```
28  19  map_input
```

will do the trick. Here 28 is the ascii character of `^\` and 19 is the ascii character for the ^S.

As another example, consider the case where the backspace key sends out a ^H instead of the `DEL` character (`^?`).

```
8  127 map_input
```

will map the ^H (8) to the delete character (127).

0.20.3 Display Sizes

On VMS and unix systems, the screen size may be changed to either 80 or 132 columns by using the functions `w80` and `w132` respectively. Simply enter the appropriate function name at the `M-x` prompt in the minibuffer. The default binding for access to the minibuffer is `ESC X`. Most window systems, e.g., DECWindows, allow the window size to be changed. When this is done, JED should automatically adapt to the new size.

On the PC, at this time the screen size cannot be changed while JED is running. Instead it is necessary to exit JED first then set the display size and rerun JED.

and to the DOS 437 code page for the IBMPC. To change the defaults, it is only necessary to load the appropriate file. For example, to load '`dos850.sl`' definitions, put

```
"dos850.sl" evalfile pop
```

in the startup file (e.g., '`site.sl`'). In addition to uppercase/lowercase information, these files also contain word definitions, i.e., which characters constitute a "word".

0.20 Miscellaneous

0.20.1 Abort Character

The abort character (^G by default) is special and should not be rebound. On the IBMPC, the keyboard interrupt 0x09 is hooked and a quit condition is signaled when it is pressed. For this reason, it should not be used in any keybindings. A similar statement holds for the other systems.

This character may be changed using the function **set_abort_char** Using this function affects all keymaps. For example, putting the line

```
30 set_abort_char
```

in your '`jed.rc`' file will change the abort character from its current value to 30 which is Control-^.

0.20.2 Input Translation

By using the function **map_input** the user is able to remap characters input from the terminal before JED's keymap routines have a chance to act upon them. This is useful when it is difficult to get JED to see certain characters. For example, consider the ^S character. This character is especially notorious because many systems use it and ^Q for flow control. Nevertheless Emacs uses ^S for searching. Short of rebinding all keys which involve a ^S how does one work with functions that are bound to key sequences using ^S? This is where **map_input** comes into play. The **map_input** function requires two integer arguments which define how a given ascii character is to be mapped. Suppose that you wish to substitute ^\ for ^S everywhere. The line

key `. If the `quoted_insert` function is called with a digit argument (repeat argument), the character with the value of the argument is inserted into the buffer. Operationally, one hits `ESC`, enters the extended Ascii code and hits the backquote key. For example, to insert character 255 into the buffer, simply press the following five keys: `ESC 2 5 5 `.

0.19.3 Upper Case - Lower Case Conversions

The above discussion centers around input and output of characters with the high bit set. How JED treats them internally is another issue and new questions arise. For example, what is the uppercase equivalent of a character with ASCII code 231? This may vary from language to language. Some languages even have characters whose uppercase equivalent correspond to multiple characters. For JED, the following assumptions have been made:

1. Each character is only 8 bits.
2. Each character has a unique uppercase equivalent.
3. Each character has a unique lowercase equivalent.

It would be nice if a fourth assumption could be made:

4. The value of the lowercase of a character is greater than or equal to its uppercase counterpart.

However, apparently this is not possible since most IBMPC character sets violate this assumption. Hence, JED does not assume it. Suppose X is the upper case value of some character and suppose Y is its lower case value. Then to make JED aware of this fact and use it it case conversions, it may be necessary to put a statement of the form:

$$X\ Y\ \texttt{define_case}$$

in the startup file. For example, suppose 211 is the uppercase of 244. Then, the line '`211 244 define_case`' will make JED use this fact in operations involving the case of a character.

This has already been done for the ISO Latin 1 character set. See the file '`iso-latin.sl`' for details. For MSDOS, this will not work. Instead use the files '`dos437.sl`' and '`dos850.sl`'. By default, JED's internal lookup tables are initialized to the ISO Latin set for Unix and VMS systems

0.19.1 Displaying Characters with the High Bit Set

There are several issues to consider here. The most important issue is how to get JED to display 8 bit characters in a "clean" way. By "clean" I mean any character with the high bit set is sent to the display device as is. This is achieved by putting the line:

```
1 =DISPLAY_EIGHT_BIT
```

in the '`jed.rc`' ('`.jedrc`') startup file. European systems might want to put this in the file '`site.sl`' for all users. The default is 1 so unless its value has been changed, this step may not be necessary.

There is another issue. Suppose you want to display 8 bit characters with extended Ascii codes greater than or equal to some value, say 160. This is done by putting '`160 =DISPLAY_EIGHT_BIT`'. I believe that ISO Latin character sets assume this. This is the default value for Unix and VMS systems. See also the file '`iso-latin.sl`'.

0.19.2 Inputting Characters with the High Bit Set

Inputting a character into JED with the high bit set is another issue. How JED interprets this is controlled by the variable `META_CHAR`. Specifically what happens is this: When JED reads a character from the input device with the high bit set, it:

1. Checks the value of `META_CHAR`. If this value is -1, JED simply inserts the character into the buffer.
2. For any other value of `META_CHAR` in the range 0 to 255, JED returns two 7-bit characters. The first character returned is `META_CHAR` itself. The next character returned is the original character but with the high bit stripped.

The default value of `META_CHAR` is -1 which means that when JED sees a character with the high bit set, JED leaves it as is. Please note that a character with the high bit set *cannot* be the prefix character of a keymap. It can be a part of the keymap but not the prefix.

Some systems only handle 7-bit character sequences and as a result, JED will only see 7-bit characters. JED is still able to insert *any* character in the range 0-255 on a 7-bit system. This is done through the use of the `quoted_insert` function which, by default, is bound to the backquote

Simply enter the above into an empty JED `*scratch*` buffer, then press `^X ESC` and enter '`1 =_traceback evalbuffer fun_one`'. This will turn on tracebacks, evaluate the buffer and call the function '`fun_one`'. JED will then be put into an infinite loop which can only be stopped by pressing the abort character which by default is ^G. Doing so, will produce the traceback messages

```
S-Lang Traceback: fun_two
S-Lang Traceback: fun_one
```

in addition to the error message '`User Break!`'. Of course, this technique only narrows down the source of an error to a particular function. To proceed further, it may necessary to put "print" statements at suitable places in the function. There are several ways to do this:

1. Use the `insert` function to insert the contents of a variable into the current buffer.
2. Use the `error` function to abort the function and display the value of a variable in the minibuffer.
3. Use the `message` function to display the value of a variable in the minibuffer. Unlike `error`, the `message` function does not abort the execution of the function.

