



Minishell

▼ Introduction

Minishell is a lightweight implementation of POSIX shell designed to provide a minimalistic command-line interface for interacting with an operating system.

Minishell will support the following features:

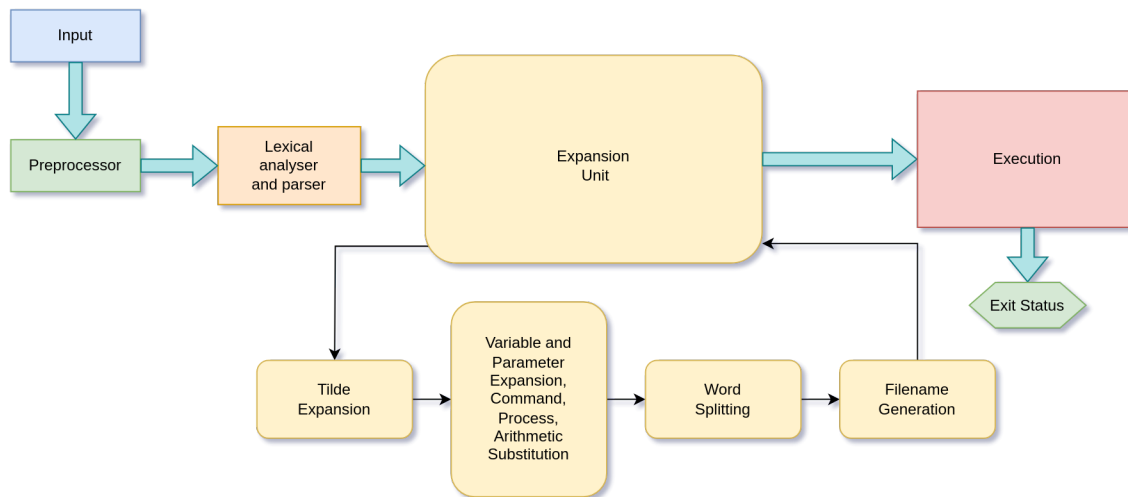
1. Display a user-friendly prompt when waiting for a new command, providing an intuitive user experience.
2. Maintain a working command history, allowing users to easily navigate and recall previously executed commands.
3. Search and launch the correct executable based on `PATH` variable or relative/absolute paths, offering a smooth command execution process.
4. Forgo support for unclosed quotes and special characters such as `\` (backlash) and `;` (semicolon), ensuring a cleaner and less error-prone user experience.
5. Support single quotes `'`, preventing the shell from interpreting characters within the quoted sequence, allowing users to work with literal strings.
6. Support double quotes `"`, preventing the shell from interpreting characters within the quoted sequence, except for the `$` (dollar sign) for

variable expansion.

7. Provide support for I/O redirections and here documents (`<` , `>` , `>>` , `<<`), enabling users to redirect input and output between commands and files.
8. Handle environment variables, allowing users to set, modify, and retrieve variables as needed.
9. Manage `ctrl-C` , `ctrl-D` and `ctrl-\` signals, ensuring proper behavior similar to that of `bash` .
10. Support special variables, such as `$?` , for context-based command behavior.
11. Support compound commands with a subshell execution `()` .
12. Support command lists with `&&` (AND_IF) and `||` (OR_IF) operators.
13. Wildcard `*` support for current working directory.
14. Implement built-in commands, including `echo` , `cd` , `pwd` , `export` , `unset` , `env` and `exit` , providing essential shell functionality out-of-the-box.

▼ Architecture

Minishell's architecture is designed to provide a modular and efficient command-line interface. The project is structured into six distinct modules, each responsible for a specific aspect of Minishell's functionality. These modules work together cohesively to deliver seamless user experience. The following sections outline each module's purpose and functionality.



Minishells architecture

Here's a description for each of the modules:

▼ Input

The Input module is responsible for reading user input and displaying a prompt when waiting for a new command. This module will be implemented using the `readline` library, which provides line editing and command history capabilities. By leveraging `readline`, Minishell ensures a user-friendly input experience with familiar key bindings and interactive features.

