The jed Editor

Contents

1 Introduction **3**

2 Starting JED **3**

3 Emulating Other Editors **4**

3.1 Emacs Emulation 4

3.2 EDT Emulation 4

3.3 Wordstar Emulation 4

4 Backup and Auto-save Files **4**

5 Status line and Windows **5**

6 Mini-Buffer **5**

6.1 Command Line Completion 5

6.2 File Names 6

6.3 Buffer Name and File Name Completion 6

7 Basic Editing **7**

7.1 Undo 8

7.2 Marking Text (Point and Mark) 8

7.3 Tab Issues. 8

7.4 Searching 9

7.5 Rectangles 9

7.6 Sorting 9

8 Modes **10**

8.1 Wrap Mode . . . 10

Formatting paragraphs . . . 10

8.2 Smart Quotes . . . 11

8.3 C Mode . . . 11

8.4 Fortran Mode . . . 11

9 Keyboard Macros **11**

10 Shells and Shell Commands **12**

11 Getting Help **12**

12 Editing Binary Files **12**

13 Dired— the Directory editor **13**

14 Customization **13**

14.1 Setting Keys . . . 13

14.2 Predefined Variables . . . 14

15 Eight Bit Clean Issues **14**

15.1 Displaying Characters with the High Bit Set . . . 14

15.2 Inputting Characters with the hight bit Set . . . 15

15.3 Upper Case - Lower Case Conversions . . . 15

16 Miscellaneous **16**

16.1 Abort Character . . . 16

16.2 Input Translation . . . 16

16.3 Display Sizes . . . 16

17 xjed **16**

17.1 Resources . . . 17

17.2 Mouse Usage . . . 18

17.3 EDT emulation under Linux . . . 18

18 Using a mouse with jed and xjed **18**

18.1 Native Mouse Support . . . 19

Clicking in a window . . . 19

Clicking on a window status line . . . 19

Tips for using the mouse . . . 19

18.2 XTerm Event Support . . . 20

Mouse Usage . . . 20

Cut/Paste Tips . . . 20

19 Frequently Asked Questions **20**

How do I obtain **jed**? . . . 20

How do I disable **jed**'s C mode? . . . 21

What is C mode? . . . 21

How do I turn on wrap mode or turn it off? . . . 22

What is the difference between internal and intrinsic functions? . . . 22

Sometimes during screen updates, **jed** pauses. Why is this? . . . 22

How do I get **jed** to recognize Control-S and Control-Q? . . . 23

Can I bind the `Alt` keys on the PC? . . . 23

How do I find out what characters a particular key generates? . . . 23

jed scrolls slow on my WizBang-X-Super-Terminal. What can I do about it? . . . 24

How do I get a list of functions? . . . 24

How can I use `edt.sl` with `jed386.exe`? . . . 24

How do I set custom tab stops in **jed**? . . . 24

Questo documento è un'elaborazione della documentazione originale di **jed**, un editor emacs-like disponibile su piattaforme UNIX, MS-DOS e VMS (esiste anche **xjed**, una versione per X11).

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 email-ed 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@space.mit.edu
```

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.

2 Starting JED

Normally, **jed** is started as

```
jed <file name>
```

However, **jed** also takes the switches defined in the following table:

`-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.

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 MS-DOS is similar to Unix except that one must use the backslash.

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 tailor **jed** to his or her personal taste. Most likely, this will involve choosing an initial set of key-bindings, setting some variables, and so on.

3 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.

3.1 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

```
evalfile ("emacs"); pop ():
```

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

3.2 EDT Emulation

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

```
evalfile ("edt"); pop ();
```

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 MS-DOS 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 `Ctrl-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 re-bind it. See the section on re-binding keys as well as the file `edt.sl` for hints. Alternatively, simply put

```
unsetkey ("^H"); setkey ("bol", "^H");
```

in the `jed.rc` startup file after `edt.sl` is loaded. Keep in mind that the `Ctrl-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 Num-Lock 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 `Esc 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.

3.3 Wordstar Emulation

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