Since each of these functions require a string argument, it is usually best to call the `string` function first for the conversion followed by the output function. This has to be done anyway if it is desired to get the contents of an integer variable. Although the second approach is prehaps the most useful in practice, it is somtimes appropriate to use a combination of these techniques.

Finally, to print the entire stack, one can use the `print_stack` function. This function dumps the S-Lang runtime stack into the '`*traceback*`' buffer.

Since S-Lang is an interpreted language, judicious application of the above techniques should lead very quickly to the source of any errors.

0.19 Eight Bit Clean Issues

A related hook called after a paragraph is formatted is **format_paragraph_hook**. This hook is only called if either **format_paragraph** or **narrow_paragraph** is called with a prefix digit argument. For example, **format_paragraph** is bound to **ESC q**. Simply pressing this key sequence will call **format_paragraph** but **format_paragraph_hook** will not be called. However, pressing **ESC 1** followed by **ESC q** will result in a call to **format_paragraph_hook**. Currently, this hook simply justifies the paragraph. That is, it fills each line in the paragraph such that the the line ends at the right margin, which is defined by the **WRAP** variable.

0.18.4 S-Lang Programming Hints (Debugging)

This section assumes some knowledge about S-Lang and is designed to explain how to debug S-Lang routines quickly. For information about S-Lang, read '**slang.txt**'.

There are two ways of loading a file of S-Lang code into JED. The most common way is through the function **evalfile**. If an error occurs while loading a file, JED will give some indication of where the problem lies by displaying the line number and the offending bit of S-Lang code in the minibuffer. In practice though, this can be quite inefficient. The **evalfile** function is primarily designed to load debugged and tested S-Lang code.

The best way to develop and test S-Lang code with JED is to use the function **evalbuffer**. Simply load the piece of code into JED as an ordinary file, press **ESC X** and enter the function **evalbuffer** If the piece of code in the buffer has any syntax errors, JED will put the cursor on the error. This is the best way to spot compile time errors such as syntax errors. However, this will not catch runtime errors.

When a runtime error occurs, JED will put the cursor on the top level function where the original call was made and NOT the actual location of the function. To aid in determining where an error occurs, JED can be made to give a symbolic traceback. As the S-Lang runtime stack unwinds, S-Lang will simply print the name of function at that particular level. If the function includes local variables, their values will be dumped as well. Hence, it is easy to quickly narrow the location of an error down to function where the error occurs. By default, the traceback is disabled. The traceback is enabled by setting the S-Lang variable **_traceback** to a non-zero value. It is simpliest to just press **^X ESC** and enter '**1 =_traceback**' at the JED prompt. This is one of those times where one needs access to the '**S-Lang>**' prompt and not the '**M-x**' prompt. For example, consider the following piece of code:

```
( {}  forever ) fun_two  ;; loops forever
( fun_two ) fun_one      ;; calls fun_two-- never returns
```

function which loads the '`jed.rc`' startup file. Unlike the other hooks, this one must be present in the file '`site.sl`' since it is the only file loaded before calling the hook.

After the startup files are loaded, JED calls the hook `jed_startup_hook` immediately before entering the main editor loop. This hook is useful to modify certain data structures which may not have existed when the startup files were loaded.

In addition to the above hooks, JED currently also looks for:

`suspend_hook`

: function to be executed before suspending

`resume_hook`

: function that gets carried out after suspension

`exit_hook`

: gets executed before exiting JED

`mode_hook`

: sets buffer mode based on filename extension

`find_file_hook`

: called before file is read into a buffer. It currently checks for presence of autosave file and warns user if it is more recent than file.

See '`site.sl`' for explicit examples of the above hooks.

Another useful hook is `is_paragraph_separator`. This hook is called when JED searches for the beginning or end of a paragraph. This search is performed by all paragraph formatting functions as well as the forward and backward paragraph movement commands. As JED performs the search, it moves from one line to another testing the line to see if it separates a paragraph. The function of the hook is to make this decision and return zero if the line does not separate paragraphs or return one if it does. The default value of this hook may be written in S-Lang as

```
( bol
  "\\" looking_at {1 return} if
  "%"  looking_at {1 return} if
  skip_white eolp
) is_paragraph_separator
```

0.18.2 Predefined Variables

JED includes some predefined variables which the user may change. By convention, predefined variables are in uppercase. The variables which effect all modes include:

`BLINK` (1) if non-zero, blink matching parenthesis.

`TAB_DEFAULT`
(8) sets default tab setting for newly created buffers to specified number of columns.

`TAB` Value of tab setting for current buffer.

`ADD_NEWLINE`
(1) adds newline to end of file if needed when writing it out to the disk.

`META_CHAR`
(-1) prefix for chars with high bit set (see section on eight bit clean issues for details)

`DISPLAY_EIGHT_BIT`
see section on eight bit clean issues.

`COLOR` (23) IBMPC background color (see '`jed.rc`' for meaning)

`LINENUMBERS`
(0) if 1, show current line number on status line

`WANT_EOB` (0) if 1, [EOB] denotes end of buffer.

`TERM_CANNOT_INSERT`
(0) if 1, do not put the terminal in insert mode when writing to the screen.

`IGNORE_BEEP`
(0) do not beep the terminal when signalling errors

In addition to the above, there are variables which affect only certain modes. See the section on modes for details.

0.18.3 Hooks

A hook is a user defined function that JED calls under certain conditions which allow the user to modify default actions. For example, when JED starts up it looks for the existence of a user defined function `command_line_hook`. If this function exists, JED calls the function. What the function does is completely arbitrary and is left to the discretion of the user. The startup file, '`site.sl`', defines such a function which reads in the files listed on the command line. It is also this

Notice two things. First of all, the key sequence **ESC m** has been written as '**"\em"**' where '**\e**' will be interpreted by JED as **ESC**. The other salient feature is that the first argument to **setkey**, the "function" argument, begins with a space. This tells JED that it is not be be interpreted as the name of a function; rather, the characters following the space are to be inserted into the buffer. Omitting the space character would cause JED to to execute a function called '**int main(int argc, char **argv)**' which would fail and generate an error.

