



The intent® Shell User Guide

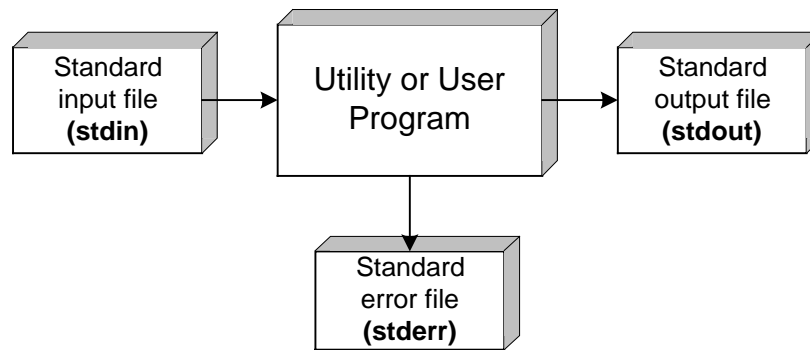
1. THE INTENT SHELL - AN INTRODUCTION	4
1.1 USING THE INTENT SHELL	4
1.2 OPTIONS	4
1.3 EXAMPLE COMMANDS	5
1.3.1 Listing Files - <i>ls</i>	5
1.3.2 Concatenating Files - <i>cat</i>	5
1.3.3 Change Directory - <i>cd</i>	5
1.3.4 Print Working Directory – <i>pwd</i>	5
1.3.5 Create New Directory – <i>mkdir</i>	6
1.3.6 Leaving <i>intent</i> - <i>Exit/Shutdown</i>	6
1.4 DIRECTORY STRUCTURE	6
2. AVAILABLE EDITORS	6
2.1 THE SHELL LINE EDITOR	6
2.1.1 Key Bindings	8
2.2 ED	9
2.2.1 Regular Expressions	9
2.2.2 Addresses	9
2.2.3 Ranges	10
2.2.4 Commands	10
2.2.5 Options	12
2.3 JOVE	12
2.3.1 Invoking <i>Jove</i>	12
2.3.2 Options	13
3. RECONFIGURING SHELL INTERACTION	13
3.1 SHELL INTERACTION	13
3.1.1 Options	13
4. ADVANCED USAGE	14
4.1 MULTITASKING FUNCTIONS	14
5. RUNNING SHELL SCRIPTS	16
6. SHELL VARIABLES	16
6.1 VARIABLES NAMES	17
6.2 \$ EXPANSION OF ENVIRONMENTAL VARIABLES	17
7. FILENAME GENERATION (GLOBBING)	19
7.1 PATTERN MATCHING	19
8. REDIRECTIONS	20
8.1 PIPING	20

9. EXCEPTIONS	22
10. SHELL GRAMMAR REFERENCE GUIDE.....	24
10.1 EBNF SYNTAX	24
10.2 COMMAND LISTS	24
10.3 COMMANDS	24
10.4 SIMPLE COMMANDS	24
10.5 REDIRECTIONS	25
10.6 ARGUMENTS.....	26
10.7 WHITESPACE.....	26

1. The intent® Shell - An Introduction

The intent® shell is a scripting command language interpreter. It is able to read and execute commands from the user, and can therefore provide an interface to the underlying Operating System.

Commands can be entered at the \$ command line prompt, which are then dealt with by the command processor, which calls on the services provided by the intent kernel as and when required. The results of most commands are displayed to the location at which standard output has been specified, for instance the screen. The intent shell has a similar feel to standard UNIX® shells, but although designed to offer a level of functionality comparable to a zsh shell, it also has a much smaller footprint.



1.1 Using The intent Shell

All *stdio* programs can be run from the shell prompt. All the commands described in this document are to be found in the '*app/stdio*' directory, although it is perfectly possible to run external commands that don't exist anywhere in the filesystem.. The shell searches all directories in the path (*\$shell.path*), which is normally set up to contain only */app/stdio*.

For further information upon the intent Shell Commands please see "*The intent Shell Commands Reference Manual*."

Commands can be entered at the \$ prompt. Where a brackets <thus> should be replaced by an actual parameter when typing a command. Parts of the command line shown in square brackets [thus] are optional.

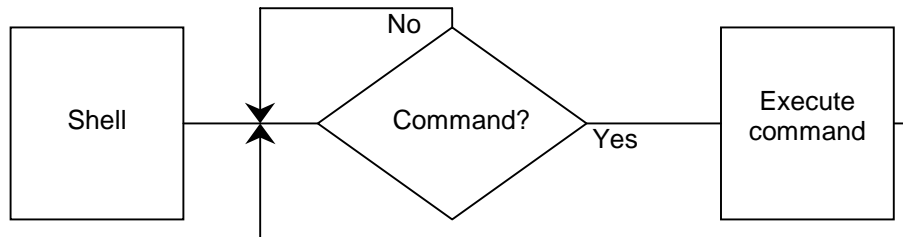
So for example:

```
command <parameter1> [<parameter2> [<parameter3> ...]]
```

In this case parameter 1 is mandatory, and parameters 2 and 3 are optional. However, parameter 3 cannot be specified unless parameter 2 also is. Parameter 3 may be repeated.

1.2 Options

Commands can be modified by specifying additional options. All options must be preceded by "-". Multiple options can be specified together, so for example, "-abc" would behave identically to "-a -b -c". However some options may take additional arguments, in which case multiple options cannot be specified. Options can appear anywhere on the command line, between parameters. For clarity however, it is recommended that options be placed immediately after the command name, and therefore before any parameters.



Simple Shell Processing

1.3 Example Commands

1.3.1 Listing Files - *ls*

The *ls* command lists the named files. If a directory is named, its contents are listed. If no filenames are given, the contents of the current directory are listed instead. Typing this in after the command prompt:

```
$ ls
```

would produce something like this:

```
dev  ebug.exe  feq.exe  lang  makefile
app  docn
```

1.3.2 Concatenating Files - *cat*

The *cat* command is used to concatenate the contents of any named files. By typing in this:

```
$ cat
```

it is possible create a new file, from text entered at stdin. When a filename is specified like so:

```
$ cat [<filename> ... ]
```

the file in question is printed to the screen. For example:

```
$ cat demo/example/hello.asm
```

Multiple files can be specified.

1.3.3 Change Directory - *cd*

```
cd [<directory>]
```

This function changes the current working directory to another specified directory.

1.3.4 Print Working Directory – *pwd*

```
pwd
```

Displays the name of the current directory to standard output.

1.3.5 Create New Directory – *mkdir*

```
mkdir <pathname>
```

Creates directories with the pathnames that have been specified.

