editor displays a message, for example:

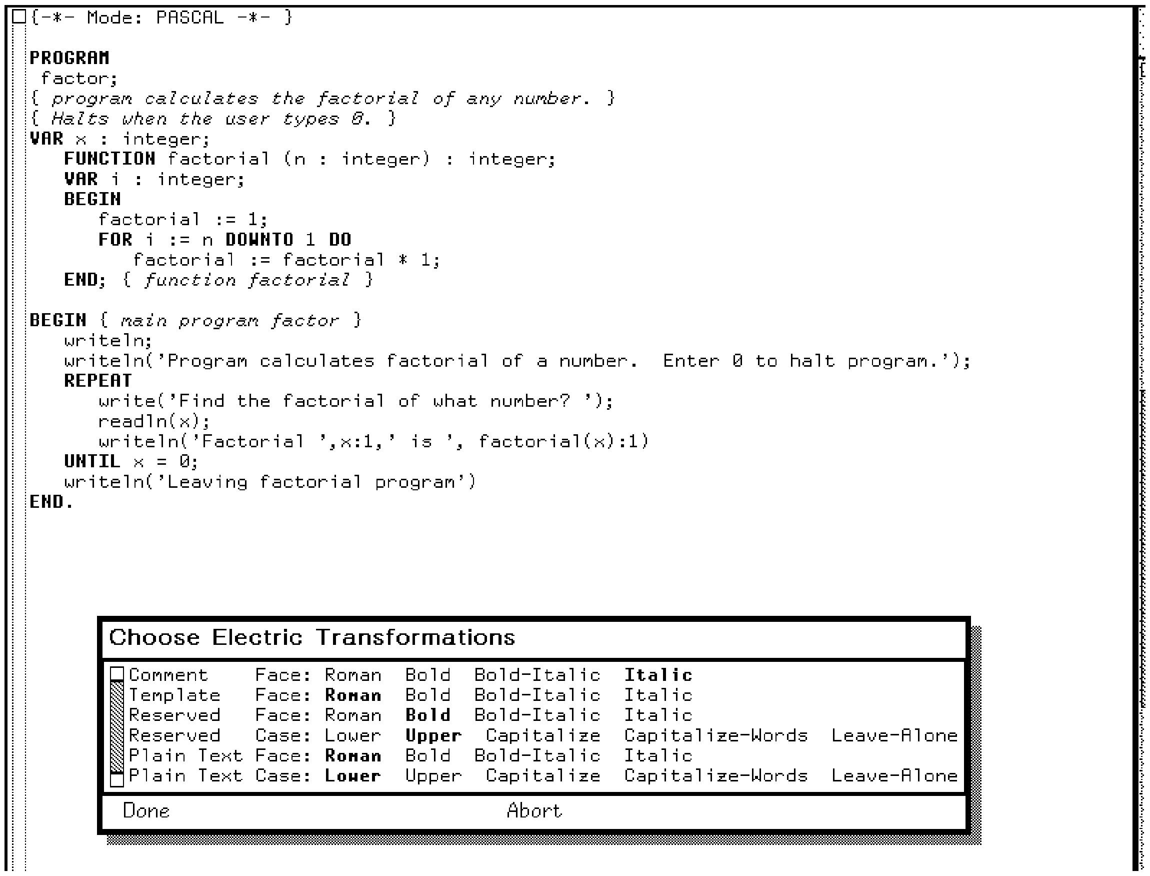
```
    Indentation for the construct changed to 2
```

However, when the change is not successful, for example, when the indentation for a construct cannot be changed, the editor will display a message to that effect. Once you are satisfied with the indentation, use `m-X` Save Indentation. The command produces a Lisp form in another buffer which reflects your changes; evaluate this form after the Pascal editor is loaded.

Changing Face and Case in the Current Buffer
Use `m-X` Adjust Face and Case to change the face and case of the syntactic contexts (templates, comments, reserved words, text) in the current Pascal mode

buffer. The command displays a menu, similar to the one shown in figure !. Boldfaced words indicate the current defaults.

Once you select new defaults for the buffer, click on Done. Invoke `m-X` Format Language Region to have the changes take effect in the current buffer.



```
{-*- Mode: PASCAL -*- }

PROGRAM
 factor;
{ program calculates the factorial of any number. }
{ Halts when the user types 0. }
VAR x : integer;
   FUNCTION factorial (n : integer) : integer;
   VAR i : integer;
   BEGIN
      factorial := 1;
      FOR i := n DOWNTO 1 DO
         factorial := factorial * 1;
   END; { function factorial }

BEGIN { main program factor }
   writeln;
   writeln('Program calculates factorial of a number.  Enter 0 to halt program.');
   REPEAT
      write('Find the factorial of what number? ');
      readln(x);
      writeln('Factorial ',x:1,' is ', factorial(x):1)
   UNTIL x = 0;
   writeln('Leaving factorial program')
END.
```

Choose Electric Transformations

Comment	Face:	Roman	Bold	Bold-Italic	**Italic**	
Template	Face:	**Roman**	Bold	Bold-Italic	Italic	
Reserved	Face:	Roman	**Bold**	Bold-Italic	Italic	
Reserved	Case:	Lower	**Upper**	Capitalize	Capitalize-Words	Leave-Alone
Plain Text	Face:	**Roman**	Bold	Bold-Italic	Italic	
Plain Text	Case:	**Lower**	Upper	Capitalize	Capitalize-Words	Leave-Alone

Figure 77. The menu displayed by the Adjust Face and Case command.

Note that you cannot alter the case of comments. The case for template items, reserved words, and plain text can be one of the following:

Case	*Meaning*	*Example*
Upper	all caps	PROCEDURE
Lower	all lowercase	procedure
Capitalize	initial cap	Initial_cap_on_first_word
Capitalize words	all initial caps	Initial_Cap_On_Each_Word
Leave alone	exactly as typed	EXACTLY_as_tYpEd

Pascal Implementation-dependent Values

Introduction

Contents
This section presents information specified by the Pascal Standards as implementation dependent. The topics covered are:

- Data representation
- Pascal file types
- Set constructors for unconstrained integers
- Standard functions

Note: This section does not present information on pathnames.

Pascal Data Representation

Introduction
The Pascal Tool Kit adheres to the proposed IEEE format for floating-point numbers (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).

The Tool Kit supports the ASCII character set, as extended for Symbolics Genera.

Table
The table below gives the range for each data type and the accuracy of the floating-point types.

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

Pascal File Types

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

Pascal 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".

Table
The following table shows the file types of Pascal source files on different systems.

System	*File type*
Genera	.pascal, .pas
UNIX	.p, .pas
All others	.pas

Set Constructors for Unconstrained Integers

Base Types

The base type of a set constructor is determined by the types of its member-designators; the base type is the type which can contain all member-designators. In the Tool Kit, set constructors of base type integer create sets with a default size of 256.

The following two examples illustrates how this fact affects the use of set constructors.

Example 1

```
PROGRAM Sets_example_1;
CONST largeNumber = 1000;
VAR s3  : SET OF 0 .. largeNumber;
     big : integer;

BEGIN
      big := largeNumber - 1;
      s3  := [1, 3, big]
END. { Sets_example_1 }
```

Example 2

```
PROGRAM Sets_example_2;
CONST largeNumber = 1000;
TYPE largeNumberType = 0 .. largeNumber;
VAR s3  : SET OF 0 .. largeNumber;
     big : largeNumberType;

BEGIN
      big := largeNumber - 1;
      s3  := [1, 3, big]
END. { Sets_example_2 }
```

Explanation

Both examples appear the same but have one major difference — in the first example `big` is of type `integer`; in the second, of type `largeNumberType`, a subrange of type `integer`. In both examples the set constructor is `[1,3,big]`, but in `Set_example_1`, the set constructed can contain 256 members (the default size of sets of type `integer`), whereas in `Set_example_2`, the set constructed can contain 1000 members of type `largeNumberType`.

In effect the compiler determines the base type and hence the size of the set from the set constructor. The rule is that the size of a set is determined by the upper bound of its largest member. For example, consider a hypothetical set that consists of a constant of 5, an unconstrained integer variable, and a variable whose type is declared to be some subrange of integer, for example 1 to 1000. The first member, the constant, contributes an upper bound of 5; the second member, the unconstrained integer, contributes an upper bound of 256; the third member contributes an upper bound of 1000. The maximum of all these is 1000; hence 1000 is the maximum element in the set.

The set constructor in Example 1, `[1, 3, big]`, contains the unconstrained integer variable `big` and some small constants. Therefore the upper bound of the set is 256.

The set constructor in Example 2, `[1, 3, big]`, contains the constrained integer variable `big` of range 0 to 1000 and some small constants. Therefore the upper bound of the set is 1000.

`Set_example_1` causes a run-time error because the variable `big` is larger than 256. `Set_example_2` runs properly.

Standard Pascal Functions

Predicates

The following are the characters to type for formatted input from an interactive stream:

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

Compiling Pascal Programs

Features of the Pascal Compiler

Lisp Code Is Produced

The Pascal compiler produces Lisp code, which is compiled into Genera "machine code" by the standard Lisp compiler. Since Lisp is essentially a machine language, 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 source-level Debugger, the Display Debugger, works on the Pascal source code.

Benefits of Lisp Code
You derive several important advantages from compiling into Lisp. One is that Lisp numeric functions are called from Pascal; indeed, many of the Pascal library subroutines, such as `sin` and `cos`, are those built into the Lisp system. Another benefit is that you can choose to implement a particular routine in either Lisp or Pascal, whichever language is more suitable.

Compilation Is Incremental
The Symbolics Genera 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 compiled. This permits you to compile any routine whose containing routine is compiled, without recompiling the containing routine.

For example, if you have compiled a Pascal program with internal procedures or functions, positioning the editor cursor inside an internal routine and typing `c-sh-C` compiles that routine. Such a feature encourages frequent and thorough debugging and greatly speeds up the program-development process. This feature also extends to unit specifications and unit implementations: See the section "Units Facility".