▼ Preprocessor

The Preprocessor module is responsible for preparing the raw input for parsing. This includes removing comments, handling line continuations, and performing other necessary preprocessing steps. By isolating the preprocessing logic, Minishell can maintain clean and organized code while ensuring accurate input interpretation.

Main purpose of preprocessor is to split

`command_list` into linked list of `complete_commands` to then parse them after previous command finishes execution.

▼ Lexical Analyzer and Parser

The Lexical Analyzer and Parser module is responsible for breaking the input into tokens based on the defined shell grammar and constructing an

Abstract Syntax Tree (AST) from these tokens. The Lexer recognizes and categorizes each token, while the Parser interprets the tokens and builds an AST according to the grammar rules. This module ensures that Minishell accurately understands the user's intended commands and control structures.

▼ Token Recognition

The shell shall read its input in terms of lines. The input lines can be of unlimited length. These lines shall be parsed using two major modes: ordinary token recognition and processing of here-documents.

When `io_here` token has been recognized by the grammar, one or more of the subsequent lines immediately following the next `NEWLINE` token form the body of one or more here-documents and shall be parsed according to the rules of Here-Document.

When it is not processing an `io_here` , the shell shall break its input into tokens by applying the first applicable rule below to the next character in its input. The token shall be from the current position in the input until a token is delimited according to one of the rules below; the characters forming the token are exactly those in the input, including any quoting characters. If it is not indicated that a token is delimited, and no characters have been included in a token, processing shall continue until an actual token is delimited.

1. If the end of input is recognized, the current token (if any) shall be delimited.
2. If the previous character was used as part of an operator and the current character is not quoted and can be used with the previous characters to form an operator, it shall be used as part of that (operator) token.
3. If the previous character was used as part of an operator and the current character cannot be used with the previous characters to form an operator, the operator containing the previous character shall be delimited.
4. If the current character is a single-quote, or double-quote and it is not quoted, it shall affect quoting for subsequent characters up to

the end of the quoted text. The rules for quoting are described in Quoting. During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for `<newline>` joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the `<quotation-mark>` and the end of the quoted text. The token shall not be delimited by the end of the quoted field.

5. If the current character is an unquoted `$`, the shell shall identify the start of any candidates for Parameter expansion from their introductory unquoted character sequences. The shell shall read sufficient input to determine the end of the unit to be expanded. While processing the characters, if instances of expansions or quoting are found nested within the substitution to its end, allowing or any recursion necessary to recognize embedded constructs, shall be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token shall not be delimited by the end of the substitution.
6. If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) shall be delimited. The current character shall be used as the beginning of the next (operator) token.
7. If the current character is an unquoted `<blank>`, any token containing the previous character is delimited and the current character shall be discarded.
8. If the previous character was part of a word, the current character shall be appended to that word.
9. The current character is used as the start of a new word.

Once a token is delimited, it is categorized as required by the grammar in Shell Grammar.

▼ Quoting

Quoting is a way to make sure that certain characters or words are treated as regular text by the shell. You can use quoting to keep the

special meaning of certain characters in the next paragraph, stop reserved words from being seen as such, and prevent parameter expansion within here-document processing.

To make sure that these characters are treated as regular text, the application shall quote them:

| | & < > () \$ " ' <space> <tab>
<newline>

These characters might need to be enclosed in quotation marks in certain situations. That is, their meaning might change based on conditions described elsewhere in this document.

| * ? = ~

The various quoting mechanisms are the single-quotes, and double-quotes. The here-document represents another form of quoting; see [Here-Document](#).

▼ Single-Quotes

Enclosing characters in single-quotes (' ') shall preserve the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

▼ Double-Quotes

Enclosing characters in double quotes (" ") shall preserve the literal value of all characters within the double-quotes, with the exception of the character `<dollar-sign>` as follows:

- The `<dollar-sign>` shall retain its special meaning introducing [Parameter Expansion](#).