1.3.6 Leaving intent - *Exit/Shutdown*

Exit terminates the shell, while *shutdown* shuts down *intent*. These have almost the same effect from the initial shell, but nowhere else.

1.4 Directory Structure

Files within *intent* are organised into directory structures - as is commonly the case, a directory is itself a file, containing the name and locations of the files it contains. Consequently any commands that apply to files are also applicable to directories.

One or more files can be specified. The following directory areas can be used as a guide to locate files:

Applications	app	All applications
AVE	ave	Multimedia Toolkit
Com.uk	com	Java™ classes, following Java™ namespace conventions
Demonstration	demo	Example Programs
Device Drivers	dev	Device Drivers
TCP/IP Subsystem	etc	General configuration files often used by TCP/IP, network and Host OS related files
Fonts	fonts	TrueType® and PostScript® Type 1 fonts.
Home	home	The user's home directory
Java	java	Java™ Libraries
Languages	lang	Programming Languages
Libraries	lib	General Library
Sounds	sounds	Audio specific parts of multimedia toolkit
System	sys	System Directory

It is easiest to find a required program by following this directory structure.

2. Available Editors

2.1 The Shell Line Editor

The Shell Line Editor (SLE) is a command line editor for the *intent* shell. It consists primarily of shell functions, with a few builtins to perform the basic operations. The following keys can be used to perform a variety of operations, although it should be stressed that this choice of keymappings is easily re-configurable.

^A, Home key	beginning of line
^E, End key	end of line
^B, Left arrow key	back one character
^F, Right arrow key	forward one character
^D,	delete character under cursor
^H, ^?	delete character to left of cursor
^K,	delete to end of line
^C,	delete entire line
^P, Up arrow key	previous line in history
^N, Down arrow key	next line in history

The intent® Shell User Guide

Page Up key	Beginning of history
Page Down key	End of history
^Q, ^V	insert next character literally

Please note that the SLE reserves all environment variables whose names start with "sle."

The default `intent` keymapping is as follows. It is similar to the Emacs keymapping, except that Esc and Del act as they do in the doskey keymapping.

^A Home	Go To Beginning of Line
^B Left Arrow Key	Go Back A Character
^C	Delete Whole Line
^D	Delete Character or End of Line
^E End of Line	Go to End of Line
^F Right Arrow Key	Go Forward a Character
^H	Backwards Delete a Character
^I	Line Completion
^J ^M	Accept or Newline
^K	Delete to End
^L	Clear Screen
^N Down Arrow Key	Downwards Command History
^P Up Arrow Key	Upwards Command History
^Q	Quoted Insert
^R	Redraw
^U	Delete Whole Line
^V	VI Quoted Insert
Esc	Delete Whole Line
^?	Forward Delete Characters
PgUp	Go To Beginning of Command History
PgDn	Go To End of Command History
anything else	Self Insert

The following command can be used to re-set the default keymapping as and when required.

```
set sle.keymap elate
```

A range of keymappings is available within `intent`, the details of which are provided here for ease of reference. The following command will set the emacs keymapping as the default.

```
set sle.keymap emacs
```

The complete list of variables is as follows; "sle.keymap.elate", "sle.keymap.emacs", "sle.keymap.doskey."

2.1.1 Key Bindings

These bindings are merely the default. It is possible to reconfigure the intent Shell so as to obtain a key binding for any desired type of text editor, by redefining *sle keymap* within the file:

```
elate/home/*/shell.rc.
```

In terms of the overall character set, the intent Shell has been configured to emulate the key bindings as recognised by UNIX ® as a default.

See also - *parse*, and *display* in “*The intent Shell Commands Reference Manual*. “

2.2 ed

Synopsis

```
ed [<filename>]
```

Description

ed is the standard text editor. *ed* is a line editor, and reads textual commands from standard input. Consequently it can be used in scripts, to automate editing tasks, in addition to being usable interactively.

At all times, *ed* maintains a buffer. When editing a file, the buffer contents are not automatically reflected in the file contents - when editing is complete, the buffer must be explicitly written back to the file.

If a filename is specified on the command line, an `e' command with that filename as its argument will be executed before starting to read commands.

2.2.1 Regular Expressions

ed regular expressions are POSIX Basic Regular Expressions. Regular expressions are always delimited by a specific character, most commonly "/", the delimiter is not treated as a delimiter if it appears in the regular expression escaped by a backslash. Newlines cannot appear in regular expressions, which are always matched against single lines.

Where a regular expression can legitimately appear at the end of a line, the closing delimiter may be omitted, in which case the closing delimiter is implicitly added, and a `p' appended after the delimiter.

An empty regular expression is equivalent to the last regular expression encountered.

2.2.2 Addresses

Each address refers to a line in the buffer. There is also a notional `line zero', which does not correspond to any line in the buffer and is not valid for all commands. There is at all times an address known as the `current line'.

The following address forms are understood:

.	The current line
\$	The last line in the buffer (line zero if there are none).
<number>	The line with the specified number (which may be zero).
'<letter>	The mark referred to by the specified lowercase letter. Marks are defined using the `k' command.
/ <i><BRE></i> /	The first line matching the specified regular expression. The search starts on the line following the current line, and if it reaches the end of the buffer will wrap back to the first line and continue up to and including the current line. It is an error if no line matches.
? <i><BRE></i> ?	The first line matching the specified regular expression, searching backwards. The search starts on the line preceding the current line, and if it reaches the start of the buffer will wrap back to the last line and continue up to and including the current line. It is an error if no line matches.
[<address>]+[<number>]	The specified address (defaulting to the current line) is evaluated, and the specified offset (default 1) added.
[<address>]-[<number>]	The specified address (defaulting to the current line) is evaluated, and the specified offset (default 1) subtracted.

2.2.3 Ranges

Where a command expects a range - two addresses - two addresses may be specified separated by "," or ";". If separated by a comma, the two addresses are evaluated normally. If separated by a semicolon, the first address is evaluated normally, but the second address is evaluated with the current line temporarily set to the first address.

A range can also be specified as a single address, in which case the range endpoints are identical. The special range "," is shorthand for "1,\$", and ";" similarly stands for ".,\$".

In any case, to be valid, the endpoint of a range must not precede the start, in the buffer.

2.2.4 Commands

ed commands have a consistent form. There is an optional address or range, followed by a single-letter command, possibly followed by arguments. The command letter, and each part of the address, may be preceded by whitespace.

After the arguments, most commands may be suffixed by `l`, `n` or `p`. This has the effect of executing the `l`, `n` or `p` command after the main command has completed. These suffixes cannot be used on certain commands, noted below, that would interpret them as arguments.