Finally, it is possible to define a key to execute a series of keystrokes similar to a keyboard macro. This is done by prefixing the "function" name with the '**@**' character. This instructs JED to interpret the characters following the '**@**' character as characters entered from the keyboard and execute any function that they are bound to. For example, consider the following key definition which will generate a C language comment to comment out the the current line of text. In C, this may be achieved by inserting symbol '**"/*"**' at the beginning of the line and inserting '**"*/"**' at the end of the line. Hence, the sequence is clear (Emacs keybindings):

1. Goto the beginning of the line: ^A or decimal '**"\001"**'.
2. Insert '**"/*"**'.
3. Goto end of the line: ^E or decimal '**"\005"**'
4. Insert '**"*/"**'.

To bind this sequence of steps to the key sequence **ESC ;**, simply use

```
"@\001/*\005*/"   "\e;"   setkey
```

Again, the prefix '**@**' lets JED know that the remaining characters will carry out the functions they are currently bound to. Also pay particular attention to the way ^A and ^E have been written. Do not attempt to use the ^ to represet "control". It does not have the same meaning in the first argument to the **setkey** function as it does in the second argument. To have control characters in the first argument, you must enter them as '*xyz*' where *xyz* is a three digit decimal number coinciding with the ASCII value of the character. In this notation, the **ESC** character could have been written as '**\027**'. See the S-Lang Programmer's Reference Manual for further discussion of this notation.

The **setkey** function sets a key in the **global** keymap from which all others are derived. It is also possible to use the function **local_setkey** which operates only upon the current keymap which may or may not be the **global** map.

0.18 Customization

To extend JED, it is necessary to become familiar with the S-Lang programming language. S-Lang not a standalone programming language like C, Pascal, etc... Rather it is meant to be embedded into a C program. The S-Lang programming language itself provides only arithmetic, looping, and branching constructs. In addition, it defines a few other other primitive operations on its data structures. It is up to the application to define other built-in operations tailored to the application. That is what has been done for the JED editor. See the document '`slang.txt`' for S-Lang basics as well as the JED Programmer's Manual for functions JED has added to the language. In any case, look at the '`*.sl`' files for explicit examples.

For the most part, the average user will simply want to rebind some keys and change some variables (e.g., tab width). Here I discuss setting keys and the predefined global variables.

0.18.1 Setting Keys

Defining a key to invoke a certain function is accomplished using the `setkey` function. This function takes two arguments: the function to be executed and the key binding. For example, suppose that you want to bind the key ^A to cause the cursor to go to the beginning of the current line. The JED function that causes this is `bol` (See the JED Programmer's Manual for a complete list of functions). Putting the line:

```
"bol"   "^A"   setkey
```

in the startup file '`jed.rc`' ('`.jedrc`') file will perform the binding. Here `^A` consists of the two characters `^` and `A` which JED will interpret as the single character `Ctrl-A`. For more examples, see either of the S-Lang files '`emacs.sl`' or '`edt.sl`'.

In addition to being able to define keys to execute functions, it is also possible to define a key to directly insert a string of characters. For example, suppose that you want to define a key to insert the string '`int main(int argc, char **argv)`' whenever you press the key `ESC m`. This may be accomplished as follows:

```
" int main(int argc, char **argv)"   "\em"  setkey
```

The default binding for the mail is ^X M. Alternatively, one may press ESC X and enter mail at the 'M-x' prompt. The mail function will cause a window to open with a buffer called *mail* which contains the three lines:

```
To:
Subject:
---text follows this line---
```

Simply enter the email address of the person that you want to send the mail to on the line containing 'To:' and put the subject of the message on the next line labeled 'Subject:'. The text that you wish to mail follows the line labeled '---text follows this line---' which is used by JED as a marker. After you have composed the mail message, press ESC X and enter send at the M-x prompt. For example, the following is an email requesting to be put on the JED mailing list:

```
To: davis@amy.tch.harvard.edu
Subject: jed mailing list
---text follows this line---
Hi,

   Please add me to the JED mailing list so that I may be notified
of upcoming releases of JED.

--Maria
```

For VMS systems, the above example will probably fail because an internet address has been used for the example. For systems using a TCP/IP package, it may be necessary to change 'davis@amy.tch.harvard.edu' to something like 'smtp%"davis@amy.tch.harvard.edu"'.

The mail function looks for a user defined hook called 'mail_hook' and execute it if it exists. This hook may be used to bind certain keys in the keymap associated with the *mail* buffer. For example,

```
( "^C"  local_unsetkey "send" "^C^C" local_setkey ) mail_hook
```

defines the key ^C^C in the mail keymap to perform the send function. Other possibilities include binding a key sequence, say ^C^W, to a function that inserts the contents of a signature file.

0.16 Dired— the Directory editor

In addition to editing files, JED is also able to rename and delete them as well. JED's Dired mode allows one to do just this is a simple and safe manner.

To run dired, simply press `ESC X` and enter `dired` at the prompt. JED will load '`dired.sl`' and prompt for a directory name. Once the directory is given, JED will display a list files in the directory in a buffer named '`*dired*`'. One may use normal buffer movement keys to move around this buffer. To delete one or more files, use the `d` key to "tag" the files. This in itself does not delete them; rather, it simply marks them for deleting. A capital 'D' will appear in the left margin to indicate that a file has been tagged. Simply hit the `U` key to untag a file. The delete key will also untag the previously tagged file.

To actually delete the tagged files, press the 'x' key. This action causes JED to display a list of the tagged files in a separate window and prompt the user for confirmation. Only when the proper confirmation is given, will the file be deleted.

Renaming a file is just as simple. Simply move to the line containg the name of the file that you wish to rename and hit the 'r' key. JED will prompt for a filename or a directory name. If a directory is given, the file will be moved to the new directory but will keep the name. However, for the operation to succeed, the file must be one the same file system. To rename tagged files to a different directory residing on the same file system, use the `m` key. This has the effect of moving the tagged file out of the current directory to the new one.

One may also use the `F` key to read the file indicated by the cursor position into a buffer for editing. If the file is a directory, the directory will be used for dired operations. In addition, one may also use the `V` to simply "view" a file.

Finally, the `g` key will re-read the current directory and the `h` and `?` keys provide some help.

0.17 Mail

This section applies to Unix and VMS systems only. On these systems, it is possible to compose and send mail directly using JED. This assumes that the Unix system has '`/usr/ucb/mail`'. It is trivial to modify '`mail.sl`' to support another Unix mailer. For VMS, JED uses the callable mail interface present on VMS versions 5.0 and later.

other editors, logging in to remote systems, et cetera. Even '`chdir`' is stupid since its effect is not permanent. That is,

```
> cd ..
> dir
```

will not do what might naively be expected. That is, the two commands above are not equivalent to the single command '`dir ..`'.