The application shall ensure that a double-quote is preceded by a `<backslash>` to be included within double-quotes.

▼ Redirection

Redirection is used to open and close files for the current shell execution environment or for any command. Redirection operators can be used with numbers representing file descriptors as described below.

The overall format used for redirection is:

```
[n]redir-op word
```

The number `n` is an optional decimal number designating the file descriptor number; the application shall ensure it is delimited from any preceding text and immediately precede the redirection operator `redir-op`. If `n` is quoted, the number shall not be recognized as part of the redirection expression. For example:

```
echo '2'>a
```

writes character 2 into file `a`. If any part of `redir-op` is quoted, no redirection expression is recognized.

For example:

```
echo 2">"a
```

writes the character 2>a to standard output. The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

Open files are represented by decimal numbers starting with zero. The largest possible value is implementation-defined; however, all implementations shall support at least 0 to 9, inclusive, for use by the application. These numbers are called “file descriptors”. The values 0, 1 and 2 have a special meaning and conventional use and are implied by certain redirection operations; they are referred to as *standard input*, *standard output* and *standard error* respectively.

Of the redirection operator is “<<”, the word that follows the redirection operator shall be subjected to quote removal; it is unspecified whether any of the other expansions occur. For the other redirection operators,

the word that follows the redirection operator shall be subjected to tilde expansion, parameter expansion and quote removal. Pathname expansion shall not be performed on the word by a non-interactive shell; an interactive shell may perform it, but shall do so only when the expansion would result in one word.

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

A failure to open or create a file shall cause a redirection to fail.

▼ Redirecting Input

Input redirection shall cause the file whose name results from the expansion of word to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

```
[n]<word
```

where the optional `n` represents the file descriptor number. If the number is omitted, the redirection shall refer to standard input (file descriptor 0).

▼ Redirection Output

The general format for redirecting output are:

```
[n]>word
```

where the optional `n` represents the file descriptor number. If the number is omitted, the redirection shall refer to standard output.

Output redirection using the '>' format shall cause the file whose name results from the expansion of word to be created and opened for output on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

▼ Appending Redirected Output

Appended output redirection shall cause the file whose name results from the expansion of word to be opened for output on the designated file descriptor. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

```
[n]>>word
```

where the optional `n` represents the file descriptor number. If the number is omitted, the redirection refers to standard output.

▼ Here-Document

The redirection operator "<<" allows redirection of subsequent lines read by the shell to the input of a command. The redirected lines are known as a "here-document".

The here-document shall be treated as a single word that begins after the next `<newline>` and continues until there is a line containing only the delimiter and a `<newline>`, with no `<blank>` characters in between. Then the next here-document starts, if there is one. The format is as follows:

```
[n]<<word
    here-document
delimiter
```

where the optional `n` represents the file descriptor number. If the number is omitted, the here-document refers to standard input (file descriptor 0). It is unspecified whether the file descriptor is opened as a regular file, a special file, or a pipe.

If any part of *word* is quoted, the delimiter shall be formed by performing quote removal on *word*, and the here-document lines shall not be expanded. Otherwise, the delimiter shall be the *word* itself.

If no part of *word* is quoted, all lines of the here-document shall be expanded for parameter expansion. In this case, the `<backslash>` in the

input behaves as the the <backslash> inside double-quotes.

When a here-document is read from a terminal device and the shell is interactive, it shall write the contents of the variable `PS2`, processed as described in [Shell Variables](#), to standard error before reading each line of input until the delimiter has been recognized.

▼ Expansion Unit

The Expansion Unit is responsible for processing the AST and performing any necessary expansions, such as variable expansion and pathname expansion. Not all expansions are performed on every word, as explained in the following sections.

Tilde expansions, parameter expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple field from a single word.