Each command can be preceded by zero, one or two addresses. It is illegal to give a command more addresses than it wants. Each command that requires addresses has a default which is used if it is given zero addresses, and a single address can always be used as a range in which both addresses are identical. The command synopses below indicate the default address and the number of addresses required for each command. Address zero is invalid except where noted.

. a	Reads lines of text, terminated by either a line containing only "." or by the end of input. The text is inserted into the buffer after the addressed line. Address zero is valid, and causes the text to be inserted at the beginning of the buffer. The current line is set to the last inserted line, or the addressed line if there were none.
.,. c	The addressed lines are deleted, and then replaced by text read in the same manner as for the `a` command. The current line is set to the last inserted line. If no lines were inserted, the current line is set to the line after the last line deleted; if the lines were deleted from the end of the buffer then the current line is set to "\$".
.,. d	Deletes the addressed lines from the buffer. The current line is set to the line after the last line deleted; if the lines were deleted from the end of the buffer then the current line is set to "\$".
e []	The filename argument, if present, may be preceded by whitespace, and extends to the end of the line. Suffixes cannot be used. The entire contents of the buffer is deleted, and then the specified file read in in the manner of the `r` command. If no filename is specified then the currently remembered filename is used. The currently remembered filename is set to the filename that is used; if a shell escape is used as the filename then no filename is remembered. The user is protected from destroying a modified buffer with this command in the same way as described for the `q` command.
E []	This is identical to the `e` command, except that the user is not protected from destroying the buffer.
f []	The filename argument, if present, may be preceded by whitespace, and extends to the end of the line. Suffixes cannot be used. If a filename is specified, the currently remembered filename is set to the specified filename. Whether the name is changed or not, the currently remembered filename is then displayed on standard output.
h	A help message is displayed on standard output, explaining the last error that occurred.
H	Toggles a mode (initially off) in which a help message is displayed for each error that occurs, immediately after the "?" notification. If turning the mode on, and an error has already occurred, it is explained in the manner of the `h` command.
. i	Reads text in the same manner as the `a` command, inserting it before the addressed

	line. The current line is set to the last inserted line, or the addressed line if there were none.
<code>.,.+1 j</code>	If only one line is addressed, do nothing. Otherwise, join the addressed lines together, removing the intermediate newlines, and set the current line to the joined line.
<code>. k</code>	Sets the specified mark to the addressed line. There are 26 marks, referred to by lowercase letters. Marks remain attached to the same line regardless of how that line moves.
<code>.,. l</code>	Writes the addressed lines to standard output in a visually unambiguous form. Unprintable characters and backslashes are represented in the C backslash escape form; long lines are split with a backslash newline sequence; and each line is terminated by a "\$". This is identical to the output form of sed's `l' command. The current line is set to the last line displayed.
<code>.,. n</code>	The addressed lines are written to standard output, each preceded by its line number and a tab character. The current line is set to the last line displayed.
<code>.,. p</code>	The addressed lines are written to standard output. The current line is set to the last line displayed.
<code>P</code>	Toggles the display of a prompt when reading commands (initially off). The prompt is "*". The -p option sets the prompt string and causes prompt display to be initially enabled.
<code>q</code>	Causes ed to exit. End of file is also treated as a `q' command. If the buffer contents has been modified since the last `e' command or `w' command that wrote the entire buffer to a file, it is an error. However, if the `q' command is then repeated with no intervening commands, it will execute normally.
<code>Q</code>	Causes ed to exit. This is identical to the `q' command, except that the user is not protected from destroying a modified buffer.
<code>\$ r []</code>	The filename argument, if present, may be preceded by whitespace, and extends to the end of the line. Suffixes cannot be used. The specified file is read (by default the currently remembered filename). If the filename given begins with "!", the rest of the line is taken as a shell command which is run, and its output is read instead. If the last byte read is not a newline, then a newline is silently appended. The number of bytes read is written to standard output. The data read is inserted into the buffer after the addressed line. Address zero is valid, and causes the data to be inserted at the beginning of the buffer. The current line is set to the last inserted line, or the addressed line if there were none. If there is no currently remembered filename, and a filename not beginning with "!" is specified, then this filename becomes the currently remembered filename.
<code>.,.s/ <BRE>/ <replacement> / <flags></code>	Any character ("/" here) may be used to delimit the regular expression and replacement string. The closing delimiter of the replacement string may be omitted, in which case a `p' suffix will be implicitly appended to the command. On each of the addressed lines, search for the specified regular expression, and replace the first match with the specified replacement string. The current line is set to the last line on which a substitution occurred; it is an error if no substitutions occur. In the replacement string, "&" is replaced by the portion of the line that matched the regular expression, and "\1", "\2", etc. are replaced by the corresponding matching subexpression. Any character other than digits can be escaped by preceding it with a backslash.

The flags can be:

<code>g</code>	Substitute all substrings matching the pattern, not just the first.
<code><number></code>	Substitute the matching substring, instead of the first.
<code>.,. t[<address>]</code>	Inserts a copy of the addressed lines after the specified address (by default the current line). Address zero is valid for the target, and causes the text to be inserted at the beginning of the buffer. It is not permitted for the target to be within the range of copied lines. The current line is set to the last inserted line.
<code>u</code>	Undoes the last modification to the buffer, and sets the current line to what it was before the modification. The `u' command itself counts as a modification, so repeated use of the `u' command will flip between two states of the buffer.

	Note that changing marks and the current line does not count as a modification. Conversely, commands capable of modifying the buffer, such as `a`, count as undoable modifications even if they don't actually change the buffer contents.
1, \$ w [<filename>]	The filename argument, if present, may be preceded by whitespace, and extends to the end of the line. Suffixes cannot be used. The addressed lines are written to the specified file (by default the currently remembered filename). If the filename given begins with "!", the rest of the line is taken as a shell command which is run, and the lines are written to its input. The number of bytes written is written to standard output. If there is no currently remembered filename, and a filename not beginning with "!" is specified, then this filename becomes the currently remembered filename.
\$ =	The line number of the addressed line is written to standard output. Address zero is valid.
!<command>	The command argument extends to the end of the line. Suffixes cannot be used. The specified command is executed. When it completes, a "!" is written to standard output. If the first character of the command is "!", it is replaced by the last shell command used via `!`; thus the ed command "!!" repeats the last `!` command.
.+1	The null command defaults to the `p` command, but has a different default address and only accepts a single address.