0.14 Getting Help

JED's help functions are bound to ^H by default. For example, `^H C` will show what function a key carries out, `^H i` will run JED's info reader, `^H f` will give help on a particular S-Lang function, etc... However, some modes may use the ^H key for something else. For example, if EDT mode is in effect, then ^H may be bound to `bol` which causes the cursor to move to the beginning of the line. See the section on EDT for more information.

If JED is properly installed, this entire document is accessable from within the editor using JED's info reader. `^H i` will load `info_mode` allowing the user to browse the document as well as other "info" documents.

0.15 Editing Binary Files

JED may edit binary files as long as the proper precautions are taken. On IBMPC systems, this involves calling the S-Lang function `set_file_translation` with an integer argument. If the argument is 0, files are opened as text files; otherwise, they are opened in binary mode. There is no need to call this function for other systems. However, beware of the user variable `ADD_NEWLINE` which if non zero, a newline character will be appended to the file if the last character is not a newline character. If you are going to edit binary files, it is probably a good idea to set this variable to zero.

string that is entered will be inserted into the buffer and the process of defining the macro continues. Every time the macro is executed, JED will prompt for a NEW string to be inserted.

Any time an error is generated, the process of defining the macro is aborted as well as execution of the macro. This is very useful and may be exploited often. For example, suppose you want to trim excess whitespace from the end of ALL lines in a buffer. Let us also suppose that the number of lines in the buffer is less than 32000. Then consider the following keystrokes:

```
^X (        ; begin macro
^E          ; goto end of line
ESC \       ; trim whitespace
DOWN        ; go down one line
^X )        ; end macro
```

Now the macro has been defined. So move to the top of the buffer and execute it 32000 times:

```
ESC <          ; top of buffer
ESC 3 2 0 0 0   ; repeat next command 32000 times
^X e           ; execute macro
```

If the buffer has less than 32000 lines, the end of the buffer will be reached and an error will be generated aborting the execution of the macro.

0.13 Shells and Shell Commands

The default binding to execute a shell command and pump the output to a buffer is `ESC !`. JED will prompt for a command line and spawn a subprocess for its execution.

Strictly speaking, JED does not support interactive subprocesses. However, JED includes S-Lang code that "emulates" such a subprocess. It may invoked by typing `shell` at the `M-x` minibuffer prompt. A window will be created with a buffer named '`*shell*`' attached to it. Any text entered at the system dependent shell prompt will be executed in a subprocess and the result stuffed back in the shell buffer. Don't try to execute any commands which try to take over the keyboard or the screen or something undesirable may happen. Examples of types of stupid commands are spawning

keys {, }, and TAB are also special in this mode. The TAB key indents the current line and the { and } keys insert themselves and reindent. If you do not like any of these bindings, simply rebind the offending one to self_insert_cmd.

Finally, the key sequence ESC ; is bound to a function called c_make_comment. This function makes and and indents a C comment to the column specified by the value of the variable C_Comment_Column. If a comment is already present on the line, it is indented.

0.11.4 Fortran Mode

Fortran Mode is written entirely in S-Lang and is designed to facilitate the writing of Fortran programs. It features automatic indentation of Fortran code as well as automatic placement of Fortran statement Labels.

In this mode, the keys 0-9 are bound to a function for_elebel which does the following:

1. Inserts the calling character (0-9) into the buffer.
2. If the character is preceded by only other digit characters, it assumes the character is for a label and moves it to the appropriate position.
3. Reindents the line.

This function is very similar to the one Emacs uses for labels.

0.12 Keyboard Macros

JED is able to record a series of keystrokes from the terminal and replay them. The saved series of keystrokes is known as a keyboard macro. To begin a keyboard macro, simply enter the begin keyboard macro key sequence which is bound to ^X(if 'emacs.sl' is loaded. To stop recording the keystrokes, enter ^X). Then to "execute" the macro, press ^Xe. Please note that it is illegal to execute a macro while defining one and doing so generates an error. A macro can be aborted at anytime by pressing the ^G key.

One nice feature JED includes is the "macro_query" function. That is, while defining a macro, the key sequence ^X q will cause JED to issue the prompt 'Enter String:' in the minibuffer. Any

Finally, if either of these functions is called from the keyboard with a prefix argument, the paragraph will be justified as well. For example, pressing **ESC 1 ESC N** on the previous paragraph yields:

```
This paragraph is the result of using the
function ``narrow_paragraph''.  Note how the
right margin is less here than in the above
paragraph.
```

See the discussion of **format_paragraph_hook** in the section on hooks for details on how this is implemented.

0.11.2 Smart Quotes

You have probably noticed that many key words in this document are quoted in double quotes like "this is double quoted" and 'this is single quoted'. By default, the double quote key (") and single quote key (') are bound to the function **text_smart_quote**. With this binding and in wrap mode, the single quote key inserts a single quote with the "proper" orientation and the double quote key inserts two single quotes of the "proper" direction. To turn this off, rebind the keys to **self_insert_cmd**. Some modes already do this (e.g., EDT).

This brings up the question: if the double quote key is bound to **text_smart_quote** then how does one insert the character (")? The most common way is to use the **quoted_insert** function which, by default, is bound to the single backquote (`) key. This is the same mechanism that is used to insert control characters. The other method is to use the fact that if the preceding character is a backslash, \, the character simply self inserts. Again, this is ideal for writing TEX documents.

0.11.3 C Mode

C Mode facilitates the editing of C files. Much of the latter part of the development of the JED editor was done using this mode. This mode may be customized by a judicious choice of the variables C_INDENT and C_BRACE as well as the bindings of the curly brace keys **{** and **}**. Experiment to find what you like or write your own using the S-Lang interface.

By default, the **RET** key is bound to the function **newline_and_indent**. This does what its name suggests: inserts a newline and indents. Again, some modes may rebind this key. In addition, the

0.11 Modes

Jed supports two internal modes as well as user defined modes. The two internal modes consist of a "C" mode for C Language programming and a "Wrap" mode for ordinary text editing. Examples of user defined modes are Fortran mode and DCL mode.

Online documentation is provided for nearly every mode JED defines. For help on the current mode, press `ESC X` and enter `describe_mode`. A window will appear with a short description of the special features of the mode as well as a description of the variables affecting the mode.

0.11.1 Wrap Mode

In this mode, text is wrapped at the column given by the `WRAP` variable. The default is 78. The text does not wrap until the cursor goes beyond the wrap column and a space is inserted.