The order of word expansion shall be as follows:

1. [Tilde Expansion](#), and [Parameter Expansion](#) shall be performed, beginning to end. See item 5 from [Token Recognition](#).
2. [Field Splitting](#) shall be performed on the portions of the field generated by 1, unless IFS is null.
3. [Pathname Expansion](#) shall be performed.
4. [Quote Removal](#) shall always be performed last.

This expansion described in this section shall occur in the same shell environment as that in which the command is executed.

If the complete expansion appropriate for a word results in an empty field, that empty field shall be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The `$` character is used to introduce parameter expansion. If an unquoted `$` is followed by a character that is not one of the following:

- A numeric character,
- The name of one of the special parameters,

- A valid first character of a variable name

the result is undefined.

▼ Tilde Expansion

A "tilde-prefix" consists of an unquoted `~` character at the beginning of a word, followed by all of the characters preceding the first unquoted `/` in the word, or all the characters in the word if there is no `/`. In an assignment, multiple tilde-prefixes can be used: at the beginning of the word (that is, following the `=` of the assignment), following any unquoted `:`, or both. A tilde-prefix in an assignment is terminated by the first unquoted `:` or `/`. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the `~` are treated as a possible login name from the user database. A portable login name cannot contain characters outside the set given in the description of the `LOGNAME` environment variable. If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix is replaced by the value of the variable `HOME`. If `HOME` is unset, the results are undefined. Otherwise, the tilde-prefix shall be replaced by a pathname of the initial working directory.

The pathname resulting from tilde expansion shall be treated as if quoted to prevent it being altered by field splitting and pathname expansion.

▼ Parameter Expansion

There are two formats for parameter expansion:

```
${expression}
$expression
```

where first `expression` consists of all characters except for `{`, `}` and `$`. Any `}` within a quoted string, and characters in embedded variable expansions, shall not be examined in determining the matching `}`.

The value, if any, of parameter shall be substituted.

The parameter expansion shall use the longest valid name, whether or not the variable represented by that name exists. Otherwise, the

parameter is a single-character symbol, and behavior is unspecified if that character is neither a digit nor one of the special parameters.

If a parameter expansion occurs inside double-quotes:

- Pathname expansion shall not be performed on the results of the expansion.
- Field splitting shall not be performed on the results of the expansion.

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of parameter, as described below), *word* shall be subjected to tilde expansion and parameter expansion. If *word* is not needed, it shall not be expanded.

▼ Field Splitting

After Parameter Expansion, the shell shall scan the results of expansion and substitutions that did not occur in double quotes for field splitting and multiple fields can result.

The shell shall treat each character of the *IFS* as a delimiter and use the delimiters as field terminators to split the results of parameter expansion, command substitution, and arithmetic expansion into fields.

1. If the value of *IFS* is a <space>, <tab>, and <newline>, or if it is unset, any sequence of <space>, <tab>, or <newline> characters at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall delimit a field. For example, the input:

```
<newline><space><tab>foo<tab><tab>bar<space>
```

yields two fields,

```
foo and bar .
```

2. If the value of *IFS* is null, no field splitting shall be performed.
3. Otherwise, the following rules shall be applied in sequence. The term "*IFS* white space" is used to mean any sequence (zero or more instances) of white-space characters that are in the *IFS*

value (for example, if *IFS* contains <space>/ <comma>/ <tab>, any sequence of <space> and <tab> characters is considered *IFS* white space).

- a. *IFS* white space shall be ignored at the beginning and end of the input.
- b. Each occurrence in the input of an *IFS* character that is not *IFS* white space, along with any adjacent *IFS* white space, shall delimit a field, as described previously.
- c. Non-zero-length *IFS* white space shall delimit a field.

▼ Pathname Expansion

After field splitting, each field in the resulting command line shall be expanded using the algorithm described Pattern Matching Notation.

▼ Quote Removal

The quote characters (single-quote, and double-quote) that were present in the original word shall be removed unless they have themselves been quoted.