2.2.5 Options

-i	Enable interactive mode. By default, ed is in interactive mode if and only if its standard input is a tty. This mode affects the handling of error conditions. Whenever an error occurs, a "?" is displayed on standard output. In non-interactive mode, ed then terminates immediately. In interactive mode, however, only the current command is aborted, and further commands are accepted. In either case, when exiting, ed will exit with a non-zero exit status if and only if at least one error occurred.
-I	Disable interactive mode. See the description of interactive mode for the -i option.
-p <string>	Set the prompt string to the specified string, and enable its display. (See the description of the `P` command.)
-s	By default, the file reading and writing commands report the number of bytes they read or wrote, and the `!` command upon completion outputs a "!" to indicate so. This option suppresses these outputs. This is useful in scripts.

2.3 jove

Synopsis

```
jove [-d directory] [-w] [-t tag] [+<n> file] [-p file] [files] jove -r
```

Description

JOVE is based on the original EMACS editor written at MIT by Richard Stallman. Although JOVE is compatible with EMACS, there are some differences between the two editors and the user should not assume that the behaviour will be identical.

2.3.1 Invoking Jove

If JOVE is run with no arguments the user will be placed in an empty buffer, called Main. Otherwise, any arguments supplied are considered file names and each is "given" its own buffer. Only the first file is actually read in - reading other files is deferred until you actually try to use the buffers they are attached to.

This is for efficiency's sake, as most of the time, when JOVE is run on a big list of files, only a few of them are actually edited.

The intent® Shell User Guide

The names of all of the files specified on the command line are saved in a buffer, called *minibuf*. The minibuffer is a special JOVE buffer that is used when JOVE is prompting for some input to many commands (for example, when JOVE is prompting for a file name). When the user is being prompted for a file name, they can type Ctrl N and Ctrl P to cycle through the list of files that were specified on the command line. The file name will be inserted where the user is typing, who can then edit it, as if they had typed it in themselves.

Help on *jove* commands should be accessed via *jove* itself. To access it press Escape followed by ?. This will bring the 'describe-command' prompt. The user should then enter the command they want help on and it displays information on what keys are set up for that command and a short description of the command.

2.3.2 Options

-d	The following argument is taken to be the name of the current directory. This is for systems that don't have a version of C shell that automatically maintains the CWD environment variable. If -d is not specified on a system without a modified C shell, JOVE will have to figure out the current directory itself, which may be slower than normal. It is possible to simulate the modified C shell, by putting the following lines into the C shell initialisation file: <pre>(.cshrc): alias cd 'cd \!*; setenv CWD \$cwd' alias popd 'popd \!*; setenv CWD \$cwd' alias pushd 'pushd \!*; setenv CWD \$cwd'</pre>
+n	This option reads the file, as designated by the following argument, and positions the point at the n'th line instead of the (default) 1'st line. This can be specified more than once, although this is unlikely to be necessary. If no numeric argument is given after the +, the point is positioned at the end of the file.
-p	This option parses the error messages in the file designated by the following argument. The error messages are assumed to be in a format similar to the C compiler, LINT, or GREP output.
-t	This option runs the find-tag command on the string of characters, immediately following the -t if there is one (as in -tTagname), or on the following argument (as in -t Tagname) otherwise (see <code>ctags(1)</code>).
-w	This option divides the window in two. When this happens, either the same file is displayed in both windows, or the second file in the list is read in and displayed in its window.

3. Reconfiguring Shell Interaction

3.1 Shell Interaction

The intent Shell reads input from a source which can be selected as described in the Options section below. (If invoked with no arguments or options, it executes the interact command. For further information upon this command please see *'The intent Shell Commands Reference Manual.'*) The input is interpreted in accordance with the grammar presented below. The shell exits with the status of the last command it executed, or zero if it didn't have any commands to execute.

Synopsis

```
shell [<arg> ...]
```

This runs a shell. Shells may be run recursively to any number of levels (limited only by available memory).

3.1.1 Options

-c <command>	This function will only execute the specified command, in the manner of the <i>eval</i> command (please see list of shell commands for further information). This takes precedence over -s.
-s	Read commands from standard input. By default, commands are read from standard input if there are no command line arguments. If arguments are given, and this option is not used, the first argument is taken as a filename, and commands are read from that file, in the manner of the <i>source</i> command.
-I	This function commands the shell to cease being interactive. By default, the shell is

interactive if it is reading commands from standard input. If this is not the case the shell cannot be interactive.

An interactive shell will prompt for each line of input. It will execute each complete command as soon as it is read, rather than parsing the entire input before doing anything, and on a parse error or other meta-error will scrap only the current line and will prompt for a new command.

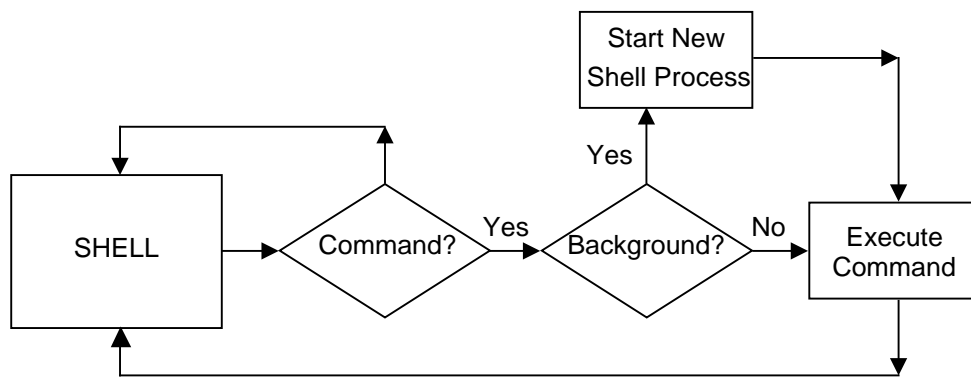
4. Advanced Usage

4.1 Multitasking Functions

In certain places a command may be run in a subshell, (if for instance, it is necessary to "background" a process). using the sub function means that the command is run in a separate process from the current shell environment. Redirections and shell variable settings in the subshell cannot affect the main shell. This is carried out by using the sub command. The syntax for this is:

```
sub <command>
```

If backgrounding of a process is required, then the *-b* option should be specified on the command line. Use of this option causes the shell to not wait for the subprocess to terminate before returning. Sub's exit status is zero. Exceptions thrown in a subshell cannot be caught in the invoking shell process, and the subshell's exit status is truncated to the size of a process exit status. The shell variable *shell.pid* always refers to the main shell's PID, even in a subshell.