```
evalfile ("wordstar"); pop ();
```

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

4 Backup and Auto-save Files

On UNIX and MS-DOS systems, **jed** creates backup files by appending a `~` character to the filename. The VMS operating system handles backup files itself. **jed** periodically auto-saves its buffers. On UNIX and MS-DOS, auto-save files are prefixed with the pound sign #. On VMS, they are prefixed with `\_$`. The auto-save 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 auto-save 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 auto-save its buffers in the event of a crash or if the user accidently disconnects from the system (`SIGHUP`).

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

5 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

6 Mini-Buffer

The Mini-Buffer 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 Mini-Buffer.

The Mini-Buffer also provides a direct link to the **S-Lang** interpreter. To access the interpreter, press `Ctrl-X Esc` and the `S-Lang>` prompt will appear in the Mini-Buffer. Enter any valid **S-Lang** expression for evaluation by the interpreter.

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

6.1 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 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 mini-buffer. 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 current buffer. The function that does this is called `insert_buffer` and has no default key-binding. 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 Mini-Buffer up until it is no longer unique. In this case, `insert_file` and `insert_buffer` are only the 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.

6.2 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 MS-DOS. Suppose the user reads a new file into the editor via the `find_file` command which emacs binds to `Ctrl-X Ctrl-F`. Then the following might be displayed in the mini-buffer:

```
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:

```
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 MS-DOS 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 MS-DOS 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.

6.3 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.

7 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, `Ctrl-[` will most likely generate and Escape character.

A "prefix argument" to a command may be generated by first hitting the `Esc` key, then entering the number followed by pressing the desired key. Normally, the prefix argument is used simply for repetition. For example, to move to the right 40 characters, one would press `Esc 4 0` followed immediately by the right arrow key. This illustrates the use of the repeat argument for repetition. However, the prefix argument may be used in other ways as well. For example, to begin defining a region, one would press the `Ctrl-@` key. This sets the mark and begins highlighting. Pressing the `Ctrl-@` key with a prefix argument will abort the act of defining the region and to pop the mark.

The following list of useful keybindings assumes that `emacs.sl` has been loaded.

`Ctrl-L`	Redraw screen
`Ctrl-_`	Undo (Control-underscore, also `Ctrl-X u`)
`Esc q`	Reformat paragraph (wrap mode). Used with a prefix argument. will justify the paragraph as well.
`Esc n`	narrow paragraph (wrap mode). Used with a prefix argument will justify the paragraph as well.
`Esc ;`	Make Language comment (Fortran and C)
`Esc \`	Trim whitespace around point
`Esc !`	Execute shell command
`Esc $`	Ispell word (unix)
`Ctrl-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
`Esc x`	Get `M-x` minibuffer prompt with command completion
`Ctrl-X Ctrl-B`	pop up a list of buffers
`Ctrl-X Ctrl-C`	exit **jed**
`Ctrl-X 0`	Delete Current Window
`Ctrl-X 1`	One Window
`Ctrl-X 2`	Split Window
`Ctrl-X o`	Other window
`Ctrl-X b`	switch to buffer
`Ctrl-X k`	kill buffer
`Ctrl-X s`	save some buffers
`Ctrl-X Esc`	Get `S-Lang>` prompt for interface to the **S-Lang** interpreter
`Ctrl-Z`	Sospende **jed** (vedi nota sotto)
`Esc .`	Find tag (unix ctags compatible)
`Ctrl-@`	Set Mark (Begin defining a region). Used with a prefix argument aborts the act of defining the region and pops the Mark

A differenza di XEmacs, `Ctrl-E` non termina la sessione di editing, in modo da consentire di accedere agevolmente alla funzione di end-of-line anche a chi non ha a disposizione un keypad come nelle tastiere Digital. Chi volesse uniformarsi alla definizione di XEmacs, dovrà aggiungere questa riga nel proprio `.jedrc`:

```
setkey("exit_jed", "^E");
```

Sotto VMS il comando `Ctrl-Z` permette di sospendere temporaneamente l'esecuzione di **jed**. Alla successiva esecuzione del comando `jed` viene ripristinata la situazione precedente (in modo analogo a quanto avviene sotto unix, dove però bisogna usare il comando `fg`).

7.1 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 `Ctrl-_` (Ascii 31). Since some terminals are not capable of generating this character, it is also bound to the key sequence `Ctrl-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 `+` 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.

7.2 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 `Ctrl-@` (`Control-2` or `Control-Space` on some keyboards). When the mark is set, the `m` mark indicator will appear on the status line. This indicates that a region is being defined. Moving the cursor (`Point`) defines the other end of a region. If the variable `HIGHLIGHT` is non-zero, **jed** will highlight the region as it is defined.

Even without highlighting, it is easy to see where the location of the mark is by using the `exchange` command which is bound to `Ctrl-X Ctrl-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 `Ctrl-X Ctrl-X` again restores the mark and Point back to their original locations. Try it.

7.3 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 `Ctrl-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. 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` (`Ctrl-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 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.

7.4 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:

```
isearch_forward    Ctrl-F
isearch_backward   Ctrl-B
```

> Dato che **jed** può venire utilizzato su teminali non grafici che utilizzano il protocollo XON/XOFF, `Ctrl-S` e `Ctrl-R` non hanno il binding tradizionale di `search_forward` e `search_backward`. Chi volesse ripristinare questi binding deve aggiungere queste righe nel proprio `.jedrc`
>
> ```
> setkey("search_forward", "^S");
> setkey("search_backward", "^R");
> ```

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.

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 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.

The search is terminated with the `Enter` key.

Finally, the `DEL` key (`Ctrl-?`) 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.

7.5 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.
`n_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.

7.6 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 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
```

8 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.

8.1 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.

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 LaTeX documents. However, it is possible for the user to change this definition. See 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.
```

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.

8.2 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.

8.3 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 `Enter` 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 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 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.

8.4 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.

9 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 `Ctrl-X (` if `emacs.sl` is loaded. To stop recording the keystrokes, enter `Ctrl-X )`. Then to “execute” the macro, press `Ctrl-X e`. 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 `Ctrl-G` key.

One nice feature **jed** includes is the `macro_query` function. That is, while defining a macro, the key sequence `Ctrl-X q` will cause **jed** to issue the prompt `Enter String:` in the minibuffer. Any 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:

```
Ctrl-X (                    (begin macro)
Ctrl-E                      (goto end of line)
ESC                         (trim whitespace)
Down Arrow                  (go down one line)
Ctrl-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
Ctrl-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.

10 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 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 ..`.

11 Getting Help

jed's help functions are bound to `Ctrl-H` by default. For example, `Ctrl-H C` will show what function a key carries out, `Ctrl-H i` will run **jed**'s info reader, `Ctrl-H f` will give help on a particular **S-Lang** function, etc. However, some modes may use the `Ctrl-H` key for something else. For example, if EDT mode is in effect, then `Ctrl-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. `Ctrl-H i` will load `info_mode` allowing the user to browse the document as well as other "info" documents.

12 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.

13 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.

14 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 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.

14.1 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 `Ctrl-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:

```
setkey ("bol", "^A");
```

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`.

The first argument to the `setkey` function may be *any* **S-Lang**expression. Well, almost any. The only restriction is that the newline character cannot appear in the expression. For example, the line

```
setkey ("bol();skip_white ();",  "^A");
```

defines the `Ctrl-A` key such that when it is pressed, the editing point will move the beginning of the line and then skip whitespace up to the first non-whitespace character on the line.

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:

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

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 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 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 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: `Ctrl-A` or decimal `"\001"`.
2. Insert `/*`.
3. Goto end of the line: `Ctrl-E` or decimal `\005`.
4. Insert `*/`

To bind this sequence of steps to the key sequence `Esc ;`, simply use

```
setkey("@\001/*\005*/", "\e;");
```

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 `Ctrl-A` and `Ctrl-E` have been written. Do not attempt to use the `^` to represent "`Ctrl`". 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.

14.2 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.

15 Eight Bit Clean Issues

15.1 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:

```
DISPLAY_EIGHT_BIT = 1;
```

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 `DISPLAY_EIGHT_BIT = 160;`. I believe that ISO Latin character sets assume this. This is the default value for Unix and VMS systems.

15.2 Inputting Characters with the hight bit Set

Inputting characters with the high bit set into **jed** is another issue. How **jed** interprets this bit is controlled by the variable `META_CHAR`. 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 it *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 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 ‘`.

15.3 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:

- Each character is only 8 bits.
- Each character has a unique uppercase equivalent.
- Each character has a unique lowercase equivalent.