▼ Execution Unit

The Execution Unit is responsible for executing the commands in the processed AST. This module handles built-in commands, process creation, and process management, while also managing input/output redirection and pipelines. The Execution Unit is the core of Minishell's command execution capabilities, ensuring that commands are executed efficiently and accurately.

A *command* is one of the following:

▼ Simple Commands

A "simple command" is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (that is, when any conditional construct such as AND-OR list has not bypassed the simple command), the following expansions, assignments, and

redirections shall all be performed from the beginning of the command text to the end:

1. The words that are recognized as variable assignments or redirections according to Shell Grammar are saved for processing in steps 3 and 4.
2. The words that are not variable assignments or redirection shall be expanded. If any fields remain following their expansion, the first field shall be considered the command name and remaining fields are the arguments for the command.
3. Redirections shall be performed as described in Redirection.
4. Each variable assignment shall be expanded for tilde expansion, parameter expansion and quote removal prior to assigning the value.

In the preceding list, the order of steps 3 and 4 may be reversed if no command name results from step 2 or if the command name matches the name of a special built-in utilities.

Variable assignments shall be performed as follows:

- If no command name results, variable assignments shall affect the current execution environment.
- If the command name is not a special built-in utility or function, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment except as a side-effect of the expansions performed in step 4. In this case it is undefined:
 - Whether or not the assignments are visible for subsequent expansions in step 4
 - Whether variable assignments made as side-effects of these expansion are visible for subsequent expansions in step 4, or in the current shell execution environment, or both.
- If the command name is a special built-in utility, variable assignments shall affect the current execution environment. it is

unspecified:

- Whether or not the variable gain the export attribute during the execution of special built-in utility
- Whether or not *export* attributes gained as a result of the variable assignments persist after the completion of the special built-in utility.

If there is a command name, execution shall continue as described in Command Search and Execution. If there is no command name, the command shall complete with a zero exit status.

▼ Pipeline

A *pipeline* is a sequence of one or more commands separated by the control operator '|'. For each command but the last, the shell shall connect the standard output of the command to the standard input of the next command as if by creating a pipe and passing the write end of the pipes as the standard output of the command and the read end of the pipe as the standard input of the next command.

The format for a pipeline is:

```
command1 [ | command2 ... ]
```

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command.

If the pipeline is not in the background, the shell shall wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

Exit Status

The exit status shall be the exit status of the last command specified in the pipeline. Otherwise, the exit status shall be the logical NOT of the exit status of the last command. That is, if the last command returns

zero, the exit status shall be the exit status of the last command specified in the pipeline.

▼ Lists

An *AND-OR list* is a sequence of one or more pipelines separated by the operators `&&` and `||`.

The operators `&&` and `||` shall have equal precedence and shall be evaluated with left associativity. For example, both of the following commands write solely **bar** to standard output:

```
false && echo foo || echo bar
true || echo foo && echo bar
```

The term "compound-list" is derived from the grammar. It is equivalent to a sequence of *lists*, separated by <newline> characters, that can be preceded or followed by an arbitrary number of <newline> characters.

AND Lists

The control operator `&&` denotes an AND list. The format shall be:

```
command1 [ && command2] ...
```

First, *command1* shall be executed. If its exit status is non-zero, *command2* shall be executed, and so on, until a command has a zero exit status or there are no more commands left to execute.

The exit status of an AND list shall be the exit status of the last command that is executed in the list.

OR Lists

The control operator `||` denotes an OR List. The format shall be:

```
command1 [ || command2] ...
```

First, *command1* shall be executed. If its exit status is non-zero, *command2* shall be executed, and so on, until a command has a zero exit status or there are no more commands left to execute.

The exit status of an OR list shall be the exit status of the last command that is executed in the list.

▼ Compound Command

The shell has several programming constructs that are “compound commands”, which provide control flow for commands. Each of these compound commands has a reserved word or control operator at the beginning, and a corresponding terminator reserved word or operator at the end. In addition, each can be followed by redirections on the same line as the terminator. Each redirection shall apply to all the commands within the compound command that do not explicitly override that redirection.