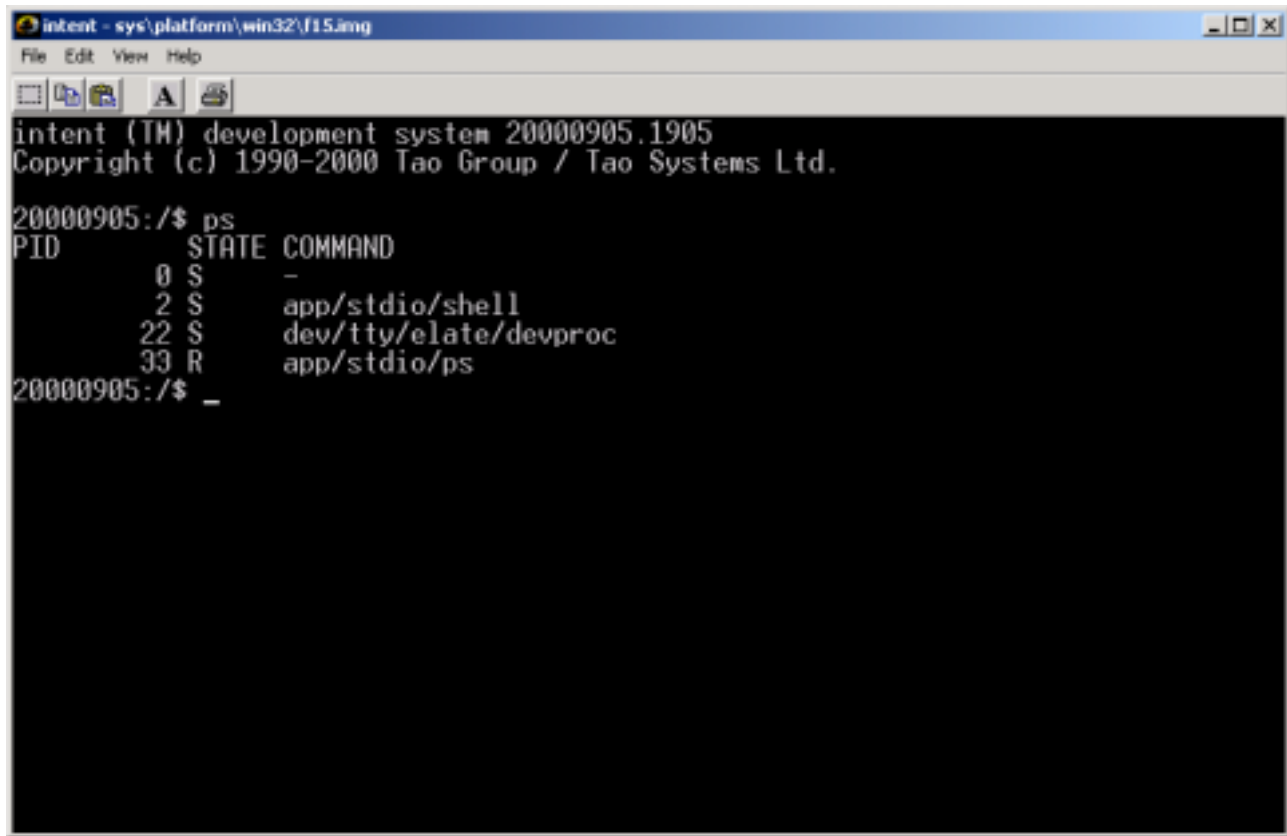


Shell Background Process

To obtain information about all processes being run by intent the user should use the *ps* command.

```
ps [<pid-list> ...]
```

This function displays the status of processes. The requisite processes may be specified by their process ID, but otherwise the status of all processes will be displayed. For instance:



```
intent - sys\platform\win32\l15.img
File Edit View Help
intent (TM) development system 20000905.1905
Copyright (c) 1990-2000 Tao Group / Tao Systems Ltd.

20000905:/$ ps
PID      STATE  COMMAND
   0  S      -
   2  S      app/stdio/shell
  22  S      dev/tty/elate/devproc
  33  R      app/stdio/ps
20000905:/$ _
```

Additional information concerning the *sub* or *ps* commands may be obtained from '*The intent Shell Commands Reference Manual*.'

5. Running Shell Scripts

To create a shell script type in the following:

```
$ cat > myscript.scr
ls
echo myscript.scr executed
Ctrl-D
```

To run this script enter either of the following commands at the shell prompt:

```
$ source myscript.scr
$ shell myscript.scr
```

By default script files are stored wherever they are created. Script files can contain any number of shell commands lines. Each line is executed sequentially, unless specified otherwise.

6. Shell Variables

As is the case with most programming languages the shell allows you to define variables, and can then keep track of an arbitrary number of them. The variables are local to the shell process, but are exported to the environment, which is itself a set of variables which all commands run have access to. Variables are initialised from the shell's environment.

The working environment is defined at login, and is set by using the values that the shell reads as it starts up. You can change your working environment by editing these files and setting new values for their variables. This can be done by using the set command, which is used to set the value of the specified shell variable to the array consisting of any arguments that may have been specified.

```
set <var> [ <arg> ... ]
```

For example:

```
$ set myvar /home/ hello
```

Following this type in:

```
$ ls $myvar
```

Note that this will turn the shell prompt into a part of the variable name. Further information upon this function can be found in *'The intent Shell Commands Reference Manual.'*

All shell variables are one-dimensional arrays of strings. The empty array is permitted, and is distinct from the array containing only the empty string. These are both, in turn, distinct from an unset variable, which has no value.

In the case of a variable whose value is the empty array, or has more than one element, the corresponding environment variable is only an approximation of the true value of the variable. However, the true array value is actually encoded in another environment variable, with a special name. This means that programs that know about this mechanism, such as the shell itself, can inherit array variable values.

6.1 Variables names

The shell permits any string to be used as a variable name. By convention, variable names are hierarchical, with a prefix indicating which part of the system uses the variable.

However, the shell reserves all variables whose names start with "shell.". Such variables may be modified at any time by the shell, and modifying them may have unexpected consequences.