0.11.1.1 Formatting paragraphs

Paragraph delimiters are: blank lines, lines that begin with either a percent character, '%', or a backslash character '\'. This definition is ideally suited for editing (La)TEX documents. However, it is possible for the user to change this definition. See the the discussion of the hook, `is_paragraph_separator`, in the section on hooks for explicit details on how to do this.

The paragraph is formatted according to the indentation of the current line. If the current line is indented, the paragraph will be given the same indentation. The default binding for this function is `ESC q`.

In addition, a paragraph may be "narrowed" by the `narrow_paragraph` function which is bound to `ESC N` by default. This differs from the ordinary `format_paragraph` function described above in that the right margin is reduced by an amount equal to the indentation of the current line. For example:

```
This paragraph is the result of using the
function ``narrow_paragraph''.  Note how the
right margin is less here than in the above
paragraph.
```

0.10.6 Sorting

JED is capable of sorting a region of lines using the heapsort algorithm. The region is sorted alphabetically based upon the ASCII values of the characters located within a user defined rectangle in the region. That is, the rectangle simply defines the characters upon what the sort is based. Simply move to the top line of the region and set the mark on the the top left corner of the rectangle. Move to the bottom line and place the point at the position which defines the lower right corner of the rectangle. Press `ESC X` to get the `M-x` prompt and enter `sort` As as example, consider the following data:

```
Fruit:          Quantity:
lemons             3
pears             37
peaches          175
apples           200
oranges           56
```

To sort the data based upon the name, move the Point to the top left corner of the sorting rectangle. In this case, the Point should be moved to the '`l`' in the word '`lemons`'. Set the mark. Now move to the lower right corner of the rectangle which is immediately after the '`s`' in '`oranges`'. Pressing `ESC X` and entering `sort` yields:

```
Fruit:          Quantity:
apples           200
lemons             3
oranges           56
peaches          175
pears             37
```

Suppose that it is desired to sort by quantity instead. Looking at the original (unsorted) data, move the Point to two spaces before the '`3`' on the line containing '`lemons`'. The cursor should be right under the '`u`' in '`Quantity`'. Set the mark. Now move the Point to immediately after '`56`' on the '`oranges`' line and again press `ESC X` and enter `sort`. This yields the desired sort:

```
Fruit:          Quantity:
lemons             3
pears             37
oranges           56
peaches          175
apples           200
```

from the current point for the string '**ap**', etc... This way, one is able to quickly locate the desired string with only a minimal amount of information.

Unlike the "ordinary" search, the incremental search is not terminated with the **RET** (^M) key. Hitting the **RET** key causes JED to search for the next occurrence of the string based on the data currently entered at the prompt. The search is terminated with the **ESC** key.

Finally, the **DEL** key (^?) is used to erase the last character entered at the search prompt. In addition to erasing the last character of the search string, JED will return back to the location of the previous match. Erasing all characters will cause the editor to return to the place where the search began. Like many things, this is one of those that is easier to do than explain. Feel free to play around with it.

0.10.5 Rectangles

JED has built-in support for the editing of rectangular regions of text. One corner of rectangle is defined by setting the mark somewhere in the text. The Point (cursor location) defines the opposite corner of the rectangle.

Once a rectangle is defined, one may use the following functions:

kill_rect

Delete text inside the rectangle saving the rectangle in the internal rectangle buffer.

open_rect

Push all text in the rectangle to the right outside the rectangle.

copy_rect

Copy text inside the rectangle to the internal rectangle buffer.

blank_rect

Replace all text inside the rectangle by spaces.

The function **insert_rect** inserts a previously killed or copied rectangle into the text at the Point.

These functions have no default binding and must be entered into the MiniBuffer by pressing **ESC X** to produce the **M-x** prompt.

used. When moving from the part of the document where the current tab setting is appropriate to the part where the other tab setting was used will cause the document to look unformatted unless the appropriate tab settings are restored. Wordprocessors store the tab settings in the file with the text so that the tabs may be dynamically changed to eliminate such unwanted behavior. However, text editors such as JED, vi, Emacs, EDT, EVE (TPU), etc, do not store this information in the file. JED avoids this problem by using simulated tabs. When using simulated tabs, tabs are not really used at all. Rather JED inserts the appropriate number of spaces to achieve the desired effect. This also has the advantage of one being able to cut and paste from the part of a document using one tab setting to another part with a different tab setting. This simple operation may lead to unwanted results on some wordprocessors as well as those text editors using real tab stops.

0.10.4 Searching

JED currently has two kinds of searches: ordinary searches and incremental searches. Both types of searches have forward and backward versions. The actual functions for binding purposes are:

```
search_forward
search_backward
isearch_forward
isearch_backward
```

There is also the `occur` function which finds all occurrences of a single word (string). This function has no backwards version. By default it is not bound to any keys, so to use it, `occur` must be entered at the `M-x` prompt (`ESC X`) or one is always free to bind it to a key.

In the following only the incremental search is discussed.

The default type of search in Emacs in the incremental search. However, since this type of search is confusing to the uninitiated, the ordinary type of search has been made the default in JED's Emacs emulation. For the traditional emacs keybinding, it is up to the user to provide the keybinding.

As the name suggests, an incremental search performs a search incrementally. That is, as you enter the search string, the editor begins searching right away. For example, suppose you wish to search for the string '`apple`'. As soon as the letter `a` is entered into the incremental search prompt, JED will search for the first occurrence of '`a`'. Then as soon as the `p` is entered, JED will search

This indicates that a region is being defined. Moving the cursor (`Point`) defines the other end of a region. Many editors will highlight the region as it is marked. I have not added this to JED since such a feature complicates the display code without contributing any useful functionality. Of course this is only my opinion and is open for debate.

It is easy to see where the location of the mark is by using the `exchange` command which is bound to ^X ^X. This simply exchanges the `Point` and the `Mark`. The region is still intact since it is defined only by the `Point` and `Mark`. Pressing ^X ^X again restores the mark and Point back to their original locations. Try it.

0.10.3 Tab Issues.

Strictly speaking, JED uses only fixed column tabs whose size is determined by the value of the `TAB` variable. Setting the `TAB` variable to 0 causes JED to not use tabs as whitespace and to display tabs as ^I. Please note that changing the tab settings on the terminal will have no effect as far as JED is concerned. The `TAB` variable is local to each buffer allowing every buffer to have its own tab setting. The variable `TAB_DEFAULT` is the tab setting that is given to all newly created buffers. The default value for this variable is 8 which corresponds to eight column tabs.