The format for grouping commands is as follows:

```
( compound-list )
```

Execute compound-list in a sub shell environment. Verbal assignments and built-in commands that affect the environment shall not remain in effect after the list finishes.

The exit status of a compound command shall be the exit status of compound-list.

Unless otherwise stated, the exit status of a command shall be that of the last simple command executed by the command. There shall be no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, and so on).

▼ Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions shall be performed:

1. If the command name does not contain any `/` characters, the first successful step in the following sequence shall occur:
 - a. If the command name matches the name of a special building-in utility, that special built-in utility shall be invoked.
 - b. Otherwise, the command shall be searched for using the *PATH* environment variable as described in [Environment Variables](#) :

- i. If the search is successful:
 1. If the system has implemented the utility as a regular built-in or as a shell function, it shall be invoked at this point in the path search.
 2. Otherwise, the shell executes the utility in a separate utility environment with actions equivalent to calling the `exec`.

Once a utility has been searched for and found (either as a result of this specific search or as part of an unspecified shell start-up activity), an implementation may remember its location and need not search for the utility again unless the `PATH` variable has been the subject of an assignment. If the remembered location fails for a subsequent invocation, the shell shall repeat the search to find the new location for the utility, if any.

- ii. If the search is unsuccessful, the command shall fail with an exit status of 127 and the shell shall write an error message.
2. If the command name contains at least one `<slash>`, the shell shall execute the utility environment with actions equivalent to calling the `exec` function.

▼ Exit Status Module

The Exit Status module is responsible for maintaining and reporting the exit status of executed commands. This module ensures that Minishell can correctly handle special variables such as `$?` and provide accurate exit status information to the user.

By structuring Minishell into these six modules, the project achieves a clean and organized architecture, allowing for efficient development and maintenance. This modular approach ensures that each aspect of Minishell's functionality is implemented in a focused and cohesive manner, ultimately delivering a powerful and user-friendly command-line interface.

▼ Built-In Commands

▼ cd - change the working directory

The `cd` utility shall change the working directory of the current shell execution environment by executing the following steps in sequence. (In the following steps, the symbol **cur path** represents an intermediate value used to simplify the description of the algorithm used by `cd`. There is no requirement that **curpath** be made visible to the application.)

1. If no *directory* operand is given and the *HOME* environment variable is empty or undefined, the default behavior is implementation-defined and no further steps shall be taken.
2. If no *directory* operand is given and the *HOME* environment variable is set to non-empty value, the `cd` utility shall behave as if the directory named in the *HOME* environment variable was specified as the *directory* operand.
3. If the *directory* operand begins with a <slash> character, set **curpath** to the operand and proceed to step 7.
4. If the first component of the *directory operand* is dot or dot-dot, proceed to step 6.
5. Starting with the first pathname in the <colon>-separated pathnames of *CDPATH* (see the ENVIRONMENT VARIABLES section) if the pathname is non-null, test if the concatenation of that pathname, a <slash> character if that pathname did not end with a <slash> character, and the *directory* operand names a directory. If the pathname is null, test if the concatenation of dot, a <slash> character, and the operand names a directory. In either case, if the resulting string names an existing directory, set **curpath** to that string and proceed to step 7. Otherwise, repeat this step with the next pathname in *CDPATH* until all pathnames have been tested.
6. Set **curpath** to the *directory operand*.
7. If **curpath** does not begin with <slash> character, set **curpath** to the string formed by the concatenation of the value of *PWD*, a <slash>

character if the value of *PWD* did not end with a <slash> character, and **curpath**.