The following shell variables have special meanings:

shell.	This prefix is reserved for use by the shell.
shell.argv	This is initialised to the shell's command line arguments, possibly after one has been taken as a script file name.
shell.func.<func>	Stores the definition of the shell function <func>.
shell.argv0	This is initialised to the shell's argv[0]. If a shell script file was specified on the shell's command line, its name is used instead.
shell.binpath	An array of directories to search to find tools to execute as builtin commands. If it is not set in the environment, the shell will initialise it to a suitable value to make it possible to execute the "set" command.
shell.path	An array of directories to search to find scripts and tools to execute as external commands.
shell.pid	Initialised to the process ID of the shell.
shell.ppid	Initialised to the parent process ID of the shell.
shell.spath	An array of tools to execute to try executing simple commands. Each tool on this path gets an attempt at each command, until one of them declares that it has succeeded in executing the command. If it is not set in the environment, the shell will initialise it to a suitable value to make it possible to execute builtin commands. This at least makes it possible to modify this variable.
shell.opt.glob	This variable contains the default options for globbing, when using the shorthand forms of globbing (ie, not using the glob modifier explicitly). If it is set to "N", glob commands will be expanded to the pattern if there are no matching files. If it is set to "n", a shell exception will be thrown if there are no matching files. The default value in interactive mode is "n". In non-interactive mode, the default value is "N".
shell.opt.interactive	This variable contains 0 if the shell is in non-interactive mode, or 1 if the shell is in interactive mode. This variable may be modified, which sets the shell into the specified mode, modifying the behaviour of various parts of the shell.
sys.	This prefix is reserved for system library use.
sys.cwd	The system uses the value of this environment variable as the "current directory" of each process. The shell itself does not treat this specially, but will be so affected by changes in its value. user. This prefix is reserved for user configuration parameters.
user.locale	The user's locale, used to determine how dates should be printed, what language messages should be displayed in, and so on. If not set, the default locale ("C") will be used.
user.tmp	The directory where temporary files should be stored (default "/tmp"). If this directory doesn't exist, or is not a directory, many programs will not be able to function correctly.
user.home	The user's home directory (default "/"). This is the default argument of the cd command

6.2 \$ Expansion of Environmental Variables

The shell automatically expands environment variables, and replaces them with their true value, so that should any word of any command executed, or the filename in any file redirection contain \$ sequences, they will be expanded before the command is actually executed. There are two means by which this can be done.

The simpler form looks like

```
"$myvar"
```

It takes the name of the specified shell variable (in this case myvar), and expands to the value of that variable. Note that all normal characters are valid after the \$, so:

```
"$shell.pid"
```

expands to the contents of the environment variable "*shell.pid*". It is an error for the variable not to exist.

If the variable name cannot be entered in this manner, it must be quoted and then wrapped in braces

```
"${thus}" ,
```

so:

```
"${{}}"
```

expands to the contents of the environment variable with the zero-length name. In the simplest case, the expression contains only a variable name, and the expansion is the variable's value. Unlike the form without braces, it is possible to specify a variable name that contains special characters, by quoting them, like "*\${this!}*". If the variable name is omitted entirely, "*\${}*", the expression expands to the empty array.

A *\${}* expression may be modified by appending a modifier introduced by a "!". Each modifier looks much like a normal command, but modifiers and commands are not interchangeable. Modifiers don't process options. Modifiers can take arguments, separated by whitespace. Multiple modifiers can be used in one expression, each one modifying the value resulting from the previous modifier. For example,

```
${foo ! e 3 !q}
```

takes the value of the variable *foo*, filters it through the modifier *e* with an argument of *3*, and then filters the result of that through the modifier *q*.

Each modifier has its own documentation entry, which can be found in the appendix attached to this document. The most important modifiers are:

c	count array length
e	select array elements
q	quote strings

If a \$ sequence expands to more than one word, everything that precedes and follows it in that word are duplicated, and are prepended and appended to each element of the expansion. If more than one \$ sequence in a single word does this, each one multiplies the number of resulting words. A \$ sequence expanding to the empty array (not the empty string) causes the word containing it to disappear. Each \$ sequence in a word is only expanded once, no matter how many words result.

7. Filename Generation (Globbing)

This section describes how the user can match a character or pattern that they have specified against the files in any directory, then making and displaying a list of all the matches. This is called globbing.

Each word of each command executed, and the filename in each file redirection, may contain special globbing characters. The reader should note that if a word contains any of the special characters in the table below, it is subject to filename generation, and will be replaced by a list of all files matching the glob pattern it represents, and should therefore bear this in mind when writing scripts and so on. The special sequences are:

?	match any single character
*	match any sequence of characters
[...]	match any of the enclosed characters
[^...]	match any character not listed
[!...]	same as [^...]
&[...]	match any of the listed patterns
x`	match zero or one x's
x'	match zero or more x's
x"	match one or more x's

The globbing characters are used in conjunction with the standard shell functions. For example,

```
ls -l *.txt
```

would produce a listing of all files bearing the suffix .txt.

All normal characters, and all quoted characters, stand for themselves. All characters that result from \$ expansion are regarded as quoted for this purpose.

Inside a [...] list, a range of characters can be given, as with regular expressions. "[a-e]" is thus equivalent to "[abcde]". For example:

```
ls -l [a-f]*.*
```

would match the listed characters in any files, while this:

```
ls -l [^acdef]*.*
```

would match any character not listed.

Any character inside [...] can be quoted using a backslash; this is the preferred way to get a literal "-" into the list. Alternatively, "-" can be placed first or last in the list. "]" can also be included literally by placing it first in the list.

A "/" must be matched explicitly. Pathname components starting with "." can only be matched if the corresponding component of the pattern starts with a literal ".". Pathname components "." and ".." can only be matched exactly.

7.1 Pattern Matching

When a shell builtin command or other shell internal provides pattern matching facilities, patterns may be specified as glob patterns. These are defined as in the above section, except that "/" and "." are not treated specially. Also, in most cases all non-glob characters stand for themselves, whereas in normal globbing cases many other special characters are active.

A repeat symbol (', " or `) can be followed by a ` to make it match the smallest number of times possible, rather than the largest.

8. Redirections

The characters described below can be used to redirect standard input (stdin), standard output (stdout), so that the output produced by one program is redirected to another file.

For example:

```
ls > demo/example/test.txt
```

Rather than printing to the screen it prints to *demo/example/test.txt* file. In this instance *demo/example/test.txt* is standard output for the ls process. It is also possible to redirect input so that whereas a program would otherwise obtain input from the stdin file, it can acquire it from *demo/example/in.dat* instead.

The symbols used for redirection are as follows:

>	Redirection of output
>>	Output appended to the end of the specified file rather than overwriting it. For instance: <pre>\$ ls >> test.txt</pre> will produce a listing within test.txt appended to the end of the file.
<	Redirection of input
<>	Redirection of both input and output

The output redirections will automatically create the file if it does not already exist. If the argument specified expands to multiple words, it is equivalent to specifying a separate redirection for each word, in sequence.

Where multiple redirections appear in a sequence, they are processed in order. A duplication duplicates the state of the file descriptor at that point in the sequence. The first redirection on any file descriptor replaces the former disposition of that file descriptor. Subsequent non-close redirections on the same file descriptor add to that file descriptor; multiple output redirections are implicitly "tee'd" (copied from standard input to standard output), multiple input redirections are implicitly concatenated. Multiple bidirectional redirections, or multiple redirections of incompatible modes, are an error.