Incremental compilation of a particular routine might fail if the pattern of enclosing references to variables, constants, or labels defined in containing routines has changed sufficiently. If this happens, the compiler issues an error message, and you must recompile the containing routine. If the containing routine is a Pascal program rather than a function or procedure, this error cannot occur, and incremental compilation of any routine whose immediate superior is a program will always succeed. The one exception to this is when an enclosing reference to a previously unreferenced label is introduced. In that event you may have to recompile the program as well. You will see a suitable error message in such cases.

Compiler Options
The Pascal Tool Kit supports two compiler options. One initializes program data; the other allows you to choose a default dialect of Pascal.

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.

See the section "Compiler Option: Initializing Pascal Program Data". See the section "Compiler Option: Setting the Default Pascal Dialect".

Compiler Warnings
The Pascal compiler produces diagnostics whenever a program violates the rules for a legal program, as specified by one of the Standards. 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` Compiler Warnings places the database in a buffer called *Compiler-Warnings-*n** and selects that buffer. `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.

Zmacs Compiler Commands
Several Zmacs commands are available for compiling Pascal 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 Pascal Program
: Checks to see that the cursor is positioned near a valid Pascal program and then compiles and executes the program, without run-time options, with predefined file 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. This feature is useful for resuming compilation when a previous attempt has failed.

`m-X` Compile File
: Compiles a file, offering to save it first if the buffer has been 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*, a contiguous delimited section of text in the editor buffer. 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-1*`, 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.

Command Processor
The Compile File Command Processor command compiles files of Pascal routines.

Compiling Small and Large Pascal Programs

Small Programs

One method 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 Pascal routine, this creates a function in the machine's virtual memory but does *not* create a file version of the object code. Note that you can have a region 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 a program (or programs) contained in one or more files is to compile the files 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 Pascal routines is defined at compile time. Instead the compiler puts instructions into the compiled-code file that causes definition to occur at load time. Load the compiled-code file into virtual memory with `m-X` Load File.

The other method of compilation is appropriate for a large Pascal program, especially one containing several files.

1. Declare a group of Pascal files (or even Pascal and Lisp files) as system. A *system* is a set of files and a set of rules that defines the relations among these files; together these files and rules constitute a complete program.
2. Use the Compile System command to compile all the program's routines and load the object code into virtual memory.

This method results in the creation of Lisp compiled-code files in the file system.

For a description of the facilities for declaring a Pascal system: See the section "Large Pascal Programs".

Compiler Option: Initializing Pascal Program Data

Problem
Normally, the Tool Kit sets variables not explicitly initialized by Pascal data statements to the Lisp string "Undefined". 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.

How to Set the Option
To avoid this potential problem, set the initialization variable that controls this behavior — **pascal::*pascal-init-to-zero***.

pascal:*pascal-init-to-zero* *Variable*

Controls whether local Pascal 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 Pascal data statements to the Lisp string "Undefined". **nil** is the default.

Example: To set the form in your init file, type:
```
(login-forms
  (setq pascal:*pascal-init-to-zero* t))
```
However, unless (1) the Pascal system is stored on the disk and therefore accessible when you log in or (2) your init file loads Pascal, 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 Pascal system to memory but before compiling the program. Type:
```
(setq pascal:*pascal-init-to-zero* t)
```
To turn off this option, reset the variable. Type to a Lisp Listener:
```
(setq pascal:*pascal-init-to-zero* nil)
```

Note on Global Variables and Arrays
The **pascal::*pascal-init-to-zero*** option controls initialization of *local* variables by the compiler. To control initialization of arrays and other "large" variables, use the **:init-to-zero** option to **pascal::execute** at run-time.

Compiler Option: Setting the Default Pascal Dialect

Two Dialects
The Tool Kit compiles two dialects of Pascal, Pascal/VS and ISO Pascal. Normally, the default Pascal dialect for an editor buffer or file, if none is specified in the attribute list, is ISO Pascal. A buffer or file is always associated with one of these dialects of Pascal.

How to Set the Option

The **pascal::*pascal-default-dialect*** compiler option controls the default dialect.

pascal:*pascal-default-dialect* *Variable*

Changes the current default Pascal dialect from ISO Pascal to Pascal/VS, or vice versa. The valid values of **pascal::*pascal-default-dialect*** are **:iso** and **:vs**, two Lisp keyword symbols representing the dialect names. The default, if none is specified in the attribute list, is ISO Pascal.

To set the default to Pascal/VS, add the following form to your init file:

```
(login-forms
  (setq pascal:*pascal-default-dialect* :vs))
```

However, unless (1) the Pascal system is stored on the disk and therefore accessible when you log in or (2) your init file loads Pascal, 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 Pascal system to memory but before compiling the program. Type:

```
(setq pascal:*pascal-default-dialect* :vs)
```

To change the value of the default dialect back to ISO Pascal, type:

```
(setq pascal:*pascal-default-dialect* :iso)
```

Building Pascal Applications with Run-time Systems

Overview of Building Pascal Applications

Pascal incorporates functionality providing you with the option of incorporating the Pascal run-time system as a part of the Pascal application you build. Users of such an application can run it regardless of whether they are running the Pascal development system.

Customers who distribute an application with the Pascal run-time system **must** sign a *Sublicense Addendum to the Terms and Conditions of Sale, License and Service*. See the section "Sublicense Addendum for Symbolics Pascal".

Introduction to Creating Pascal Run-time Systems

Components of a Run-time System

A run-time system (as opposed to the development system) for a language is made up of the minimum subset of the 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 initiates 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 Pascal run-time system is called *Pascal-Runtime*.

Creating Applications

Normally, you develop an application in a development environment, and deliver it with the Pascal run-time system. You can minimize the size of your Pascal system by filtering out information needed for debugging support or support for the Pascal editor mode at compile time.

There are two steps for creating an application that includes a run-time system:

1. During program compilation, you set a global variable to filter out debugging information from binary files. This step helps reduce the size of the finished application.

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

Compiling Pascal Files to Execute in Run-time Systems

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*. It is possible to exclude this information from the bin file by setting the special variable **compiler-tools-system::*compile-for-run-time-only*** to the Lisp boolean "**t**". Doing so minimizes the size of the binary files produced for an application.

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. Use of this facility does not change the generated code. The section "Program Configurations: Development System and Run-time System Options for Pascal Systems" specifies the capabilities of *rto* binary files.

Incorporating the Run-time System as Part of a Pascal Application

You can package the run-time system should as a dependent component system of the application. There are two requirements you must meet when defining such a packaged system:

- The packaged system definition must load the run-time system before loading the application program. Specify the appropriate `:serial` dependency in order to load the run time system.

- The definition must read the system declaration file for the run-time system to be read before any Pascal file is encountered in an application system.

The following example illustrates how an application named *a1* is packaged. Note that *a1* is a component system (with accompanying separate *sysdcl* file) and is not a separate subsystem when the sysdcl contains any references to objects defined in the system and/or user package defined by the run-time system in question.

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

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

The "write distribution tape" and "read distribution tape" software is used to distribute the packaged software.

Program Configurations: Development System and Run-time System Options for Pascal Systems

Given the ability to produce rto bin files and to make the Pascal run-time system part of a Pascal application, you can incorporate a program in a configuration 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.

Program Configurations: Development System and Run-time System Options

	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 being incrementally compiled are compiled.

Purpose of Configurations

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

2. *Normal bin, Run-time system* This configuration is advantageous when software is actively developed, but also simultaneously used in a run-time system.

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

4. *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.

Large Pascal Programs

Introduction to Large Pascal Programs

Introduction

Many programmers like to 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 are edited but not recompiled.

To overcome the drawbacks inherent in manipulating small chunks of a large program, the Tool Kit provides a way to define a collection of Pascal routines, possibly spanning several files, as a Pascal *system*. The separate compilation units facility provides a way to allow the routine definitions in the various files of the Pascal system to be compiled in the appropriate declaration scope.

Contents
This chapter introduces the Pascal Tool Kit facilities that aid in the definition and maintenance of Pascal programs that span several files. Two topics are discussed:

1. The *units facility*, which permits compilation of routines in separate files.

2. The *system definition facility*, which allows you to specify the order of compilation and loading of a group of Pascal source files.

Read the following documents describing analogous facilities for large Lisp programs before using the system definition facility.

- See the section "Packages".

- See the section "System Construction Tool".

Units Facility

Rationale

The ISO Pascal specification requires that all internal routines of a Pascal program be in the same compilation unit as the program, which typically means in the same file as the main program. Software considerations, however, dictate that large programs are separated into manageable units rather than placed in one large file. The purpose of the Tool Kit units facility is to overcome this shortcoming of ISO Pascal.

Note: The Pascal Tool Kit has not implemented the Pascal/VS separate compilation facility.

General Description

The units facility provides a simple means for programs to call stand-alone units of code residing in the same or other files. The term *stand-alone* means that each unit is compiled separately but executable only within the scope of a main program. Typically, you can enable many programs to call primitive functions and procedures defined in another file and available to many other users.

Any number of main programs can call a particular unit. A unit in turn can also call other units. A program or unit that calls a unit is said to *use* the unit and is known as the *using* routine. The called unit is referred to as *used* and is known as the *used* routine.

Note that the using program or unit and the used unit are at the same lexical level; no hierarchical relationship exists between them. The scope of the using routine *expands* to include the declarations in the used units.

A unit consists of two major parts — the unit specification and the unit implementation. The facility introduces three new reserved words: `unit_specification`, `unit_implementation`, and `using_units`.

Unit Specification

The unit specification is the public interface listing all the routines available to other units and programs as well as to other routines in the same unit. The unit specification does not itself contain executable code but rather specifies the procedures and functions usable by other routines. Each specification must have a corresponding unit implementation, which actually contains code. The implementation can be in the same file as the specification.

All types and variables declared in the used unit specification are global to the using program or unit. The using routine and the used unit are at the same lexical level.

You have to compile the unit specification by itself before compiling the using routine and corresponding unit implementation.

The unit specification begins with the reserved word `unit_specification` followed by a unit identifier and a semicolon.

If the unit specification uses one or more unit specifications, the second statement in the specification must begin with the reserved word `using_units` followed by a valid unit specification identifier. Use a comma to separate multiple identifiers. The statement ends with a semicolon.