JED is also able to "simulate" arbitrary tabs as well through the use of user defined tab stops. Calling the function `edit_tab_stops` allows the user to interactively set the tab stops. That is, one simply presses `ESC X` to get the `M-x` prompt and enters `edit_tab_stops`. A window will pop open displaying the current tab settings. To add a tab stop, simply place a `T` in the appropriate column. Use the space bar to remove a tab stop.

Here an argument is presented in favor of simulated tabs over real tab stops. First, consider what a "tab" really is. A "tab" in a file is nothing more than a character whose ASCII value is 9. For this reason, one also denotes a tab as ^I (Control-I). Unlike most other ASCII characters, the effect of the tab character is device dependent and is controlled through the device tab settings. Hence, a file which displays one way on one device may look totally different on another device if the tab settings do not correspond. For this reason, many people avoid tabs altogether and others the adopt "standard" of eight column tabs. Even though people always argue about what the correct tab settings should be, it must be kept in mind that this is primarily a human issue and not a machine issue.

On a device employing tab stops, a tab will cause the cursor to jump to the position of the next tab stop. Now consider the effect of changing the tab settings. Assume that in one part of a document, text was entered using the first setting and in another part, the second setting was

`ESC x`	Get M-x minibuffer prompt with command completion
`^X ^B`	pop up a list of buffers
`^X ^C`	exit JED
`^X 0`	Delete Current Window
`^X 1`	One Window
`^X 2`	Split Window
`^X o`	Other window
`^X b`	switch to buffer
`^X k`	kill buffer
`^X s`	save some buffers
`ESC .`	Find tag (unix ctags compatible)
`^@`	Set Mark (Begin defining a region)

0.10.1 Undo

One of JED's nicest features is the ability to undo nearly any change that occurs within a buffer at the touch of a key. If you delete a word, you can undo it. If you delete 10 words in the middle of the buffer, move to the top of the buffer and randomly make changes, you can undo all of that too.

By default, the `undo` function is bound to the key `^_` (Ascii 31). Since some terminals are not capable of generating this character, it is also bound to the key sequence `^X u`.

Due to the lack of virtual memory support on IBMPC systems, the `undo` function is not enabled on every buffer. In particular, it is not enabled for the `*scratch*` buffer. However, it is enabled for any buffer which is associated with a file. A "plus" character on the left hand side of the status line indicates that undo is enabled for the buffer. It is possible to enable undo for any buffer by using the `toggle_undo` function.

0.10.2 Marking Text (Point and Mark)

Many commands work on certain regions of text. A region is defined by the `Point` and the `Mark` The `Point` is the location of the current editing point or cursor position. The `Mark` is the location of a mark. The mark is set using the `set_mark_cmd` which is bound to `^@` (Control-2 or Control-Space on some keyboards). When the mark is set, the '`m`' mark indicator will appear on the status line.

0.9.3 Buffer Name and File Name Completion

When JED prompts for a file name or a buffer name, the space bar and the `TAB` keys are special. Hitting the `TAB` key will complete the name that is currently in the minibuffer up until it is no longer unique. At that point, you can either enter more characters to complete the name or hit the space bar to cycle among the possible completions. The spacebar must be pressed at least twice to cycle among the completions.

On MSDOS and VMS, it is possible to use wildcard characters in the file name for completion purposes. For example, entering `*.c` and hitting the space bar will cycle among file names matching `*.c`. Unfortunately, this feature is not available on unix systems.

0.10 Basic Editing

Editing with JED is pretty easy— most keys simply insert themselves. Movement around the buffer is usually done using the arrow keys or page up and page down keys. If '`edt.sl`' is loaded, the keypads on VTxxx terminals function as well. Here, only the highlights are touched upon (cut/paste operations are not considered "highlights"). In the following, any character prefixed by the ^ character denotes a Control character. On keyboards without an explicit Escape key, Control-[will most likely generate and Escape character. The following list of useful keybindings assumes that '`emacs.sl`' has been loaded.

`^L`	redraw screen
`^_`	Undo (Control-underscore, also ^X u)
`ESC q`	reformat paragraph (wrap mode)
`ESC n`	narrow paragraph (wrap mode)
`ESC ;`	Make Language comment (Fortran and C)
`ESC \`	Trim whitespace around point
`ESC !`	Execute shell command
`ESC $`	Ispell word (unix)
`^X ?`	Show line/column information
` ` `	`quoted_insert` — insert next char as is (backquote key)
`ESC s`	Center line
`ESC u`	Upcase word
`ESC d`	Downcase word
`ESC c`	Capitalize word

```
Find File: C:\JED\src\video.c
Find File: C:\JED\SLANG\..\src\video.c
Find File: C:\JED\SLANG\../src/video.c
```

Note that the on MSDOS systems, JED replaces the '/' with a '\' and that case is not important. Now suppose you wish to get the file VIDEO.C from disk A:. The following are also valid:

```
Find File: A:\video.c
Find File: A:video.c
Find File: C:\JED\SLANG\a:\video.c
```

In the last case, JED is smart enough to figure out what is really meant. Although the above examples are for MSDOS systems, the rules also apply to unix and VMS systems as well. The only change is the file name syntax. For example, on VMS

```
sys$manager:[misc]dev$user:[davis.jed]vms.c
dev$user:[davis.jed]vms.c
```

become equivalent filenames upon expansion. For unix, the following are equivalent:

```
/user1/users/davis/jed/unix.c
/usr/local/src//user1/users/davis/jed/unix.c
/usr/local/src/~/jed/unix.c
```

Note the last example: the tilde character '~' always expands into the users **HOME** directory, in this case to '**/user1/users/davis**'.

When JED writes a buffer out to a file, it usually prompts for a file name in the minibuffer displaying the directory associated with the current buffer. At this point a name can be appended to the directory string to form a valid file name or the user may simply hit the **RET** key. If the latter alternative is chosen, JED simply writes the buffer to the file already associated with the buffer. Once the buffer is written to a file, the buffer becomes attached to that file.

simply unreasonable to require the user to remember if a function is bound to a key or not and, if it is, to remember the key to which it is bound. This is especially true of those functions that are bound but rarely used. More often than not, one simply forgets the exact name or spelling of a function and requires a little help. For this reason, JED supports command line completion in the minibuffer. This function, called **emacs_escape_x**, is bound to the key **ESC X**. This is one binding that must be remembered!