8.1 Piping

Piping is a way for two processes to communicate with each other. Instead of redirecting to a file it is possible to pipe to another process. The '|' character redirects the stdout of the command on its left to the stdin of the command on its right. For instance,

```
dir | sort
```

Pipes output to another program. The basic variants upon this are as follows:

Redirect stdout to a file:

```
x >file
```

Redirect stdout appending to a file:

```
x >>file
```

Redirect stderr to a file:

```
x >(2)file
```

Redirect both stdout and stderr to the same file:

```
x >file >(2)=1
```

Pipe stdout to the stdin of the command y:

```
x | y
```

Pipe stderr to stdin:

```
x |(2>0) y
```

The capability is actually much wider than the standard piping syntax. In particular the shell also enables the user to use zsh-style multios. For instance:

```
ls >x >y
```

copies output to more than one place, and

```
tr a-z A-Z <x <y
```

concatenates input from more than one place (the tr command copies standard input to standard output, modifying the data as specified by any options and arguments. For more information upon this command please see *'The intent Shell Commands Reference Manual'*). Combining these capabilities,

```
ls | < x tr a-z A-Z
```

takes input both from a command and a file (concatenating). However,

```
ls > x | tr a-z A-Z
```

will not copy output to both a file and a command, because pipelines associate left-to-right.

```
tr a-z A-Z |(<) ls > x
```

will copy output to a command and a file, or it can be more clearly expressed as

```
ls >x %>{tr a-z A-Z}
```

9. Exceptions

In order to avoid the necessity for special casing the shell supports an exception mechanism. An exception is identified by an arbitrary string, determined when the exception is thrown. Unless caught, an exception will abort enclosing shell constructs, and ultimately terminate the shell.

Exceptions can be generated by error conditions detected by the shell, or by user commands. As exceptions are purely a shell concept, external commands cannot generate exceptions - only things done within the shell, including builtin commands and shell functions, can.

Exceptions conventionally start with a hierarchical identifier, in the manner of environment variable names. This makes it easy to use pattern matching to classify an exception with any desired granularity. Following the hierarchical part, exceptions may contain an error message, separated by a "!". If an uncaught exception causes the shell to terminate, this error message is displayed. If the exception does not have such an error message, a standard message is used instead.

The exceptions used by the shell are:

shell	All shell-generated exceptions have this prefix
shell.err	Error detected by the shell
shell.err.emptycmd	An empty simple command was executed
shell.err.ext	Error creating process for external command
shell.err.file	Error reading file
shell.err.glob	Globbering error
shell.err.mod	Error executing expansion modifier
shell.err.nocmd	A non-existent command was executed
shell.err.novar	A non-existent variable was referenced
shell.err.parse	Parse error
shell.err.redir	Redirection error
shell.err.sub.	Error creating process for subshell
shell.err.usage	Bad usage of a builtin command
shell.exit	Shell exiting
shell.return	Returning from a shell function
shell.return.g	Return from the innermost function
shell.return.s.<func>	Return from function <func>
shell.return.x.<except>	Return and propagate exception <except>

Exceptions can be manually thrown using the builtin command `throw`, and caught using the shell function `catch`. For further information upon these please see *'The intent Shell Commands Reference Manual.'*

```
throw <exception> [<status>]
```

Description

This throws the specified exception. The specified numerical status (default zero) is used for this.

```
catch <try-command> <exception-variable> <status-variable> <catch-command>
```

Description

This function executes the `<try-command>`, and saves its exit status in the variable `<status-variable>`. If it has exited normally, set `<exception-variable>` to an empty array, and return that status. If it exited with an exception, save the exception in `<exception-variable>`, and execute the `<catch-command>`.

Normally the `<catch-command>` should rethrow the caught exception if it is of no interest.

Under normal circumstances, one does not wish to catch absolutely all exceptions, and must therefore be careful to rethrow a caught exception. Typically this looks like this:

```
catch {
  ... # code that might throw an exception
}
except status {(Note 1)
case -- $except ~ (Note 2)
  {...} {
  ... # handle an exception of interest
  } ~ ...
  {*} { (Note 3)
  throw -- $except $status (Note 4)
  }
}
}
```

Note 1. This should not be represented as `$except` or `$status` - these are arguments to the `catch` command, telling it the names of the variables in which to store the exception and exit status.

Note 2. The tilde here prevents the following newline from terminating the command.

Note 3. The asterisk here is a pattern matching anything - the default case.

Note 4. This default exception handler is used to rethrow the code if required.

10. Shell Grammar Reference Guide

10.1 EBNF syntax

This section describes the basic syntax of the shell programming language. It is laid out in the form of BNF syntax, i.e recursively with each following line defining its predecessor. However two extensions to common Backus Nauer Form syntax are used:

FOO-seq	: FOO
"	FOO-seq FOO
FOO-opt	: <empty> FOO

In the first of these FOO-seq may stand either for a single command, or for a sequence of commands. FOO-opt represents optional commands, which may not be specified, or may be specified as an option to another command.

10.2 Command lists

CMD-LIST	: COMMAND-1-seq-opt COMMAND
COMMAND-1	: COMMAND TERM
COMMAND	: LWSP-opt CMD
TERM	: <newline> ";"

Here a command list is at least one command, or a list of commands. The input must be a sequence of COMMANDs, separated by TERMS. Please note that TERM is a separator, not a terminator; the final command is delimited by the end of input. In turn a command must be followed by a term, so that a command list could either be one command or a series of commands either separated by ; symbols or by newlines.

A command is a command followed by an optional whitespace followed by a command. See below for more information upon whitespace characters.

10.3 Commands

CMD : <empty>	PIPELINE
---------------	----------

The empty command does nothing; its sole effect is to permit multiple adjacent terminators. The pipeline is the usual form of command. As seen below a pipeline is either a simple command such as *ls* or *dir*, or an input redirection.

PIPELINE	: SIMPLE-CMD PIPELINE PIPE-REDIR SIMPLE-CMD
----------	--

The pipeline really runs its final simple command; the "pipeline pipe-redir" part acts like an command redirection for the simple command.

PIPE-REDIR : "	" PIPE-SPECS-opt WSP-opt
----------------	--------------------------

The | PIPELINE PIPE-REDIR SIMPLE-CMD e pipe-redir is syntactic sugar for a command redirection. "PIPELINE | CMD" is equivalent to "%<{PIPELINE} CMD" (see below). If a PIPE-SPECS is specified, it overrides the default. The usual syntax is reversed; "PIPELINE |(2>0) CMD" translates to "%(0<2){PIPELINE} CMD".