Specify the constant, type, or variable declarations, as usual.

Specify each routine header you wish to declare available for public use. Begin with the reserved word `procedure` or `function`, as appropriate. Then specify the procedure or function identifier. Include the parameter list in parentheses in the usual fashion. If the routine is a function, type a colon and the return value type. End the statement with a semicolon. The routine heading for a function looks like this:

FUNCTION *function-identifier* (*parameter-list*) : *return-value-type*;

The routine heading for a procedure looks like this:

PROCEDURE *procedure-identifier* (*parameter-list*);

Close the unit specification with the reserved word `end` and a period. Note the absence of a corresponding `begin`.

The format of the entire unit specification is as follows:

```
unit_specification unit-identifier;
 using_units unit-identifer-1, ... ; {specify only if ...}
 constant-declarations;              {it uses another unit}
 type-declarations;
 variable-declarations;
 routine-header-1;
        .
        .
 routine-header-n
END.
```

Unit Implementation

The unit implementation is the non-user-visible part of the unit defining every routine listed in its corresponding unit specification. While it must contain the code for at least as many routines listed in its corresponding unit specification, it might in fact contain more, for example, in the case where a publicly available routine calls a routine not available for general use.

You can compile the entire unit implementation or any contained routine by itself. You have to compile the implementation *after* its corresponding unit specification but *before* the using routine is executed.

The type and variable declarations in the unit implementation are available *only* to that implementation. These declarations are local to the unit implementation.

Note this important restriction on variable identifiers: all unit implementations must have unique variable names. For example, `unit1` can declare variable `i` as an integer, but `unit2` cannot legally redefine `i` as a boolean value.

The unit implementation begins with the reserved word `unit_implementation` followed by a unit identifier and a semicolon. Note that the unit identifier must be the same as that used in the corresponding unit specification.

Specify the constant, type, or variable declarations, as usual.

Define all procedures and functions listed in the unit specification as well as any private routines that they call. Note that the public routines must include the reserved word **procedure** or **function**, the routine identifier, and a semicolon, but *not* the parameter list and (in the case of a function) a return value type. This information is provided in the corresponding unit specification.

Close the unit implementation with the reserved word `end` and a period. Note the absence of a corresponding `begin`.

The format of the entire unit implementation looks like this:

```
unit_implementation unit-identifier;
  constant-declarations;
  type-declarations;
  variable-declarations;
  function-or-procedure-body-1; {Don't include the parameter list}
          .
          .
  function-or-procedure-body-n
END.
```

Calling a Unit

After the unit specification and unit implementation is written, using a unit routine in a main program or unit is quite simple. Place a `using_units` statement in the calling program or unit. Make sure you compiled the unit specification and implementation of the used routine. Then call the routine within the body of the main program or unit specification.

Remember that the using program or unit and the used unit are at the same lexical level; no hierarchical relationship exists between them.

If included, make the `using_units` statement the second statement in the main program or unit specification. Begin the statement with the reserved word `using_units` followed by a valid unit specification identifier. Use a comma to separate multiple identifiers. The statement ends with a semicolon.

The format of a `using_units` statement within a main program looks like this:

```
PROGRAM program-identifier;
 using_units unit-identifer-1, ... ;
 constant-declarations;
 type-declarations;
 variable-declarations;
BEGIN
          .
          .
 call-a-routine-in-specified-unit;
 call-another-routine-in-specified-unit;
          .
END.
```

Compilation

The incremental compilation capability of Symbolics Genera extends to the units facility. The compiler maintains a description of the declaration scope of every routine compiled. You can compile any routine whose containing routine is compiled, usually without recompiling the containing routine.

Both the unit specification and unit implementation are separate compilation units. In addition, you can compile any routine in the implementation incrementally.

The order of compilation is important, as expressed in the following rules.

1. Compile the unit specification before compiling the using routine and corresponding unit implementation.
2. Compile the using routine after you compile the unit specification; however, you do not have to compile the corresponding unit implementation.
3. Compile the unit implementation *after* its corresponding unit specification is compiled and *before* the using routine is executed.

Example
In the commented example below the program `main` uses the factorial function declared in unit specification `unit1`. The body of the function is defined in unit implementation `unit1`.

```
PROGRAM main;
using_units unit1;
  { Usual declarations here. }
  VAR value : int;
BEGIN
  write('Find the factorial of what number?  ');
  readln(value);
  writeln('Factorial ',value,' is ',factorial(value))
  { Call one of the routines specified in unit1, unit2, }
  { or unit3.  Call is valid only within main program. }
END.

unit_specification unit1;
using_units unit2, unit3;
      {Declarations in unit2 & unit3 are global to unit1 and}
      {hence to program main.}
  TYPE int = integer;
  {Declarations here are global only to unit1 and its using routines.}
  FUNCTION factorial (n : int) : int;  { Parameter list here. }
          .
  {Additional routine headers}
END.       { Note the absence of begin. }
```

```
unit_implementation unit1;
     {Declarations here are available only to this unit implementation.}
  FUNCTION factorial;   { No parameter list here. }
  VAR i : integer;  { Local declaration available only to factorial.}
  BEGIN
     factorial := 1;
     FOR i := n DOWNTO 1 DO
       factorial := factorial * i
  END; { Function factorial }
           .
  { Additional routine bodies }
  { Note: routines defined here can use other routines in }
  { this unit, as well as routines in unit2 and unit2. }
END.       { Note the absence of begin. }
```

Zmacs commands
The following Zmacs commands support the units facility. They are available from Pascal mode.

- m-X Show Dependent Units prompts for the name of a Pascal unit specification and displays a list of those units dependent on that specification.
- m-X Show Units Depended On prompts for the name of a Pascal program or unit and displays a list of those units depended on by that program or unit.

What Is a Pascal System?

Definition
A *system* is a set of files and rules defining the relations among these files; together these files and rules constitute a complete program that you can manipulate as a unit. Use the Lisp function **pascal::def-program** to designate a group of Pascal files (or Pascal and Lisp files) as a Pascal system. The declaration allows you to specify the files composing the system as well as the desired properties of the system, such as the package into which the object code is compiled.

pascal::def-program is analogous to the **defsystem** function used to declare Lisp systems. In fact, the facilities provided for Pascal are simply Lisp macros that expand into **defsystem** invocations.

Benefits
The Pascal **def-program** facility offers several benefits:

- Compiled code is stored on disk until loaded.
- You can compile and load all the system files to your environment *at one time* and in accordance with the properties you specified in the system definition. Moreover, you can compile only those files that need compiling, that is, only those source files that are newer than their corresponding object files.

- It supports the Tool Kit's units facility by providing syntax to express compilation-order dependencies.

Procedure
The procedure below summarizes all the steps necessary for declaring, compiling, and loading a Pascal system.

1. Make a package declaration for the files composing the system, using **pascal::package-declare-with-no-superiors**. Alternatively, you can use the default package **pascal-user**. See the section "Declaring a Pascal Package". Also add the name of the package to the attribute list of the individual files in the system.
2. Make a system declaration for all files, using **pascal::def-system**. 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 Pascal System".
3. 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".
4. Compile and load the Pascal system defined in step 2, using the command processor. See the section "Compiling and Loading a Pascal System".

Write the package and system declarations in the same file, called the *system declaration file*. This file must have a Lisp file type and should be in the **user** package.

Declaring a Pascal Package

Introduction
After you load Pascal and Lisp routines into the environment, they remain there until replaced by recompilation or until the machine is cold booted. Since you can have two large Pascal programs that have the same name, you have to specify a method that Symbolics Genera uses as 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*.

Packages avoid naming conflicts; two different Pascal programs can have the same name only if each exists in its own package. For example, you have to place two Pascal programs assigned the name `primes` by the compiler in two different packages.

You can avail yourself of the default package provided by the Tool Kit, or you can create your own using the **pascal::package-declare-with-no-superiors** special form. The facility for declaring Pascal systems enables you to specify only an *existing* package in the definition; that is, you have to previously define and compile the package. All files in the system are compiled in the package you specify in the system declaration.

Definition
The Tool Kit provides the Lisp special form **pascal::package-declare-with-no-superiors** for creating packages appropriate for Pascal code.

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

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

```
(pascal: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, 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 Pascal 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 have to 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. You can enable the editor to set the mode automatically by creating the buffer 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 the same file in which you enter your Pascal system declaration (called the *system declaration file*). You have to assign the file name a Lisp file type and compile it to the **user** package.

  Compile the package declaration.

Predeclared Pascal Packages

The Pascal Tool Kit recognizes two packages not needing explicit definition.

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

  CAUTION: **cl-user** inherits all the symbols in the **global** package containing the basic symbols of the Lisp language. As a result, the names of Pascal routines or variables in the **cl-user** package can conflict with those of existing Lisp functions.

- **pascal-user** is the default package of the Tool Kit. Since **pascal-user** was originally declared using **pascal::package-declare-with-no-superiors**, no name conflicts can occur for routines or variables in this package.

How to Assign a Pascal Package

You should always 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 **pascal::package-declare-with-no-superiors**. Invoke `m-X` Reparse Attribute List to have the change take effect.

If the file is to be part of a system declaration, it is recommended that you specify the package as value of the **:default-package** option in the system declaration. The files you list in the definition become associated with the specified package. You should note that all Pascal files of a system declaration must be compiled into the same package or they will not correctly reference each other.

Declaring a Pascal System

Before declaring your own Pascal system, see the section "System Construction Tool".

Definition

pascal:def-program *name options* &rest *body* *Function*

Declares a set of files as a Pascal system, where *name* is the name you have chosen for your Pascal system, not necessarily the name of a main program. *options* are the valid options to **defsystem**, the analogous function for Lisp. See the section "Defining a System". Note that Pascal systems can consist of files of Pascal code as well as files of Lisp code or just files of Pascal code.

body is a file specification or a list of file specifications, or **:using**. Each specification is a string representing the name of a file in the system. No *dependencies* can exist between the files in the list; that is, one file cannot depend on another file compiled or loaded before it. Note: Unless you specify the **:using** option, no order of compilation or loading is implied or guaranteed.

The **:using** keyword is the mechanism for establishing dependencies between files. Use this option when the order of compilation and loading is important to the proper operation of the system. Using, as a concept, refers to the condition that a routine in one file can use (call) separately compilable unit specifications residing in another file. See the section "Units Facility".

The first argument to the **:using** keyword is a string or list of strings, specifying a file or set of files containing a unit specification used by other routines or units. This file contains the unit specification called last.

You can make each argument following the first argument a file specification string, a list of file specification strings, or another **:using** form (nested **:using** options!). Each of the file specification strings can refer to a file which uses one of the files specified by the first argument to **:using**.

Example: If a main program `one` in file "apple" calls unit specification `spec_one` in file "banana", which in turn uses unit specification `spec_two` in file "cantaloupe", then the **:using** option is written as follows:

```
(:using "cantaloupe" {Spec in this file called last, ...}
 (:using "banana"                     {compiled first}
  "apple"))         {Program in this file called first, ...}
                                      {compiled last}
```

The significance of the **:using** keyword is that you can compile and load files specified by the first argument before performing compilation and loading operations on the remaining files.

In addition to **:using**, you can specify several of the keyword options to and transformations of **defsystem**, the analogous function for defining Lisp systems.

Example 1

fred-program consists of one main program residing in one Pascal file — "fred".