As an example, suppose that you are editing several buffers and you wish to insert the contents of one buffer into the the current buffer. The function that does this is called **insert_buffer** and has no default keybinding. Pressing **ESC X** produces the prompt '**M-x**'. This prompt, borrowed from the Emacs editor, simply means that **ESC X** was pressed. Now type '**in**' and hit the space bar or the **TAB** key. In this context (completion context) the space bar and the **TAB** will expand the string in the MiniBuffer up until it is nolonger unique. In this case, '**insert_file**' and '**insert_buffer**' are only two functions that start with '**in**'. Hence, '**in**' will expand to '**insert_**' at which point it becomes necessary to enter more information to uniquely specify the desired function. However, in a completion context, the space bar also has a special property that enables the user to cycle among the possible completions. For this example, hitting the space bar twice consecutively will produce the string '**insert_file**' and hitting it again produces the desired string '**insert_buffer**'.

The role of the space bar in completion is a point where Emacs and JED differ. Emacs will pop up a buffer of possible completions but JED expects the user to press the space bar to cycle among them. Both have there pros and cons. Frequently, one sees messages on the Usenet newsgroup **gnu.emacs.help** from Emacs users asking for the kind of completion JED employs.

0.9.2 File Names

JED takes every file name and "expands it" according to a set of rules which vary according to the Operating System. For concreteness, consider JED running under MSDOS. Suppose the user reads a new file into the editor via the **find_file** command which emacs binds to ^X ^F. Then the following might be displayed in the minibuffer:

```
Find File: C:\JED\SLANG\
```

Here JED is prompting for a file name in the directory \JED\SLANG on disk C:. However, suppose the user wants to get the file C:\JED\SRC\VIDEO.C. Then the following responses produce equivalent filenames when JED expands them internally:

0.8 Status line and Windows

JED supports multiple windows. Each window may contain the same buffer or different buffers. A status line is displayed immediately below each window. The status line contains information such as the JED version number, the buffer name, "mode", etc. Please beware of the following indicators:

'**'	buffer has been modified since last save
'%%'	buffer is read only
'm'	mark set indicator. This means a region is being defined
'd'	file changed on disk indicator. This indicates that the file associated with the buffer is newer than the buffer itself.
's'	spot pushed indicator
'+'	undo is enabled for the buffer
'[Macro]'	a macro is being defined.
'[Narrow]'	buffer is narrowed to a region of LINES.

0.9 MiniBuffer

The MiniBuffer consists of a single line located at the bottom of the screen. Much of the dialog between the user and JED takes place in this buffer. For example, when you search for a string, JED will prompt you for the string in the MiniBuffer.

The MiniBuffer also provides a direct link to the S-Lang interpreter. To access the interpreter, press ^X ESC and the 'S-Lang>' prompt will appear in the MiniBuffer. Enter any valid S-Lang expression for evaluation by the interpreter.

It is possible to to recall data previously entered into the MiniBuffer by using the up and down arrow keys. This makes it possible to use and edit previous expressions in a convenient and efficient manner.

0.9.1 Command Line Completion

The JED editor has several hundred built–in functions as well as many more written in the S-Lang extension language. Many of these functions are bound to keys and many are not. It is

to your '`jed.rc`' ('`.jedrc`') startup file will enable JED's Wordstar emulation.

0.6 File Types and Sizes

JED is primarily a text editor; however, it can also edit binary files (see the section on editing binary files). As a result, JED may edit lines of arbitrary length (actually this depends upon the size of an integer). It is capable of editing arbitrarily large buffers as long as there is enough memory for the buffer as well as the overhead involved. This editor employs a linked list representation; hence, the overhead can be quite high.

0.7 Backup and Autosave Files

On UNIX and MSDOS systems, JED creates backup files by appending a '`~`' character to the filename. The VMS operating system handles backup files itself. JED periodically autosaves its buffers. On UNIX and MSDOS, autosave files are prefixed with the pound sign '`#`'. On VMS, they are prefixed with '`_$`'. The autosave interval may be changed by setting the variable MAX_HITS to the desired value. The default is 300 "hits" on the buffer. A "hit" is defined as a key which MAY change the state of the buffer. Cursor movement keys do not cause hits on the buffer.

Like many of JED's features, the names of autosave and backup files can be controlled by the user. The file '`site.sl`' defines two functions, `make_backup_filename`, and `make_autosave_filename` that generate the file names described in the previous paragraph. Like all S–Lang functions, these functions may be overloaded and replaced with different ones. See also information about `find_file_hook` in the section on hooks.

On UNIX systems, JED catches most signals and tries to autosave its buffers in the event of a crash or if the user accidently disconnects from the system ('`SIGHUP`').

If an autosave file exists and you is desire to recover data from the autosave file, use the function `recover_file`. Whenever JED finds a file, it checks to see if an autosave file exists as well as the file's date. If the dates are such that the autosave file is more recent JED will display a message in the minibuffer alerting the user of this fact and that the function `recover_file` should be considered.

is in present in the '`jed.rc`' ('`.jedrc`') Startup File. JED is distributed with EDT emulation enabled on VMS and Unix systems but the above line is commented out in the '`jed.rc`' file on MSDOS systems.

This emulation provides a near identical emulation of the EDT keypad key commands. In addition, the smaller keypad on the newer DEC terminals is also setup. It is possible to have both EDT and Emacs emulation at the same time. The only restriction is that '`emacs.sl`' must be loaded before '`edt.sl`' is loaded.

One minor difference between JED's EDT emulation and the real EDT concerns the ^H key. EDT normally binds this to move the cursor to the beginning of the line. However, JED uses it as a help key. Nevertheless, it is possible to rebind it. See the section on rebinding keys as well as the file '`edt.sl`' for hints. Alternatively, simply put

```
"^H" unsetkey "bol" "^H" setkey
```

in the '`jed.rc`' startup file after '`edt.sl`' is loaded. Keep in mind that the ^H key will no longer function as a help key if this is done.

EDT emulation for PCs only work with the enhanced keyboard. When '`edt.sl`' is loaded, a variable `NUMLOCK_IS_GOLD` is set which instructs JED to interpret the numlock key on the square numeric keypad to function as the EDT GOLD key. In fact, this keypad should behave exactly like the keypad on VTxxx terminals. The only other problem that remains concerns the `+` key on the PC keypad. This key occupies two VTxxx key positions, the minus and the comma (delete word and character) keys. Thus a decision had to be made about which key to emulate. I chose the `+` key to return the characters `^[ O l` which JED maps to the delete character function. This may be changed to the delete word function if you prefer. See the file '`edt.sl`' for details.