8. The **curpath** value shall then be converted to canonical form as follows, considering each component from beginning to end, in sequence:
 - a. Dot components and any <slash> characters that separate them from the next component shall be deleted.
 - b. For each dot-dot component, if there is a preceding component and it is neither root nor dot-dot, then:
 - i. If the preceding component does not refer (in the context of pathname resolution with symbolic links followed) to a directory, then the *cd* utility shall display an appropriate error message and no further steps shall be taken.
 - ii. The preceding component, all <slash> characters separating the preceding component from dot-dot, dot-dot, and all <slash> characters separating dot-dot from the following component (if any) shall be deleted.
 - c. An implementation may further simplify **curpath** by removing any trailing <slash> characters that are not also leading <slash> characters, replacing multiple non-leading consecutive <slash> characters with a single <slash>, and replacing three or more leading <slash> characters with a single <slash>. If, as a result of this canonicalization, the **curpath** variable is null, no further steps shall be taken.
9. If **curpath** is longer than {PATH_MAX} bytes (including the terminating null) and the *directory* operand was not longer than {PATH_MAX} bytes (including the terminating null), then **curpath** shall be converted from an absolute pathname to an equivalent relative pathname if possible. This conversion shall always be considered possible if the value of *PWD*, with a trailing <slash> added if it does not already have one, is an initial substring of **curpath**. Whether or not it is considered possible under other circumstances is unspecified. Implementations may also apply this

conversion if **curpath** is not longer than {PATH_MAX} bytes or the *directory* operand was longer than {PATH_MAX} bytes.

10. 1. The *cd* utility shall then perform actions equivalent to the *chdir()* function called with **curpath** as the *path* argument. If these actions fail for any reason, the *cd* utility shall display an appropriate error message and the remainder of this step shall not be executed. The *PWD* environment variable shall be set to the value that **curpath** had on entry to step 9 (i.e., before conversion to a relative pathname). If there is insufficient permission on the new directory, or on any parent of that directory, to determine the current working directory, the value of the *PWD* environment variable is unspecified.

If, during the execution of the above steps, the *PWD* environment variable is set, the *OLDPWD* environment variable shall also be set to the value of the old working directory (that is the current working directory immediately prior to the call to *cd*).

▼ OPERANDS

The following operands shall be supported:

▼ *directory*

And absolute or relative pathname of the directory that shall become the new working directory. The interpretation of a relative pathname by *cd* depends on the **-L** option and the *CDPATH* and *PWD* environment variables. If *directory* is an empty string, the results are unspecified.

▼ -

When a <hyphen-minus> is used as the operand, this shall be equivalent to the command:

```
cd "$OLDPWD" && pwd
```

which changes to the previous working directory and then writes its name.

▼ **STDIN**

Not used

▼ **INPUT FILES**

None.

▼ **ENVIRONMENT VARIABLES**

The following environment variables shall affect the execution of `cd`:

▼ *CDPATH*

A <colon>-separated list of pathnames that refer to directories. The `cd` utility shall use this list in its attempt to change the directory, as described above. An empty string in place of a directory pathname represents the current directory. If *CDPATH* is not set, it shall be treated as if it were an empty string.

▼ *HOME*

The name of the directory, used when no *directory* operand is specified.

▼ *OLDPWD*

A pathname of the previous working directory, used by `cd -`.

▼ *PWD*

This variable shall be set as specified above. If an application sets or unsets the value of *PWD*, the behavior of `cd` is unspecified.

▼ **STDOUT**

If a non-empty directory name from *CDPATH* is used, or if `cd -` is used, an absolute pathname of the new working directory shall be written to the standard output as follows:

```
"%s\n", <new directory>
```

Otherwise, there shall be no output.

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXTENDED DESCRIPTION**

None.

▼ **EXIT STATUS**

The following exit values shall be returned:

- 0 - successful completion,
- >0 - an error occurred.

▼ **CONSEQUENCES OF ERRORS**

The working directory shall remain unchanged.

▼ **echo - write arguments to standard output**

The echo utility writes its arguments to standard output, followed by a <newline>. If there are no arguments, only the <newline> is written.