```
(pascal:def-program fred-program
                    (:default-pathname "f:>fred>")
                    "fred")
```

Example 2
You can use the following code for creating a system called **freds-family** consisting of two main programs residing in "fred" and "freds-sister" respectively. No dependencies exist between the programs.

```
(pascal:def-program freds-family
                    (:default-pathname "f:>fred>family>")
                    ("fred" "freds-sister"))
```

or

```
(pascal:def-program freds-family
                    (:default-pathname "f:>fred>family>")
                    "fred" "freds-sister")
```

Example 3
You have a Pascal program `plot` with three files "onedplot", "twodplot", and "axislabels" residing in directory "f:>fred>plot>". "Oneplot" contains the definition of a Pascal main program which calls two unit-specifications — "twodplot" and "axislabels". In addition, you are compiling the object code into a previously defined and compiled package called `plot`.

```
(pascal:def-program plot
                    (:default-pathname "f:>fred>plot>"
                     :default-package "plot")
                    (:using "twodplot" "axislabels"
                     "onedplot"))
```

Note that by supplying a pathname default you avoid having to type the full pathname of every file in the system; instead, just the file name portion of the pathname suffices.

Example 4
The **units-program** includes a nested **:using** form. The main program "fred" calls a specification-unit "sally", which in turn calls specification-unit "david". Compile and load "david" first. The main program that calls both units is compiled/loaded last.

```
(pascal:def-program units-program
                    (:default-pathname "f:>peter>units>"
                    (:default package "units")
                    (:using "david"
                     (:using "sally"
                      "fred")))
```

Example 5
System declarations can mix Pascal and Lisp code. Note carefully that the Lisp code can in no way depend on the Pascal code, and vice versa. In the example below, the value of the **:default-package** keyword is a Pascal package. Assume

that the Lisp code is compiled into the **cl-user** package, specified in the attribute list of each Lisp file.

```
(pascal:def-program lisp-and-pascal
                    (:full-name "Registration System"
                     :default-pathname "f:>sr>registrar>"
                     :default-package "registrar")
   (:serial (:parallel "definitions" "macros")
            "display")
   (:using "david"
    (:using "sally"
     "fred")))
```

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. These options apply only to Lisp code within the **pascal::def-program** form. (see the section "Short-form Module Specifications".)

Combining Pascal, FORTRAN, and Lisp
Use **defsystem** to create a system combining Pascal, FORTRAN, and Lisp code. Note in the example below that the files of each language are specified separately in their own **:module** declaration.

Use **:module** (considered the long-form module specifier) 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 compile or load dependencies, each module compiles and loads in turn. FORTRAN compile and loads first, then Pascal, then Lisp.

Example:

```
(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 Pascal System

Compile System and Load System compile and load your Pascal system, respectively. These commands load the system declaration file if (1) you create 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 to load 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
```

Executing Pascal Programs

No Link-and-Load Step

Virtual Memory

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

No Executable Module

The *executable module*, as most Pascal programmers understand the term, does not exist. Rather, once routines are brought into virtual memory by compilation in an editor buffer or a Pascal system is loaded via the Load System command, they are immediately executable. Thus, the Tool Kit requires no separate link-and-load step.

Compilation is incremental; consequently, the absence of the link step is of great significance when making small changes to large Pascal programs, since the link step in traditional computing environments is time-consuming.

How to Run a Pascal Main Program

Introduction

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

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

- Lisp: **pascal::execute**
- Zmacs: `m-X` Execute Pascal Program or Compile and `m-X` Execute Pascal Program
- Command processor: Execute Pascal

Zmacs Commands

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

Compile and Execute Pascal Program performs identically to Execute Pascal Program, except that it first compiles the program to virtual memory before executing it.

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

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

Example: You can invoke `(pascal:execute pascal-user:mean :init-to-zero t)` from the command processor as follows:

```
Execute Pascal pascal-user:mean :Init_to_zero yes
```

Lisp Function

pascal:execute *main-program-name* &key *input output (trap-underflow t) init-to-zero pathname-defaults (stack-size *pascal-stack-increment-size*) save-environment reload*
Function

Runs a Pascal program, where *name* is a symbol representing the name of the main program. **pascal::xqt** is a synonym for **pascal::execute**.

The Pascal run-time system supports numerous keywords: **:init-to-zero**, **:input**, **:output**, **:pathname-defaults**, **:reload** **:save-environment**, **:stack-size**, **:stream**, and **:trap-underflow**.

See the section "Pascal Main Program Options".

Example 1
To run a main program `convert`, invoke a Lisp Listener and type:

```
(pascal:execute 'pascal-user:convert)
```

Example 2
To run a main program `convert`, which reads a single line of parameters from the standard text file `input`, invoke a Lisp Listener and type:

```
(pascal:execute 'pascal-user:convert)1 2 3 RETURN
```

Lisp Listener Package
Run a Pascal 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 Pascal program.

For example, assuming that the Lisp Listener package is **cl-user** and the Pascal program package is **pascal-user**, then the following invocation of `convert` signals an error:

```
(pascal:execute 'convert)
```

The **user** package does not recognize `convert` as a Pascal program.

Determine the package of the Lisp Listener as follows:

At a top-level Lisp Listener the package is the default **cl-user**, unless you have explicity changed it. In the editor, the Lisp Listener package corresponds to that of the Zmacs buffer. If you display two or three windows at a time and associate the contents of each with a different package, the package of the Lisp Listener is the same as that of the window in which the breakpoint was invoked. Example: If the cursor was sitting in the middle window of a three-window screen when you invoked 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:

```
(pascal:execute 'pascal-user:quadratic)
```

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

Pascal Main Program Options

Introduction

All Pascal main programs accept several pairs of keywords and values. Still, these keywords are recognized:

- **:init-to-zero**
- **:input**
- **:output**
- **:pathname-defaults**
- **:reload**
- **:save-environment**
- **:stack-size**
- **:streams**
- **:trap-underflow**
- **:use-abort-mode**

Format

The options are specified as keyword-value pairs:

`(pascal:execute` *program-name keyword value* `...)`

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

```
(pascal:execute 'pascal-user:mean :input "f:>tr>abc.data"
                                  :save-environment t)
```

:init-to-zero

Normally, the Tool Kit sets Pascal variables to the Lisp string "Undefined". 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.

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

Example: To run the program `mean`, setting all variables to zero, type:

```
(pascal:execute 'pascal-user:mean :init-to-zero t)
```

Note that this option controls initialization of arrays, common variables, and other global variables. To control initialization of local variables by the compiler, set the **pascal::*pascal-init-to-zero*** compiler option.

:input and :output

In the call to the main program, you can specify the **:input** and/or **:output** keywords with an argument of *file-source*. *file-source* refers to the files associated with Pascal standard input and output files and are either:

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

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-defaults** main program option, if present. Then the file is opened and passed into Pascal. On exit from the program, the file is closed.

If you specify a stream, it will not be closed on exit from the Pascal program.

Example: When you run the main program `mean`, standard input is associated with the file "f:>tr>abc.pascal" and standard output with Lisp stream *error-output*.

```
(pascal:execute 'pascal-user:mean :input "f:>tr>abc.pascal"
                                  :output *error-output*)
```

:pathname-defaults

The **:pathname-defaults** option specifies the pathname used as the default by the Pascal I/O system when parsing file names specified in the **:input** or **:output** option to **pascal::execute** or in such file-opening predefined procedures as `reset` or `rewrite`. See the section "Extensions to Pascal".

The pathname specified as the default is merged with the prevailing default. If the **:pathname-defaults** option is not specified, then the prevailing default is used.

Example: Assume that the prevailing pathname default is "s:>tc>kr.lisp".

You run `payroll` supplying the **:pathname-defaults** option:

```
(pascal:execute 'pascal-user:payroll :pathname-defaults ".pascal")
```

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

Assume also that the statement in your program calling `reset` reads:

reset`(f, 'name = >tc>payroll-dir>')`

When the program is executed, the pathname of the reset file is constructed from the new prevailing default and the value of the name parameter:

```
"s:>tc>payroll-dir>sr.pascal"
```

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

:reload

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. **nil** is the default.

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

```
(pascal:execute 'pascal-user:mean :save-environment t
                                  :reload t)
```

:save-environment

For very large programs, allocating the global data is time consuming. For this reason, the Pascal run-time system supports the **:save-environment** keyword. *Note*: An environment remains saved until you explicitly discard it using the **:reload** option.

Value	*Meaning*
t	The old environment, if any, is incrementally modified to reflect the current compilation state before running the program.
nil	The old saved environment is ignored when running the program. **nil** is the default.

Example: Suppose you want the global variables allocated only once instead of every time you run `mean`. Type:

```
(pascal:execute 'pascal-user:mean :save-environment t)
```

:stack-size

The **:stack-size** option specifies the initial size of the memory stack used when beginning execution of a Pascal program. Variables declared immediately inside a Pascal program are allocated their own arrays and do not take up space in the stack. Thus, for many programs, the stack space used is modest. If, however, substantial amounts of data are declared in procedures and functions inside the program, or if there is some memory data allocated in each frame of a deeply recursive program, this option is useful.

If you do not specify a **:stack-size**, the initial stack size is 8192. The Pascal Tool Kit allocates more stack if the default stack overflows, whether or not you supply this option.

Example: To run program **towers** with an initial stack size of 50,000, type:

```
(pascal:execute 'pascal-user:towers :stack-size 50000.)
```

Note the period after 50000, which denotes a decimal number.

:streams

The **:streams** keyword option allows Pascal I/O routines to operate on streams created in the Genera environment. The value of **:streams** is a list of dotted lists, specifying the input and output streams. The format is

:streams `((*Pascal-stream-name* . ,*Lisp-stream*) ...)

Example: *instream* and *outstream* are generated in Lisp and used during the execution of the Pascal program `run_tool`. The following call takes input from an editor buffer specified by the parameter *definition-name* and creates an editor buffer named code.data, sending the output to the buffer rather than to the terminal.