The `GOLD`–`GOLD` key combination toggles the keypad between application and numeric states. On the PC, this is not possible. Instead, the PC `F1` key has been instructed to perform this task.

0.5.3 Wordstar Emulation

'`wordstar.sl`' contains the S-Lang code for JED's Wordstar emulation. Adding the line

```
"wordstar.sl"     evalfile   pop
```

Now suppose that you want to read in a file with the name of one of the switches, say `-2`. How can this be done? The answer depends upon the operating system. For unix, instead of '`jed -2`', use '`jed ./-2`'; for VMS, use '`jed []-2`'. The case for MSDOS is similar to unix except that one must use the backslash.

It is possible to completely change JED's command line syntax through the use of the user defined function `command_line_hook`. In fact, the behavior described above is dictated by the value of `command_line_hook` as distributed in '`site.sl`'. See the section on hooks for details.

0.5 Emulating Other Editors

JED's ability to create new functions using the S–Lang programming language as well as allowing the user to choose key bindings, makes the emulation of other editors possible. Currently, JED provides reasonable emulation of the Emacs, EDT, and Wordstar editors.

0.5.1 Emacs Emulation

Emacs Emulation is provided by the S-Lang code in '`emacs.sl`'. The basic functionality of Emacs is emulated; most Emacs users should have no problem with JED. To enable Emacs emulation in JED, make sure that the line

```
"emacs.sl"  evalfile pop
```

is in your '`jed.rc`' ('`.jedrc`') startup file. JED is distributed with this line already present in the default '`jed.rc`' file.

0.5.2 EDT Emulation

For EDT emulation, '`edt.sl`' must be loaded. This is accomplished by ensuring that the line

```
"edt.sl"     evalfile pop
```

or

`jed` *file_list*.

However, JED also takes the following switches:

`-batch`	run JED in batch mode. This is a non–interactive mode.
`-n`	do not load '`jed.rc`' ('`.jedrc`') file.
`-g` *n*	goto line *n* in buffer
`-l` *file*	load *file* as `S-Lang` code.
`-f` *function*	execute S-Lang function named *function*
`-s` *string*	search forward for *string*
`-2`	split window
`-i` *file*	insert *file* into current buffer.

For example, the command line:

```
jed slang.c -g 1012 -s error -2 file.c -f eob
```

will start up JED, read in the file '`slang.c`', goto line 1012 of '`slang.c`' and start searching for the string '`error`', split the window, read in '`file.c`' and goto the end of the file.

If the '`-batch`' parameter is used, it must be the first parameter. Similarly, if '`-n`' is used, it must also be the first parameter unless used with the '`-batch`' parameter in which case it must the second. JED should only be run in batch mode when non–interactive operation is desired. For example, JED is distributed with a file, '`mkdoc.sl`', that contains S–Lang code to produce a help file for functions and variables. In fact, the help file '`jed_funs.hlp`' was created by entering

```
jed -batch -n -l mkdoc.sl
```

at the command line.

directory. The environment variable `JED_LIBRARY` is a comma separated list of directories. Here are some examples of setting this variable for different systems:

```
  VMS:  define/job JED_LIBRARY  dev$lib:[jedfiles]
 UNIX:  setenv JED_LIBRARY '/usr/local/lib/jed,~/jed'
IBMPC:   set JED_LIBRARY = c:\editors\jed\lib
```

You will probably want to put define `JED_LIBRARY` in your login startup file, e.g., '`autoexec.bat`', '`login.com`', or '`.cshrc`'.

JED versions 0.92 and later allow the value of `JED_LIBRARY` to be specified at compile time an it may only be necessary to define `JED_LIBRARY` as an environment variable to override its pre–compiled value.

0.3 Startup Files

Once JED has loaded the startup file '`site.sl`', it will try to load the user's personal initialization file. It first looks in the directory pointed to by the environment variable `JED_HOME`. If that fails, it then searches the `HOME` directory and upon failure simply loads the one supplied in `JED_LIBRARY`.

The name of the user initialization file varies according to the operating system. On Unix systems this file must be called '`.jedrc`' while on VMS and MSDOS, it goes by the name '`jed.rc`'. For VMS systems, the `HOME` directory corresponds to the `SYS$LOGIN` logical name while for the other two systems, it corresponds to the `HOME` environment variable.

The purpose of this file is to allow an individual user to taylor JED to his or her personal taste. Most likely, this will involve choosing an initial set of keybindings, setting some variables, and so on.

0.4 Starting JED

Normally, JED is started as

`jed` *filename*

0.1 Introduction

This document presents some basic information that users should know in order to use JED effectively. Any questions, comments, or bug reports, should be emailed to the author. Please be sure to include the version number. To be notified of future releases of JED, email to the address below and your email address will be placed on the announcement list. The email address is:

davis@amy.tch.harvard.edu (internet)
davis@ohstpy (BITNET)

0.2 Installing JED

Building JED from its sources requires the use of a C compiler that understands function prototypes. JED has been successfully built with `cc` on the ULTRIX, VMS, and IRIX operating systems. In addition, it has been created using `gcc` under SunOS and Borland's BCC 3.0 for the PC.

Detailed installation instructions are in separate, operating system dependent files. They are:

UNIX:	install.unx
VMS:	install.vms
IBMPC:	install.pc

When JED starts up, it will first try to load a site initialization file called '`site.sl`'. Site specific commands are placed here. Most likely, '`site.sl`' will define some functions, default hooks, etc... What goes in it is left to the discretion of the user or system manager. See the file `site.sl` for examples.

> If you intend to use characters with ASCII codes greater than 127, it is important to read the section on "Eight Bit Clean Issues" in order to install JED in an 8 bit clean manner.

When loading '`site.sl`' as well as other '`S-Lang`' files (the user's personal initialization file, '`.jedrc`' or '`jed.rc`', is a special case, see below), JED searches all directories specified by the environment variable `JED_LIBRARY`, and if the file is not found, JED will look for it in the default

JED

A Programmer's Editor

Version 0.92

July 1993

John E. Davis

Copyright © 1992, 1993 John E. Davis

Permission to use, copy, modify, and distribute this software and its documentation for any non-commercial purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

This program is provided AS IS with no warranties of any kind. The author shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by this program or any part thereof. In no event will the author be liable for any lost revenue or profits or other special, indirect and consequential damages.