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

- 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 case conversions, it may be necessary to put a statement of the form:

```
define_case (X, Y);
```

in the startup file. For example, suppose 211 is the uppercase of 244. Then, the line

```
define_case (211, 244);
```

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 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

```
evalfile ("dos850"); 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".

16 Miscellaneous

16.1 Abort Character

The abort character (`Ctrl-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

```
set_abort_char (30);
```

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

16.2 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 `Ctrl-S` character. This character is especially notorious because many systems use it and `Ctrl-Q` for flow control. Nevertheless Emacs uses `Ctrl-S` for searching. Short of rebinding all keys which involve a `Ctrl-S` how does one work with functions that are bound to key sequences using `Ctrl-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 `Ctrl-\` for `Ctrl-S` everywhere. The line

```
map_input (28, 19);
```

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

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

```
map_input (8, 127);
```

will map the `Ctrl-H` (ascii 8) to the delete character (ascii 127).

16.3 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**.

17 xjed

These are some notes about using **xjed**, the X Window version of **jed**. It also mentions information about how to setup the EDT emulation under Linux.

Suspending **xjed** is not allowed. If `^Z` is pressed, the window is raised if it is obscured, or lowered if it is totally visible.

17.1 Resources

xjed recognizes the following resources:

`Display`	`[d]`	Display to run on
`Name`		Instance name
`Geometry`		Initial geometry specifications
`font`		Default font to use
`background`	`[bg]`	Background color
`foreground`	`[fg]`	Forground color
`Title`		name to be displayed on the title bar
`fgStatus`	`[fgs]`	foreground color of the xjed buffer status line
`bgStatus`	`[bgs]`	background color of the xjed buffer status line
`fgRegion`	`[fgr]`	foreground color of a region as defined by point and mark
`bgRegion`	`[bgr]`	background color of a region as defined by point and mark
`fgCursor`	`[fgc]`	text cursor foreground color
`bgCursor`	`[bgc]`	text cursor background color
`fgMouse`	`[fgm]`	mouse cursor foreground color
`bgMouse`	`[bgm]`	mouse cursor background color
`fgMessage`	`[fgms]`	Foreground color for messages
`bgMessage`	`[bgms]`	Background color for messages
`fgError`	`[fger]`	Foreground color for error messages
`bgError`	`[bger]`	Background color for messages

These resources specify color syntax highlighting options:

`fgOperator`	`[fgop]`	forground color for operators (+, -, etc...)
`bgOperator`	`[bgop]`	background color for operators
`fgNumber`	`[fgnm]`	foreground color for numbers
`bgNumber`	`[bgnm]`	background color for numbers
`fgString`	`[fgst]`	foreground color for strings
`bgString`	`[bgst]`	background color for strings
`fgComments`	`[fgco]`	forground color for comments
`bgComments`	`[bgco]`	background color for comments
`fgKeyword`	`[fgkw]`	foreground color for keywords
`bgKeyword`	`[bgkw]`	background color for keywords
`fgKeyword1`	`[fgkw1]`	foreground color for keywords1
`bgKeyword1`	`[bgkw1]`	background color for keywords1
`fgDelimiter`	`[fgde]`	foreground color for delimeters
`bgDelimiter`	`[bgde]`	background color for delimeters
`fgPreprocess`	`[fgpr]`	foreground color for preprocessor lines
`bgPreprocess`	`[bgpr]`	background color for preprocessor lines

Any of the above items may be specified on the **xjed** command line. Quantities enclosed in square brackets may be used as a shortform of their longer counterparts.

For example,

```
xjed -d space:0.0 -font 9x15 -bg blue -fg white
```

will start **xjed** using the server on amy using a white on blue 9x15 font.

Once the X Window resource specifications have been parsed, any remaining command line arguments are parsed as normal **jed** command line arguments.

The easiest way to specify the resources is to make use of a `.Xdefaults` in your `$HOME` directory. Here is an example entry for **xjed**:

```
xjed*Geometry: 80x36+100+100
xjed*font: 10x20
xjed*background: white
xjed*foreground: black
xjed*fgNumber: blue
```

The first line specifies that the initial window size is 80 columns by 36 rows and that the top left corner of the window is to be positioned at (100, 100). The second line specifies a fixed 10x20 font. The other two lines specify the foreground and background colors of the window.

17.2 Mouse Usage

`x_set_window_name`	Set the name of the window (for title bar)
`x_warp_pointer`	Move mouse position to cursor position
`x_insert_cutbuffer`	insert contents of system cut buffer in current buffer
`x_copy_region_to_cutbuffer`	insert a region in system cutbuffer
`x_set_keysym`	define an equivalence string to be returned when a function key is pressed

also, `set_color()` may be used to set colors of mouse, cursor, normal, region, and status line as well as the colors used by the syntax highlighting routines. For example,