```
(defun foo (definition-name)
  (with-editor-stream
    (outstream :buffer-name "code.data"
               :no-redisplay t
               :kill t :start :beginning
               :create t)
    (with-editor-stream
      (instream :buffer-name definition-name
                :kill nil :start :beginning)
      (pascal:execute 'pascal-user:run_tool
                      :streams `((outstream . ,outstream)
                                 (instream . ,instream))))))
```

:trap-underflow

If an underflow is too small for expression as a double-precision floating point number — less than ˜1.175 x 10^{-38} and less than ˜2.2 x 10^{-308}respectively — your machine may signal an error.

You can turn off trapping mode using 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 to 0 or a denormalized number, if it is too small for expression as a normalized floating-point number.
t	Turns off nontrapping mode. Underflow is detected if a result is too small for expression as a normalized floating-point number. **t** is the system default.

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

```
(pascal:execute 'pascal-user:add :trap-underflow nil)
```

:use-abort-mode

Enables you to specify whether open files are closed in abort mode or normal mode when aborting a Pascal program. **t** deletes and expunges the incomplete file, **nil** writes the incomplete file.

Debugging Pascal Programs

Overview of Debugging Pascal Programs

Genera enables you to debug Pascal programs from the Pascal source level. In the Debugger you examine Pascal variables, values, and types. In addition, you can evaluate expressions and statements from the Debugger.

If you are unfamiliar with the Genera Debugger, you can refer to the Genera documentation set for background information. See the section "Debugger". This discussion assumes you have some knowledge of Debugger concepts and capabilities. In particular, it refers to these terms:

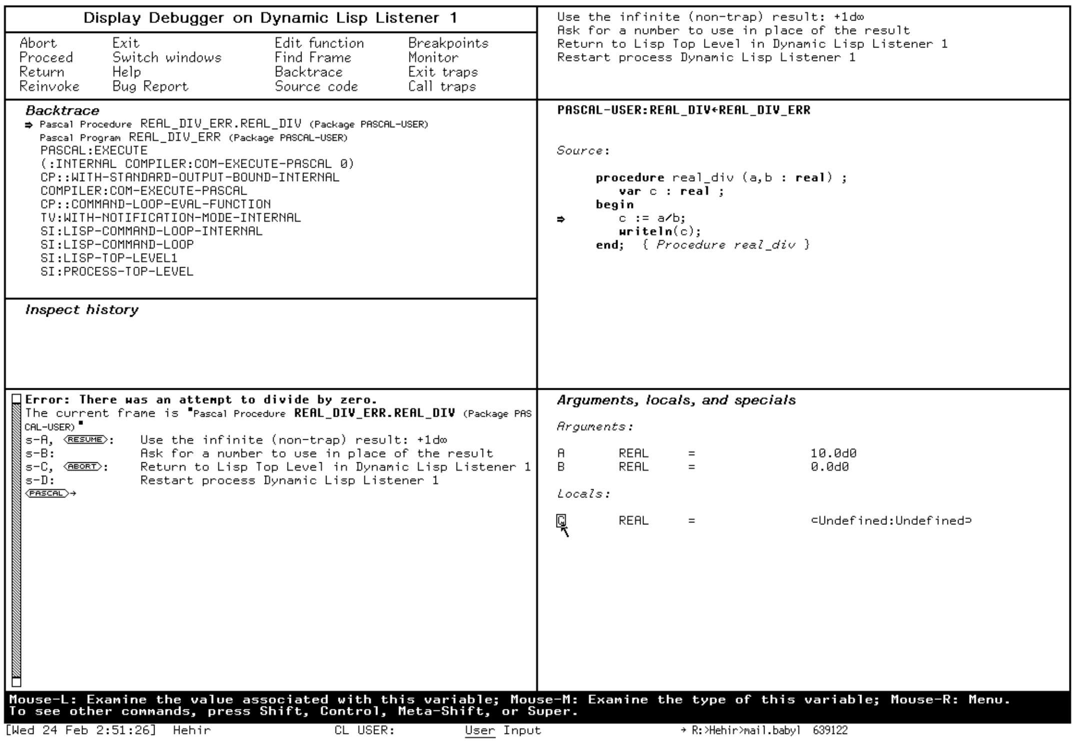
Stack frame	A frame from the control stack that holds the local variables for the routine.
Current stack frame	The context within which Debugger commands operate. The Debugger uses the current frame environment to perform operations according to the suspended state of your program. It evaluates forms in the lexical context of the function suspended in the current frame.

Display Debugger

You can use the Debugger from a Lisp Listener or from the Display Debugger. The Display Debugger is a version of the standard Debugger that uses its own multi-paned window. For further information,

See the section "Using the Display Debugger".

To invoke the Display Debugger, press `c-m-W` from the Pascal Debugger prompt. For an example of a Pascal program in the Display Debugger, see Figure !.



```
Use the infinite (non-trap) result: +1d∞
Ask for a number to use in place of the result
Return to Lisp Top Level in Dynamic Lisp Listener 1
Restart process Dynamic Lisp Listener 1
```

```
Backtrace
⇒ Pascal Procedure REAL_DIV_ERR.REAL_DIV (Package PASCAL-USER)
  Pascal Program REAL_DIV_ERR (Package PASCAL-USER)
  PASCAL:EXECUTE
  (:INTERNAL COMPILER:COM-EXECUTE-PASCAL 0)
  CP::WITH-STANDARD-OUTPUT-BOUND-INTERNAL
  COMPILER:COM-EXECUTE-PASCAL
  CP::COMMAND-LOOP-EVAL-FUNCTION
  TV:WITH-NOTIFICATION-MODE-INTERNAL
  SI:LISP-COMMAND-LOOP-INTERNAL
  SI:LISP-COMMAND-LOOP
  SI:LISP-TOP-LEVEL1
  SI:PROCESS-TOP-LEVEL
```

```
PASCAL-USER:REAL_DIV←REAL_DIV_ERR


Source:

     procedure real_div (a,b : real) ;
        var c : real ;
     begin
⇒       c := a/b;
        writeln(c);
     end;  { Procedure real_div }
```

Inspect history

```
Error: There was an attempt to divide by zero.
The current frame is Pascal Procedure REAL_DIV_ERR.REAL_DIV (Package PAS
CAL-USER)
s-A, <RESUME>:   Use the infinite (non-trap) result: +1d∞
s-B:             Ask for a number to use in place of the result
s-C, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
s-D:             Restart process Dynamic Lisp Listener 1
<PASCAL>→
```

```
Arguments, locals, and specials

Arguments:

A       REAL      =               10.0d0
B       REAL      =               0.0d0

Locals:

C       REAL      =               ⊂Undefined:Undefined⊃
```

```
Mouse-L: Examine the value associated with this variable; Mouse-M: Examine the type of this variable; Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Wed 24 Feb 2:51:26]  Hehir              CL USER:        User Input               → R:>Hehir>mail.babyl  639122
```

Figure 78. A Pascal program in the Display Debugger

Invoking the Debugger for Pascal

Invoking the Debugger

You use the Debugger in these cases:

- 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.
- When you set a breakpoint from the editor.

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. The `ABORT` key is disabled by default in the Display Debugger. The command :Enable Aborts enables it.

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
```

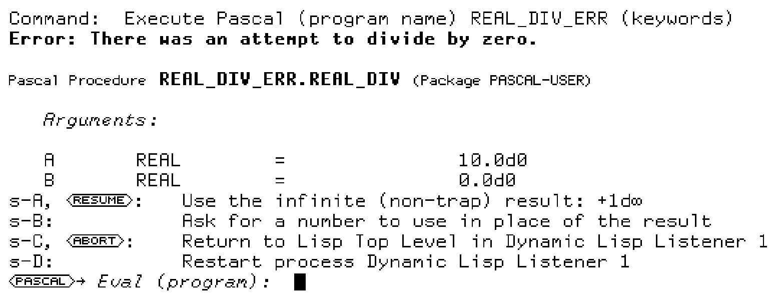
```
These are the possible keyword arguments:
:Detailed            Show locals and disassembled code
:Internal            Show internal interpreter frames
:Nframes             Move this many frames
:To Interesting      Move out to an interesting frame
```

Pascal Frames in the Debugger

When you use the Debugger on a frame compiled in Pascal, 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.