10.4 Simple Commands

SIMPLE-CMD	: PRE-REDIR-opt ARG CMD-PART- seq-opt LWSP-opt REDIR-LWSP-seq
------------	---

PRE-REDIR	: REDIR-LWSP-seq-opt REDIR LWSP
CMD-PART	: LWSP-opt REDIR LWSP ARG
REDIR-LWSP	: REDIR LWSP-opt

A simple command consists of a (possibly empty) sequence of redirections, and a (non-empty) sequence of arguments, intermixed freely. See below for further information on arguments. When a simple command is run, it does not terminate until not only the main command process, but also the processes associated with any command redirections, including pipelines have also terminated. The exit status of the whole command is the bitwise OR of the exit status of all these processes.

If no arguments are listed - only redirections - the redirections are applied to the shell itself, affecting all future commands. In this case, the processes associated with command redirections will not be not waited for.

10.5 Redirections

REDIR	: REDIR-OP FD-opt LWSP-opt REDIR-ARG
	CMD-REDIR
REDIR-OP	: ">"
	">>"
	"<"
	"<>"
FD	: "(DIGIT-seq-opt)"
REDIR-ARG	: ARG
	"=" LWSP-opt DIGIT-seq
	"=" LWSP-opt "-"

The "REDIR-OP" indicates the mode of the file descriptor affected, and determines which file descriptor is affected by default, i.e it determines to where output is to be redirected, and how. This is as follows:

>	1 Output
>>	1 Output (append)*
<	0 Input
<>	0 Input and output

- >> will append to the end of the file rather than overwrite that file.

The output redirections will create the file if it doesn't exist. If the ARG specified expands to multiple words, it is equivalent to specifying a separate redirection for each word, in sequence. If the ARG expands to no words, the redirection is ignored.

If "=" and a file descriptor are specified instead of a filename, the specified file descriptor is duplicated. A "-" means to close the file descriptor being modified.

Where multiple redirections appear in a sequence, they are processed in order. A duplication duplicates the state of the file descriptor at that point in the sequence. The first redirection on any file descriptor replaces the former disposition of that file descriptor. Subsequent non-close redirections on the same file descriptor add to that file descriptor; multiple output redirections are implicitly tee'd, multiple input redirections are implicitly cat'd. Multiple bidirectional redirections, or multiple redirections of incompatible modes, are an error.

CMD-REDIR	: "% PIPE-SPECS-1 LWSP-opt
PIPE-SPECS-1	: PIPE-OP
	PIPE-SPECS
PIPE-SPECS	: "(" ")"
	"(PIPE-SPEC PIPE-SPEC-1- seq-opt ")"
PIPE-SPEC-1	: "; PIPE-SPEC
PIPE-SPEC	: DIGIT-seq PIPE-OP DIGIT-seq
	"<" ">"
PIPE-OP	: "<" ">" "<>"

The CMD-LIST is executed in parallel with the main command. Pipes are constructed between these two processes. The CMD-REDIR construct acts like a sequence of redirections to the main process' end of these pipes.

Each PIPE-SPEC specifies a pipe; it contains the main process' file descriptor, the direction of the pipe, and the subordinate process' file descriptor, in that order. "<" indicates a pipe directed from the subordinate process to the main process (an input redirection), ">" the reverse, and "<>" a bidirectional pipe.

The commonest PIPE-SPECS can be abbreviated; "<" means "0<1", and ">" means "1>0". The entire PIPE-SPECS list can be similarly abbreviated: "<" means "(0<1)" ("(<)"), ">" means "(1>0)" ("(>)"), and "<>" means "(0<1;1>0)" ("(<>)"). It is an error to mention a file descriptor on either side more than once in the PIPE-SPECS.

10.6 Arguments

ARG	: ARG-1-seq
ARG-1	: ARG-NORM GLOB-PART
ARG-NORM	: NORM-CHAR QUOTE EXPANSION
QUOTE	: "\" <any char except NUL> BRACE-QUOTE
BRACE-PART	: QUOTE <any char except NUL, "\", "{" or ">"

Arguments can contain quoting, globbing and various forms of expansion. In a BRACE-QUOTE, only the outer braces are stripped off: everything inside them is a quoted part of the argument. This can be used to quote an arbitrarily complex set of shell commands as a single argument to a command.

EXPANSION	: "\$" NORM-CHAR-seq "\$" "{" LWSP-opt NAME-CHAR- seq-opt LWSP-opt MODIFIER-seq- opt "}"
NAME-CHAR	: NORM-CHAR QUOTE
MODIFIER	: "!" LWSP-opt NORM-ARG MOD- ARG-seq-opt LWSP-opt
MOD-ARG	: LWSP NORM-ARG
NORM-ARG	: ARG-NORM-seq

This syntax provides access to a wide range of forms of expansion. See the section "\$ Expansion" above.

GLOB-PART	: GLOB-SPEC GLOB-REP-opt ARG-NORM GLOB-REP
GLOB-SPEC	: "*" "?" "[" RANGE-NEG-opt "]"-opt RANGE-CHAR-seq "]" "&" "[" ARG-1-seq-opt "]"
RANGE-NEG	: "^" "!"
RANGE-CHAR	: NORM-CHAR "\" <any char except NUL>
GLOB-REP	: "`" "'" ""

Globbing is described in the section "Filename Generation", above.

10.7 Whitespace

LWSP-2	: <whitespace characters except newline> COMMENT
COMMENT	: "#" COMMENT-PART-seq-opt

The intent® Shell User Guide

	(see note below)
COMMENT-PART	: QUOTE
LWSP-1	: LWSP-2
	"~" LWSP-2-seq-opt <newline>:
LWSP	: LWSP-1-seq
WSP-1	: LWSP-1 <newline>
WSP	: WSP-1-seq

The COMMENT symbol always matches as much input as possible; it can only be followed by a newline, "}", end of input, or an incomplete QUOTE.

"#" starts a comment that continues to the end of the line. The content of comments must have balanced quoting, so that code containing comments will itself have balanced quoting. "~" can be used to continue a command across multiple lines.

© Tao Group Ltd or Tao Systems Ltd. 2000, 2001. All Rights Reserved.

Copyright in the software either belongs to Tao Group Ltd or Tao Systems Ltd. The software may not be used, sold, licensed, transferred, copied or reproduced in whole or in part or in any manner or form other than in accordance with the licence agreement provided with the software or otherwise without the prior written consent of either Tao Group Ltd or Tao Systems Ltd.

No part of this publication may be reproduced in any material form (including photocopying or storing it in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication) without the written permission of the copyright owner.

*Elate® is a registered trademark of Tao Group Ltd
intent™ is a trademark of Tao Group Ltd
Digital Heaven™ is a trademark of Tao Group Ltd
The rights of third party trademark owners are acknowledged.*