```
set_color ("mouse", "red", "blue");
```

gives the mouse cursor a red forground with a blue background. The color values must be recognizable by the X server.

In addition to the usual keybindings, the X version binds:

`Control-UP`	goto top of buffer
`Control-DOWN`	goto end of buffer
`Shift-UP`	move to top of window
`Shift-DOWN`	move to bottom of window
`Control-RIGHT`	Pan the window to the right
`Control-LEFT`	Pan the window to the left
`Shift-RIGHT`	skip to next word
`Shift-LEFT`	skip to previous word

17.3 EDT emulation under Linux

Angelo Pagan (pagan@astrpd.pd.astro.it) suggests putting

```
keycode 22  = Delete
keycode 77  = KP_F1
keycode 112 = KP_F2
keycode 63  = KP_F3
keycode 82  = KP_F4
keycode 86  = KP_Separator
```

in the `.Xmodmap` file to enable EDT keypad emulation.

Send comments and suggestions to davis@space.mit.edu

18 Using a mouse with jed and xjed

jed provides native support for a mouse on the following systems:

- A Linux console running the GPM server. This server is a replacement for the 'selection' program. It is available from `sunsite.unc.edu:/pub/Linux/system/Daemons/gpm-0.97.tar.gz`
- MSDOS
- **xjed**

Later, OS/2 support will be added.

In addition to "native" mouse support, **jed** is able to interact with a mouse using the 'XTerm Event Protocol'. Support for this protocol is available when running **jed** in an XTerm as well as interacting with **jed** from an MSDOS terminal emulator, e.g., MS-Kermit, using the PCMOUSE TSR.

This document is divided into two sections. The first section describes native mouse support (Linux, MSDOS, **xjed**) and the second section describes the support for the XTerm Event Protocol.

18.1 Native Mouse Support

The S-Lang file `jed/lib/mouse.sl` provides a user interface to the mouse. It can only be loaded for systems which provide native support for the mouse. Currently this includes MSDOS, Linux console, and **xjed**. This file is automatically loaded from os.sl when **jed** is started up. (See os.sl for how this is accomplished). Once this file has been loaded, the mouse buttons behave as described below.

This interface assumes the presence of a three button mouse. Unfortunately, in the MSDOS world, two button mice are are quite common. Nevertheless, **jed** is able to emulate a three button mouse by using the ALT key. Any button pressed in combination with the ALT key is considered to be the **middle** mouse button. For example, to get the effect of pressing Ctrl-Middle, hold down on the ALT and Ctrl key while pressing any mouse button.

Clicking in a window

Left If a region is already marked, simply un-mark it. If one is not marked, move cursor to the mouse point crossing windows if necessary. If the button is held down and the mouse is dragged, a region will be highlighted and then copied to the cutbuffer when the button is released.

Middle If a region is already marked, copy it to the mouse paste-buffer. Otherwise, paste text from the system cut buffer to current editing point. This **may not be** the position of the mouse.

Right If a region is already marked, delete it and place a copy into the mouse paste-buffer. Otherwise, simply position the editing point at the position of the mouse. If the button is held down and the mouse is dragged, a new region will be marked.

Shift Middle Insert contents of the last **jed** mouse copy or kill. This function may be identical to simply clicking on the middle button without using the shift key on non-X systems. Simply clicking the middle mouse button will insert the contents of the current selection which may not be owned by **jed**.

Other buttons combinations are undefined. Some modes may utilize the mouse in a slightly different manner.

Clicking on a window status line

Left Switch to next buffer

Ctrl-Left Kill buffer described by status line

Shift-Left Scroll window back one page

Shift-Right Scroll window forward one page

Middle Split the window

Right Delete the window

For example, one can quickly move from one buffer to the next by simply clicking on the status line with the left mouse button.

Tips for using the mouse

- To quicky move the cursor to another location, simply point the mouse at that spot and click the LEFT mouse button.
- To copy a region for subsequent pasting, move the mouse to the beginning of the region and press the LEFT mouse button. While holding it down, “drag” the mouse to the end of the region and release it.
- To cut a region and put it in the paste buffer, define a region by dragging with the RIGHT mouse button. Now release the RIGHT button and then press and immediately release it.

18.2 XTerm Event Support

Xterm event support is provided by not only Xterm but also the Linux console running the 'selection' program. Only versions 1.6 and greater of selection provide this support. In addition, one must be using a recent Linux kernel (1.1.35 or newer.) Please note that the selection program is considered obsolete and should be replaced by the GPM mouse server.

Mouse Usage

Left Button If the left button is clicked on the status line of a window, the window will switch to a different buffer.
If the button is pressed anywhere else in the window, the cursor will be positioned at the location of the click.

Middle Button On status line: split the window
Anywhere else: If the region is highlighted, the region will be copied to the pastebuffer. This does not delete the region. Otherwise, the contents in the pastebuffer will be pasted to the current editing point.

Right Button On status line: delete the window.
Anywhere else: If a region is highlighted, the region will be extended to the position of the mouse. Otherwise, the mark is set and a region will be defined.

Cut/Paste Tips

To mark and manipulate a region do:

1. Click the LEFT mouse button at the beginning of the region.
2. Move the mouse to the end of the region and click the RIGHT mouse button. The region should now be marked.
3. Click the MIDDLE button to copy the region to the pastebuffer.
4. To paste from the pastebuffer, move the cursor to where you want to paste and press the MIDDLE button.

19 Frequently Asked Questions

How do I obtain jed?

jed is available via anonymous ftp from space.mit.edu in the `pub/davis/jed` directory. **jed** comes in three forms:

```
jedxxx.tar.Z     unix distribution for version xxx
jedxxx.*_of_n    n part VMS share distribution for xxx
jedxxx.zip       PC distribution with precompiled jed.exe
```

All distributions are identical except that the zip file also contains a precompiled executable for PC systems.

jed may also be obtained by email for those without ftp access. To learn about how to ftp using email, send email to ftpmail@pa.dec.com with the single line `help`. A message will be returned with instructions.

For those with VMS systems, Hunter Goatley has made **jed** available via anonymous ftp from ftp.spc.edu in `[.MACRO32.SAVESETS]JE`

This distribution includes VMS `.OBJ`s and a `.EXE` file that was linked under VMS V5.1. [Note that although this distribution is intended for VMS systems, it includes makefiles and sources for unix as well. However, you will need to get unzip for your unix system. –John]

How do I disable jed's C mode?

The startup file 'site.sl' contains the function `mode_hook` which is called whenever a file is loaded. This function is passed the filename extension. If a file with `c` or `h` extension is read, this function will turn on C-mode for that buffer. You could modify this function to not select C-mode. However, this is not recommended. Rather, it is recommended that you simply rebind the offending keybinding. These include: `{`, `}`, the `TAB` key, and the `RETURN` key.

Simply put any or all of:

```
"self_insert_cmd"  "{"   setkey
"self_insert_cmd"  "}"   setkey
"self_insert_cmd"  "^I"  setkey
"newline"          "^M"  setkey
```

in your personal startup file (`jed.rc` or `.jedrc`).

Before you do this, are you sure that you really understand what C mode does? If not, please read on.

What is C mode?

In C mode, the `TAB` key does not insert tabs. Instead, it runs a command called `indent_line`. It is really the quickest way to edit C code. In this mode, the `TAB`, `RETURN`, `{`, and `}` keys are special.

If you edit a file called `x.c`, **jed** will invoke its C mode. Entering the 28 characters (no newline, `TAB`, etc...)

```
main (){if (x == 2){x = 4;}}'
```

should result in:

```
main () {
   if (x == 2) {
     x = 4;
   }
}
```

which would take alot more time using the `TAB` and `NEWLINE` keys. If you do not like the indentation style, you can customize it by setting the appropriate variables in `jed.rc`.

To see the use of the tab key, delete the whitespace in front of all the lines, move to any of the lines (anywhere on the line) and hit the `TAB` key. This should correctly indent the line to according to your preferences (i.e., the variables in `jed.rc`).

Finally, move to one of the lines and enter `ESC ;`. This should produce a C comment.

Using the C mode and the `TAB` key as `indent_line` also helps you avoid syntax errors. Basically, a line simply will not indent properly. This indicats that you left off a brace, mismatched parenthesis, etc... If you bind `TAB` away from `indent_line`, you lose some of this.

Note that these same comments apply to Fortran mode. Get a file called `x.for`. Enter the characters:

```
TABprogram mainRETinteger*4 iRETdo 10 i=1,3RETcall f(i)RET10continueRETend
```

Here `TAB` means hit `TAB` and `RET` means hit return. This will result in:

```
        program main
integer*4 i
do 10 i=1,3
  call f(i)
   10   continue
        end
```

Again, the editor has done all the work. Once you get used to this style of editing, you will not want to go back.

Also note that this will not work if EDT is loaded. To get this functionality back, you will need to do:

```
setkey("indent_line_cmd", "\t");
setkey("newline_and_indent_cmd", "^M");
```

AFTER edt.sl is loaded.

How do I turn on wrap mode or turn it off?

Normally, this is done automatically when **jed** loads a file with extensions `.txt`, `.doc`, etc... See question 2 for a discussion of how this is done. To turn on wrap mode for the current buffer, simply press `Escape-X` and enter:

```
text_mode
```

at the prompt. To turn it off, you must change the mode to something else. A fairly generic choice is the `no_mode` mode. To do this, press `Escape-X` and enter:

```
no_mode
```

at the prompt. It is easy to write a function to toggle the mode for you that can be bound to a key. This one (`toggle_wrapmode`) will work:

```
define toggle_wrapmode ()
{
   variable mode, modestr;
   (modestr, mode) = whatmode ();
   if (mode & 1)          % test wrap bit
     mode = mode & ~(1);  % wrap bit on so mask it off
   else mode = mode | 1;  % wrap bit off so set it.
   setmode (modestr, mode);
}
```

What is the difference between internal and intrinsic functions?

An intrinsic function is a function that is directly callable from S-Lang while an internal function cannot. However, internal functions can be called indirectly through the use of the intrinsic function `call`. For example, consider the internal function `self_insert_cmd`. Most typing keys are bound to this function and cause the key to be directly inserted into the buffer. Consider the effect of this. After a character to be inserted is received by **jed**, the buffer is updated to reflect its insertion. Then the screen is updated. Here lies the essential difference between the two types of functions. If the screen was in sync before the insertion, **jed** can simply put the terminal in insert mode, send out the character and take the terminal back out of insert mode. However, this requires knowing the state of the screen. If called from a S-Lang routine, all bets are off. Since the screen update is not performed until after any S-Lang function has returned to **jed**, the buffer and the screen will almost always be out of sync with respect to one another and a full screen update will have to be performed. But this is very costly to have to do for every insertion. Hence, **jed** makes a distinction between the two types of functions by making the most common ones internal. The upshot is this: intrinsic functions will cause a full screen update while internal ones may not.

Sometimes during screen updates, jed pauses. Why is this?

Since version 0.91, **jed** checks the baud rate and tries to output characters based on reported rate. **jed** will literally sleep when outputting many characters if the reported baud rate is small. One should first check to see that terminal driver has the baud rate set appropriately. On Unix, this is done by typing `stty -a` at the shell prompt. If setting the baud rate to the correct value does not help, set the internal global variable `OUTPUT_RATE` to zero. This is achived by uncommenting the line referring to `OUTPUT_RATE` in the `jed.rc` initialization file. If there is still a problem, contact me.

How do I get jed to recognize Control-S and Control-Q?

Many systems use ^S/^Q for flow control— the so-called XON/XOFF protocol which is probably the reason **jed** does not see either of these two characters. Perhaps the most portable solution to this problem is to simply avoid using ^S and ^Q altogether. This may require the user to rebind those those functions that have key bindings composed of these characters.

jed is able to enable or disable flow control on the system that it is running. This may be done by putting the line:

```
enable_flow_control (0);  % turn flow control off
```

in your .jedrc file. Using a value of 1 turns flow control on.

Another solution is to use the map_input function to map a different key to ^S (and ^Q). For example, one might simply choose to map ^\ to ^S and ^^ (Control-^) to ^Q. To do this, simply put:

```
map_input (28, 19);    % ^\ --> ^S
map_input (30, 17);    % ^^ --> ^Q
```

in your .jedrc (jed.rc) file.

Can I bind the Alt keys on the PC?

Yes. The ALT keys return a two character key sequence. The key sequence for a particular ALT key as well as other function keys are listed in the file pc-keys.txt.

Many users simply want to use the ALT key as a Meta Character. To have **jed** interpret ALT-X as ESC-X, put

```
ALT_CHAR = 27;
```

int your jed.rc file. Here 'X' is any key. (Actually, this should not be necessary– the default value for ALT_CHAR is 27).

How do I find out what characters a particular key generates?

The simpliest way is to start **jed** via the command:

```
jed -l keycode -f keycode
```

jed will then prompt for a key to be pressed and return the escape sequence that the key sends. If **xjed** is used, it will also return the keysym (See online help on the x_set_keysym function for more information).

An alternative approach is to use the quoted insert function. By default, this is bound to the backquote ` key. Simply switch to the *scratch* buffer, press the backquote key followed by the key in question. The key sequence will be inserted into the buffer. This exploits the fact that most multi-character key sequences begin with the ESC character followed one or more printable characters.

If this fails, the following function will suffice:

```
define insert_this_key ()
{
  variable c;
pop2buf ("*scratch*");
eob ();
  message ("Press key:"); update (1);
forever
  {
    c = getkey ();
    if (c == 0) insert("^@"); else insert (char (c));
    !if (input_pending (3)) break;
  }
}
```

Simply type it into the scratch buffer, press `ESC-X` and type `evalbuffer`. Then, to use the function, press `ESC-X` again and enter `insert_this_key`.

jed scrolls slow on my WizBang-X-Super-Terminal. What can I do about it?

On Unix, **jed** uses termcap (terminfo) and the value of the `TERM` environment variable. Chance are, even though you are using an expansive state of the art terminal, you have told unix it is a vt100. Even if you have set the `TERM` variable to the appropriate value for you terminal, the termcap file may be missing entries for your "WizBang" features. This is particularly the case for Ultrix systems— the vt102, vt200, and vt300 termcap entries are missing the `AL` and `DL` termcap flags. In fact, the Ultrix man page for termcap does not even mention these capabilities!

jed is able to compensate for missing termcap entries only for vtxxx terminals. If your terminal is a fancy vtxxx terminal, put the line:

```
set_term_vtxxx (0);
```

in your `.jedrc` file.

How do I get a list of functions?

Help on any documented function is available by pressing 'Ctrl-H f' and entering the function name at the prompt. If you simply hit return, you will get the documentation for all functions.

How can I use `edt.sl` with `jed386.exe`?

The basic problem is the current generation of the 32 bit compiler (DJGPP) used to generate `jed386.exe` cannot handle the hardware keyboard interrupt used to remap the numeric keypad. Nevertheless, it is possible to use `edt.sl` with `jed386`. However, the function keys, `F1` to `F10` must be used for the EDT keypad.

The remapping is as follows:

```
                                        VT100 Keys
        IBM Function                  On the Numeric Keypad
   -------------------------      -------------------------
   |  F1 |  F2 | F3  | F4  |      | PF1 | PF2 | PF3 | PF4 |
   |-----+-----+-----+-----|      |-----+-----+-----+-----|
   |  F5 |  F6 | F7  | F8  |      |  7  |  8  |  9  |  -  |
   |-----+-----+-----+-----|      |-----+-----+-----+-----|
   |  F9 | F10 | F11 | F12 |      |  4  |  5  |  6  |  ,  |
   |-----+-----+-----+-----|      |-----+-----+-----+-----|
   | SF1 | SF2 | SF3 | SF4 |      |  1  |  2  |  3  |     |
   |-----------+-----|-----|      |-----------+-----|ENTER|
   | SF5 | SF6 | SF7 | SF8 |      |     0     |  .  |     |
   -------------------------      -------------------------
```

Here, `SF1` means `SHIFT-F1`, etc...

How do I set custom tab stops in jed?

Put something like:

```
   variable Tab_Stops;
   Tab_Stops = create_array('i', 20, 1);
   %% The following defines the tab stops to be 8 column:
   _for (0, 19, 1)
     { =$1;
Tab_Stops[$1] = $1 * 8 + 1;
     }
```

in your `jed.rc`. To individually set them, do:

```
Tab_Stops[0] = 4;
Tab_Stops[1] = 18;
```

etc...