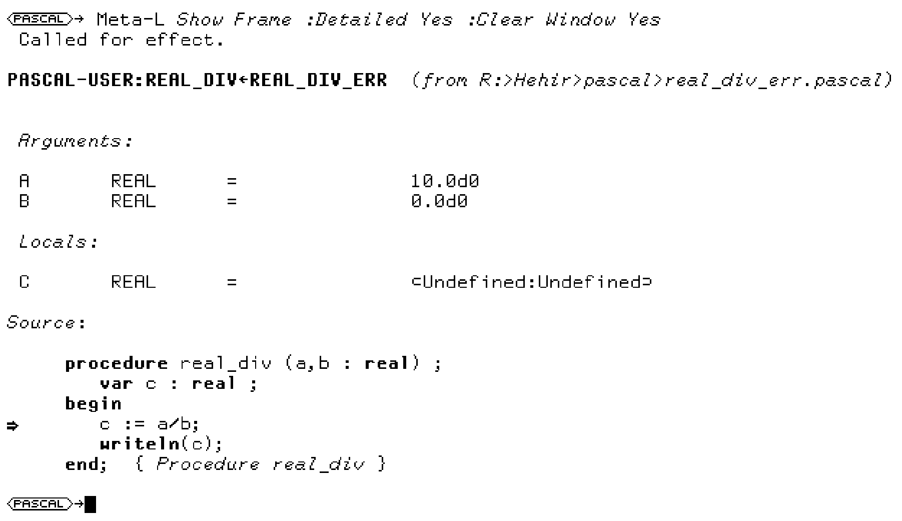
The Debugger prompt indicates whether the frame is compiled in Pascal or in Lisp. The Lisp prompt is a plain arrow. The Pascal prompt is labeled Pascal. You can use most Genera Debugger commands after this prompt.

The next example shows the Debugger operation in the context of a Pascal frame.



```
Command:  Execute Pascal (program name) REAL_DIV_ERR (keywords)
Error: There was an attempt to divide by zero.

Pascal Procedure REAL_DIV_ERR.REAL_DIV (Package PASCAL-USER)

   Arguments:

   A       REAL        =              10.0d0
   B       REAL        =              0.0d0
s-A, <RESUME>:   Use the infinite (non-trap) result: +1d∞
s-B:             Ask for a number to use in place of the result
s-C, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
s-D:             Restart process Dynamic Lisp Listener 1
<PASCAL>→ Eval (program):
```

Using m-L at the Pascal prompt causes the Debugger to display a list of local variables, their values for the current procedure, and the source code in the vicinity of the error (see the figure on the following page). You can use the mouse to manipulate these variables and values, as well as the source code displayed. For example, you can use the mouse to set a breakpoint in the source code, or display a value for a variable. The mouse documentation line at the bottom of the screen describes the options for each item. For further information: See the section "Looking At Variables, Types, and Values in Pascal".



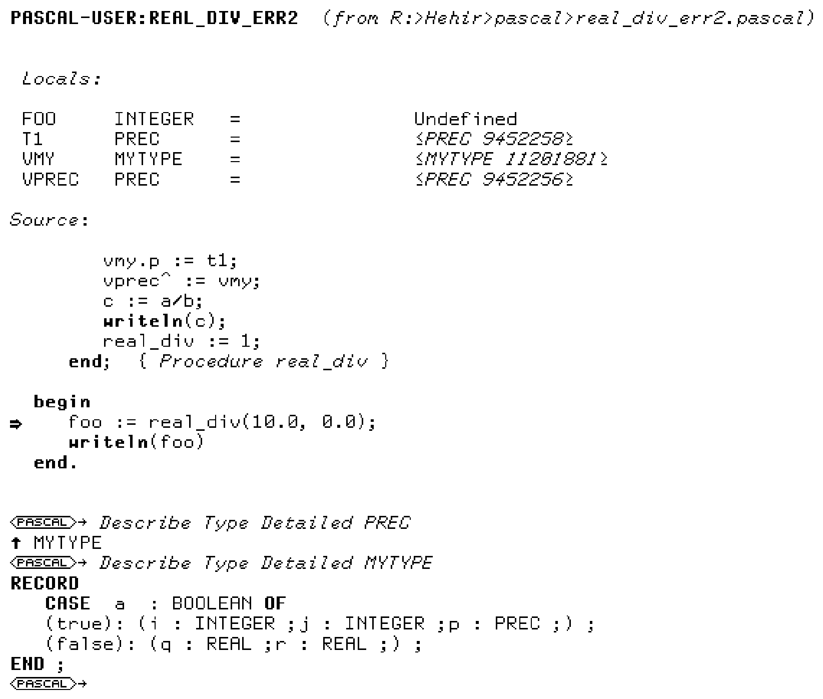
```
<PASCAL>→ Meta-L Show Frame :Detailed Yes :Clear Window Yes
 Called for effect.

PASCAL-USER:REAL_DIV←REAL_DIV_ERR  (from R:>Hehir>pascal>real_div_err.pascal)


 Arguments:

 A       REAL      =               10.0d0
 B       REAL      =               0.0d0

 Locals:

 C       REAL      =               ⊂Undefined:Undefined⊃

Source:

     procedure real_div (a,b : real) ;
        var c : real ;
     begin
⇒       c := a/b;
        writeln(c);
     end;  { Procedure real_div }

<PASCAL>→█
```

Looking At Variables, Types, and Values in Pascal

Debugger output from Pascal frames is mouse-sensitive. In fact, you can operate on local variables and their values solely through the use of the mouse. When you point the mouse cursor at an item, the mouse documentation line displays what action is associated with each mouse click. The following table summarizes operations 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 enables you to inspect a variable, value, or type. In addition, you can get a complete description of a user-defined type as shown in the following example.



```
PASCAL-USER:REAL_DIV_ERR2  (from R:>Hehir>pascal>real_div_err2.pascal)

 Locals:

 FOO     INTEGER   =              Undefined
 T1      PREC      =              ≤PREC 9452258≥
 VMY     MYTYPE    =              ≤MYTYPE 11281881≥
 VPREC   PREC      =              ≤PREC 9452256≥

Source:

        vmy.p := t1;
        vprec^ := vmy;
        c := a/b;
        writeln(c);
        real_div := 1;
     end;  { Procedure real_div }

  begin
⇒    foo := real_div(10.0, 0.0);
     writeln(foo)
  end.


<PASCAL>→ Describe Type Detailed PREC
↑ MYTYPE
<PASCAL>→ Describe Type Detailed MYTYPE
RECORD
   CASE  a  : BOOLEAN OF
   (true): (i : INTEGER ;j : INTEGER ;p : PREC ;) ;
   (false): (q : REAL ;r : REAL ;) ;
END ;
<PASCAL>→
```

This example shows the result of pointing the mouse at the local PREC and clicking the Middle button and then pointing at MYTYPE and again clicking the Middle button.

Expressions and Statements

You can evaluate Pascal expressions and statements from the Debugger. Type the expression in following the Pascal prompt; press END to evaluate the expression.

How Pascal 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*

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

Summarized values

Summarized values are given for objects whose values are too large to be printed out by default. For instance, unless requested explicitly by the user, arrays and structures 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 instance, the following example shows the summarized values for MY_PREC, a user-defined record and MY_TYPE, a user-defined type, such as a record.

≤MY_PREC 9458958≥
≤MY_TYPE 2342256≥

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. For example, an attempt to obtain the value of a variable declared as an integer but actually a real yields a result in this form.

For example,

```
⊂1.3⊃
```

Pascal Language Debugger Commands

Variables, Values, and Types

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

Commands for variables:

	:Show Locals (m-L)	

Commands for values:

	:Show Variable's Value	Examine the value associated with this variable

Commands for types:

	:Show Variable's Type	Examine the type associated with this variable
	:Describe Type Detailed	Describe the type in greater detail
	:Show Type Name	Show the type name
Other:	Edit Viewspecs (menu only)	

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

```
  c-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 for Use with Pascal

You can use Genera Debugger commands from the following areas of functionality in debugging Pascal programs. For more information, see the Genera documentation set.

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

Pascal-Lisp Interaction

Overview

Contents

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

- `lispobject`: a new data type representing Lisp objects in Pascal programs.
- `lisp`: a new Pascal routine directive allowing the declaration of an existing Lisp function, which you can subsequently call from a Pascal routine.
- Calling Pascal from a Lisp function.
- Calling a Lisp function from Pascal.

`lispobject`: Pascal Data Type for Handling Lisp

Description

In addition to the data types defined in the Standards, the Tool Kit supports a new scalar data type, called `lispobject`, which facilitates 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 are not read, written, or even compared against each other. No other type is coerced into a `lispobject`.

`lisp`: Pascal Routine Directive

Introduction

The Pascal Tool Kit supports a new routine directive, known as `lisp`, allowing you to declare a Lisp function you can call from a Pascal procedure or function.

Format

The format for declaring a Lisp function in a Pascal procedure is:

`procedure` *pascal-name-for-lisp-routine* (*parameter-list*);
        `lisp` *'lisp-routine-name'* ;

The format for declaring a Lisp function in a Pascal function is:

`function` *pascal-name-for-lisp-routine* (*parameter-list*) : *output-type*;
        `lisp` *'lisp-routine-name'* ;

Example of Function Declaration

```
Declaration   pascal-name-for-lisp-routine    routine
name          |                               directive
|             |   parameter-list  output-type    |     lisp-routine-name
|             |         |             |          |           |
↓             ↓         ↓             ↓          ↓           ↓
function length (list: lispobject) : integer;  lisp 'global:length' ;
```

Example of Procedure Declaration

```
Declaration  pascal-name-for-lisp-routine
name         |                                 lisp-routine-name
|            | parameter-list  routine directive     |
|            |         |              |              |
↓            ↓         ↓              ↓              ↓
procedure  ball (i, j, k : integer) ;  lisp  'user:ball' ;
```

Description

pascal-name-for-lisp-routine   Specifies the name by which Pascal calls the Lisp routine.

pascal-name-for-lisp-routine is the name of an existing, valid Lisp function (predefined or user-defined) or any valid Pascal identifier that you select to represent a Lisp function.

When the Pascal name for the Lisp routine is identical to the Lisp function name, it is not necessary to specify the *lisp-routine-name* argument. If you do not specify the the *lisp-routine-name*, the Lisp function must have the same name as *pascal-name-for-lisp-routine* and must be in the same package as the Pascal file. Bear in mind, however, that it is considered bad practice to compile Pascal in a package that inherits from **global**.

Make sure the *pascal-name-for-lisp-routine* is a valid Pascal identifier and that it does *not* contain hyphens, colons, and other nonalphabetic characters.

parameter-list Specifies the names and types of the parameters, and follows the same syntactic rules as any Pascal parameter list. The standard Pascal scalar data types are permitted: `integer`, `real`, `shortreal`, `boolean`, `char`, enumerated type, or `lispobject`. These scalar data types are passed by value to Lisp, *not* by reference. See the section "Passing Real Numbers in Pascal".

Array and record parameters are passed either by value or by reference. Multidimensional arrays and records containing arrays are treated as if they were single-dimensional Lisp **art-q** arrays. Note that packed arrays and packed records do not necessarily have one element per Lisp array word. Also, arrays of `reals`, which by default are double-precision floating-point numbers, are stored unpacked.

You can pass packed arrays of characters and Pascal/VS `string` types to Lisp by value or by reference. The Lisp array type **art-string** is used to hold the characters.

Enclose the list of all parameters in parentheses.

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

output-type Specifies the type of the value returned from the function *lisp-routine-name*. Supply any of the standard Pascal scalar data types — `integer`, `real`, `shortreal`, `boolean`, `char`, an enumerated type, or `lispobject`. See the section "Passing Real Numbers in Pascal".

Example 1

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

```
          .
          .
FUNCTION length (list : lispobject) : integer;
                  lisp 'global:length';
VAR a : lispobject;
          .
          .
```

Use the function **length** in the routine.

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

Example 2
The following program passes a Pascal array to the built-in Lisp function **listarray**, which returns a list of the elements. It then calls the Lisp function **reverse** to reverse the `lispobject` list returned, and finally calls the Lisp function **print** to display the result on the console: the list `(10 9 8 7 6 5 4 3 2 1)`, assuming the Lisp output radix is 10. Note that all the parameters are passed by value, except for the *arr* parameter to **listarray**.

```
{-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }

PROGRAM revarray ;
type INTARRAY = ARRAY[1 .. 10] OF integer;
VAR a : intarray;
VAR list : lispobject;
VAR i : integer;
   FUNCTION listarray (VAR arr : intarray) : lispobject;
                        lisp 'global:listarray';
   FUNCTION reverse (list : lispobject) : lispobject;
                      lisp 'global:reverse';
   PROCEDURE print (any_object : lispobject); lisp 'global:print';

BEGIN
     FOR i := 1 TO 10 DO
         a[i] := i;
     list := listarray(a);
     list := reverse(list);
     print(list)
END.
```

Example 3
The Lisp function **screen-clear** refreshes the screen and then **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 Pascal program `throwball` declares the Lisp function **ball**.

```
{-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }

  PROGRAM throwball;
  VAR i : integer;
  PROCEDURE ball (i, j, k: integer) ; lisp 'cl-user::ball';
  PROCEDURE beep ; lisp 'tv:beep';
  PROCEDURE screenclear ; lisp 'cl-user::screen-clear';
BEGIN
      screenclear;
      FOR i := 2 TO 15 DO
              ball(i*20, i*20, 10);
      beep  { get the user's attention }
END.
```

Passing Real Numbers in Pascal

Boxed vs. Unboxed Numbers

`real` numbers are represented as *boxed* numbers in the Lisp world, and as *unboxed* numbers in Pascal. Pascal routines expect to receive data of type `real` in unboxed form. They also return their results unboxed. Note that in the following example the caller of **pascal-user::addup←calculate** first unboxes a double-precision floating-point number to pass to Pascal, and when it gets the result, it boxes the answer.

When Pascal calls a Lisp routine, it boxes any arguments of type `real`; however, it expects `real` results to be unboxed. Therefore, the function **square** doesn't have to box its input, but it has to unbox its result.

To pass `real` numbers to Pascal, you have to first unbox the Lisp number by calling **si:double-components** or **si:with-double-components** on the Lisp double. They return two "integers", x and y, representing the high and low portions of the number. Conversely, you can call **si:%make-double** on x and y to produce a boxed double for Lisp to manipulate.

Example

```
;;; -*- Mode: LISP; Package: CL-USER; Base: 10; Syntax: COMMON-LISP -*-
```

```
;;;Calls the Pascal routine addup, which adds an
;;;integer, a single-float, and the square of a
;;;double-float.  Returns the answer as a double-float.
(defun callpascal (integer single double)
  (si:with-double-components (double double-hi double-lo)
    (multiple-value-bind (result-hi result-lo)
        (pascal-user:addup←calculate nil integer single
                                     double-hi double-lo)
      (si:%make-double result-hi result-lo))))

;;;Takes a double-float number from Pascal and
;;;returns the square of the number.
(defun square (double)
  (si:with-double-components ((* double double) hi lo)
    (values hi lo)))
```

```
{-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }
```

PROGRAM calculate;
  {Calls Lisp to get the square of a real number.}
  FUNCTION square(dub: real): real; lisp 'cl-user::square';

  {Adds an integer, a shortreal, and the square of a real number.}
  {Returns the answer as a real number.}
  FUNCTION addup(int: integer; sng: shortreal; dub: real): real;
  BEGIN
    addup := int + sng + square(dub);
  END;
BEGIN
END.

Calling Pascal from Lisp

Before reading this section: see the section "Executing Pascal Programs". see the section "Lisp Syntax for Pascal Users".

If you are passing real numbers to Pascal or receiving a real number as a result: see the section "Passing Real Numbers in Pascal".

Calling a Main Program From Lisp

A Lisp function can call a Pascal main program directly.

Example: To call the Pascal main program `writeones` from the Lisp function **callwrites** and specify Pascal standard output at run time, type:

```
(defun callwrites (file)
  (pascal:execute 'pascal-user:writeones :output file))
```

To run **callwrites** designating Pascal standard output as "s:>stryker>ones.data", type:

```
(callwrites "s:>stryker>ones.data")
```

or

```
(pascal:execute 'writeones "s:>stryker>ones.data")
```

In figure !, the Lisp function **callwrites**, shown in the top pane, calls the Pascal program `writeones`, associating Pascal standard output with the output file "s:>stryker>ones.data". The bottom pane displays the Pascal source code.

Calling a First-level Internal Routine From Lisp

The format of a Lisp function calling a top-level Pascal routine is as follows:

```
(package:top-level-routine-name compiler-display arguments)
```

package. The name of the pascal package or **user**, the default package.

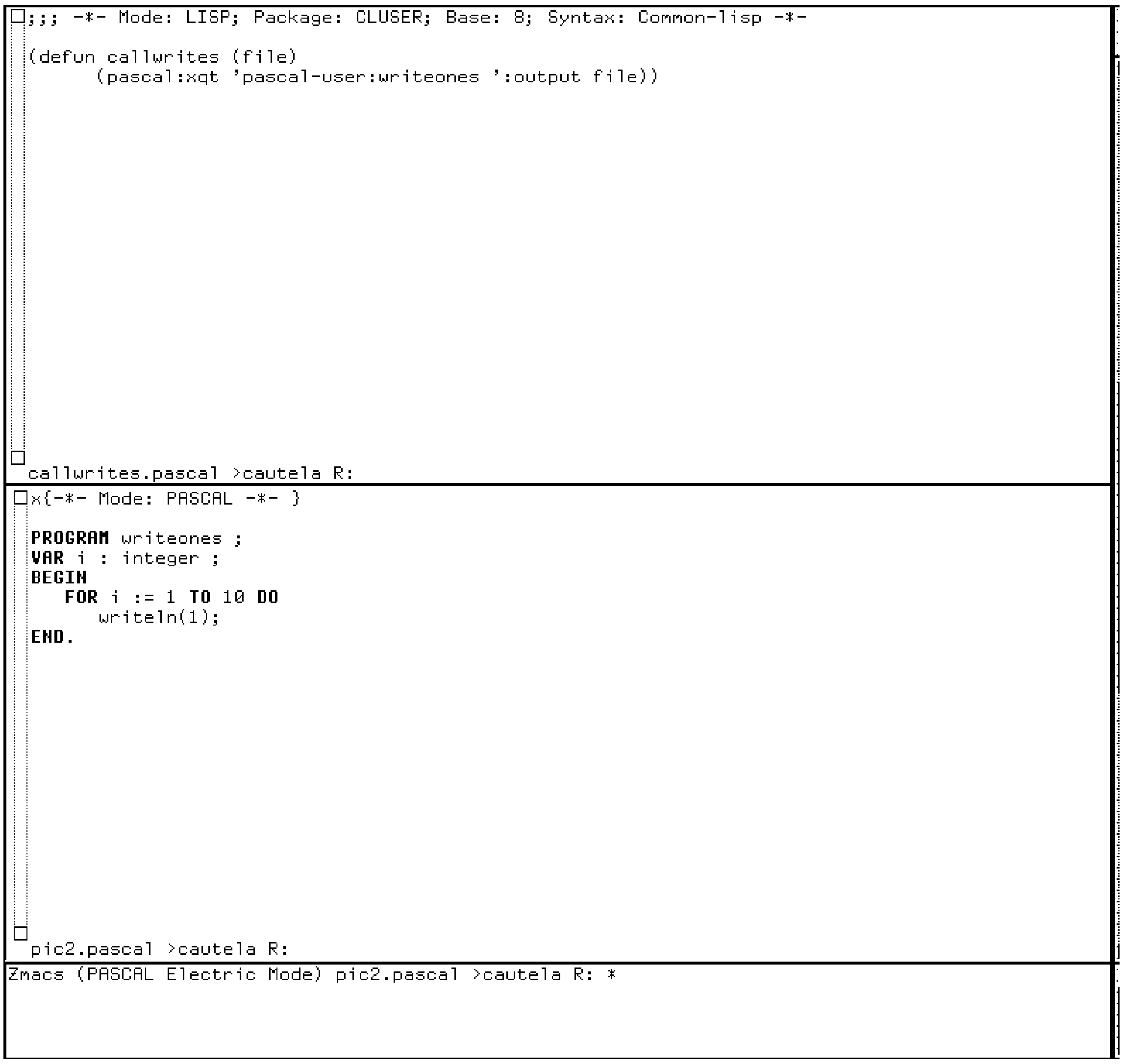
top-level-routine-name. The symbol made from the concatenation of the name of a top-level routine with that of the calling program, with a backarrow character (←) in between. For example, the symbol of a first-level routine `first` called by program `main` is **first←main**.

compiler-display. The lexical environment of *top-level-routine-name* generated by the Pascal compiler. For procedures and functions at top level the display is always **nil**.

arguments. The arguments to *top-level-routine-name*.

Example: The Lisp routine **factor** calls the Pascal routine `factorial`. The source code of both routines is shown.

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defun factor ()
 (let ((x (prompt-and-read
            :number
            "Find the factorial of what number? ")))
  (format t "Factorial of ~D is: ~D "
          x (pascal-user:factorial←factor nil x))))


{-*- Mode: PASCAL; Dialect: ISO; Package: PASCAL-USER -*- }

PROGRAM factor;
VAR x : integer;
```



```
;;; -*- Mode: LISP; Package: CLUSER; Base: 8; Syntax: Common-lisp -*-

(defun callwrites (file)
      (pascal:xqt 'pascal-user:writeones ':output file))
```

callwrites.pascal >cautela R:

```
x{-*- Mode: PASCAL -*- }

PROGRAM writeones ;
VAR i : integer ;
BEGIN
   FOR i := 1 TO 10 DO
      writeln(1);
END.
```

pic2.pascal >cautela R:

Zmacs (PASCAL Electric Mode) pic2.pascal >cautela R: *

Figure 79. A Lisp function calls a Pascal main program.

```
FUNCTION factorial (n : integer) : integer;
VAR i : integer;
BEGIN
    factorial := 1;
    FOR i := n DOWNTO 1 DO
      factorial := factorial * i
END; { Function factorial }
```

```
BEGIN
  writeln;
  write('Find the factorial of what number? ');
  readln(x);
  writeln('Factorial ',x:1,' is ',factorial(x):1)
END.
```

Editor Extensions for Pascal

This section contains a brief summary of the commands available for editing Pascal programs in Zmacs, as well as for finding out about and interacting with your Pascal program. Note: 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*. Example: The cursor is positioned in the middle of a routine. To move forward from the current routine to the end of the next routine, type:

`c-2 c-m-E`

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

`c-m-A` Moves the cursor backward to the beginning of a Pascal routine.

`c-m-E` Moves the cursor forward to the end of a Pascal routine.

`c-sh-C` Compiles 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. Region commands are explained in Region Operations: See the section "Summary of Standard Editing Features".

The echo area at the bottom of the screen displays the names of the routines being compiled. An unsuccessful compilation results in the display of compiler warnings and suggested actions in a typeout window at the top of the screen. Typing any character causes the window to disappear.

`m-X` Clear All Breakpoints Clears all breakpoints in the buffer.

`m-X` Compile And Execute Pascal Program
Checks to see that the cursor is positioned near a valid, compiled Pascal program and then compiles and executes the program, without run-time options, 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. This feature is useful for resuming compilation when a previous attempt has failed.

m-X Compile File Compiles a file, offering to save it first if the buffer has been 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*, a contiguous delimited section of text in the editor buffer. 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-1*, 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 Edit Definition or m-. Edits the definition of a compiled Pascal routine. When it prompts for a 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) use the mouse to select a name 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 Pascal system, whether or not it is in a file that is currently in a buffer.

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

m-X Electric Pascal Mode Turns on Electric Pascal mode, or, if it is on, turns it off. This command works only when the buffer is in Pascal mode.

m-X Execute Pascal Program Checks to see that the cursor is positioned near a valid, compiled Pascal program and then executes the program, without runtime options, with predefined files' input and output bound to the editor typeout window.

m-X Format Language Region Adds formatting (face, case, and correct capitalization) to an editor region (just as if the code were typed in Electric Pascal mode). If no region is defined, it acts on the current Pascal routine definition. With a numeric argument, it removes the formatting.

m-X List Breakpoints Lists all currently active breakpoints.

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 programs, procedures, 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 Reparse Attribute List Causes all changes made to the attribute list to take effect.

m-X Set Dialect Sets the dialect for the buffer. When it prompts for a name, type either iso or vs; the default is to toggle the dialect. It offers to set the dialect for the attribute list as well as for the buffer. All routines in the buffer will be compiled in the chosen dialect.

m-X Set Package Sets the package for the buffer. When it prompts for a name, type the name of a package. It offers to create the package if it does not exist. The commands also offers to set the package for the attribute list as well as for the buffer.

m-X Show Dependent Units Prompts for the name of a Pascal unit specification and displays a list of those units dependent on that specification.

m-X Show Units Depended On Prompts for the name of a Pascal program or unit and displays a list of those units depended on by that program or unit.

m-X Update Attribute List Creates or updates the attribute list for the file. Executing the command after entering Pascal mode causes the attribute list to include the phrase:

```
Mode : Pascal;
```

.

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.

Summary: Pascal Editor Mode Commands

Cursor movement commands

Keystroke	*Meaning*
c-m-F	Moves the cursor to the end of the current or next language-specific unit.
c-m-B	Moves the cursor to the start of the current or previous language-specific unit.
c-m-A	Moves the cursor to the start of the current or previous language definition.
c-m-E	Moves the cursor to the end of the current or next language definition.
c-sh-F	Moves the cursor to the end of the current or next language expression.
c-sh-B	Moves the cursor to the start of the current or previous language expression.
c-m-H	Moves the cursor to the start of the current language definition and marks the entire definition as a region. The editor underlines the region.
c-m-N	Moves the cursor to the next template in the buffer, if any.
c-m-P	Moves the cursor to the previous template in the buffer, if any.

Deletion commands

Keystroke	*Meaning*
m-sh-X	Deletes the language expression to the left of the cursor.
c-sh-X	Deletes the language expression to the right of the cursor.
c-m-RUBOUT	Deletes the language construct to the left of the cursor.
c-m-K	Deletes the language construct to the right of the cursor.
c-sh-K	Deletes the language construct around point (the cursor).

c-sh-T	Deletes the language token (for example, an identifier or comment) to the right of the cursor.
m-sh-T	Deletes the language token (for example, an identifier or comment) to the right of the cursor.

Syntax Error Detection Commands

Keystroke	*Meaning*
c-sh-N	Finds the nearest syntax error to the right of the cursor, if any, and moves the cursor there. With a numeric argument, it finds the last syntax error in the buffer.
c-sh-P	Finds the nearest syntax error to the left of the cursor and moves the cursor there. With a numeric argument, it finds the first syntax error in the buffer.

Template and Completion Commands

Keystroke	*Command*
END	Inserts a template that matches the keyword to the right of the cursor.
c-END	Inserts whatever uniquely closes a language construct to the left of the cursor. For example, s-"C inserts a close bracket ("]") to match a ("["), or an then to match an if.
COMPLETE	Completes a keyword to the left of the cursor or further fills in the current template.
c-HELP	Provides a list of templates for valid language constructs to be inserted at the cursor.
c-?	Lists in an editor typeout window the possible completions of predeclared identifiers for the name immediately to the left of the cursor. This usage is specific to Pascal editor mode.
m-X Remove Template Item	Deletes the next template to the right of the cursor.
SPACE	Deletes the next template item to the right of the cursor.
c-m-N	Moves the cursor to the next template in the buffer.
c-m-P	Moves the cursor to the previous template in the buffer.

Indentation Commands

Keystroke	*Command*
c-m-Q	Corrects the indentation of the language structure following point (cursor position).

LINE	Indents the current line correctly with respect to the line above it. It also positions the cursor on the next line and aligns it with the preceding line. LINE opens a new blank line. If a syntax error is found on that line, the editor points out the error.
TAB	Indents the current line correctly with respect to the line above it and positions the cursor at the first character on the line.
m-X Save Indentation	After using c-I to change global indentation of Pascal language constructs, the command produces a Lisp form reflecting the new indentation values; evaluate this form after the Pascal editor is loaded.

Formatting Commands

Keystroke	*Command*
m-X Adjust Face and Case	Modifies the face and case settings for a particular language or dialect.
m-X Blink Matching Construct	Allows you to check that block constructs are balanced. When you turn the feature on and position the cursor at a reserved word that closes a block statement, the editor flashes the reserved word that opens the block statement. For example, positioning the cursor at end if makes the matching if construct blink. Invoke the command again to turn this feature off; off is the default condition.
m-X Electric Pascal Mode	Turns on Electric Pascal mode, or, if it is on, turns it off. Once the mode is on, you can use the Adjust Face and Case command. The Electric Pascal Mode command works only when the buffer is in Pascal mode.
m-X Format Language Region	Conforms the face and case in the region to the settings for the buffer. A numeric argument removes any special typefaces from the region but leaves the case untouched.

Sublicense Addendum for Symbolics Pascal

Your purchase of Symbolics Pascal under the *Terms and Conditions of Sale, License, and Service* (3/89) allows you to use this product on a designated processor. Customers who distribute an application that includes the Pascal run-time system **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 the Symbolics Pascal run-time system. 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 application internally — to end users who work for your company. To help ensure the quality of future releases of the run-time system, Symbolics requests that you provide us with a copy of your sublicensed application program.*

Sublicense Addendum to Symbolics Inc. Terms and Conditions of Sale, License and Service (3/89)

Addendum made this _____ day of __________, 198__, ("this Addendum") to Symbolics, Inc. Standard Terms and Conditions of Sale, License and Service (3/89) dated, _______, 198__, (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**	**Release**
SLAN-Pascal	Symbolics Pascal	5.2

2. **Right of Sublicense**.

Customer may sublicense all or any portion of the run-time system binary code (the "Code") of the Software to Customer's end users provided that:

(i) such Code is part of Customer's application software program sublicensed to such end users;

(ii) the Customer's application software program is licensed by Customer to Customer's end users to run on a Symbolics computer system or processor; and

(iii) Symbolics' copyright and trademark notices shall not be removed from the Software.

3. **End User**.

The term "end user" for the purposes of this Addendum shall mean Customer's customers and includes Customer's own internal end users of its application software programs.

CUSTOMER	SYMBOLICS, INC.
______________________________	______________________________
Name	Name
______________________________	______________________________
Title	Title
______________________________	______________________________
(Address)	(Address)