▼ **OPTIONS**

The following option shall be implemented:

▼ **-n**

Do not print the trailing newline character. This may also be achieved by appending `\c` to the end of the string.

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **STDOUT**

The *echo* utility arguments shall be separated by single <space> characters and a <newline> character shall follow the last argument.

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXTENDED DESCRIPTION**

None.

▼ **EXIT STATUS**

The following exit values shall be returned:

- 0 - succesful completion,
- >0 - An error occurred.

▼ **CONSEQUENCES OF ERRORS**

Default.

▼ **env - get the environment for command invocation**

The *env* utility shall obtain the current environment, and write the environment to the standard output, with one `name= value` pair per line.

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **STDOUT**

Each `name= value` pair in the resulting environment shall be written in the form:

```
"%s=%s\n", <name>, <value>
```

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXIT STATUS**


The *env* utility shall exit with one of the following values:

1. 0 - The *env* utility completed successfully.
2. (1 - 125) - An error occurred in the *env* utility.

▼ **CONSEQUENCES OF ERRORS**

Default.

▼ **export - set the export attribute for variables**

The shell shall give the export attribute to the variables corresponding to the specified names, which shall cause them to be in the environment of subsequently executed commands. If the name of a variable is followed by  *word*, then the value of that variable shall be set to *word*.

The shell shall format the output, including the proper use of quoting, so that it is suitable for rein put to the shell as commands that achieve the same exporting results, except:

1. Read-only variables with values cannot be reset.
2. Variables that were unset at the time they were output need not be reset to the unset state if a value is assigned to the variable between that time the state was saved and the time at which the saved output is rein put to the shell.

When no arguments are given, the results are unspecified.

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **ENVIRONMENT VARIABLES**

None.

▼ **STDOUT**

See above.

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXIT STATUS**

- 0 - All name operands were successfully exported.
- >0 - At least one name could not be exported.

▼ **CONSEQUENCES OF ERRORS**

Default.

▼ **exit - cause the shell to exit**

The exit utility shall cause the shell to exit from its current execution environment with the exit status specified by the unsigned decimal integer `n`. If the current execution environment is a sub shell environment, the shell shall exit from the sub shell environment with the specified exit status and continue in the environment from which that sub shell environment was invoked; otherwise, the shell utility shall terminate with the specified exit status. If `n` is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **ENVIRONMENT VARIABLES**

None.

▼ **STDOUT**

Not used.

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXIT STATUS**

The exit status shall be `n`, if specified, except that the behavior is unspecified if `n` is not an unsigned decimal integer or is greater than 255. Otherwise, the value shall be the exit value of the last command executed, or zero if no command was executed.

▼ **CONSEQUENCES OF ERRORS**

Default.

▼ **pwd - return working directory name**

The *pwd* utility shall write to standard output an absolute pathname of the current working directory, which does not contain the filenames dot or dot-dot.

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **ENVIRONMENT VARIABLES**

The following environment variables shall affect the execution of *pwd*:

1. *PWD* - An absolute pathname of the current working directory. If an application sets or unsets the value of *PWD*, the behavior of *pwd* is unspecified.

▼ **STDOUT**

The *pwd* utility output is an absolute pathname of the current working directory:

```
"%s\n", <directory pathname>
```

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXIT STATUS**

The *env* utility shall exit with one of the following values:

1. 0 - The *env* utility completed successfully.
2. >0 - An error occurred.

▼ **CONSEQUENCES OF ERRORS**

If an error is detected, output shall not be written to standard output, a diagnostic message shall be written to standard error, and the exit status is not zero.

▼ **unset - unset values and attributes of variables and functions**

```
unset name...
```

Each variable specified by `name` shall be unset.

Unsetting a variable that was not previously set shall not be considered an error and does not cause the shell to abort.

Note that

```
VARIABLE=  
# Is not the same as  
unset VARIABLE
```

▼ **STDIN**

Not used.

▼ **INPUT FILES**

None.

▼ **ENVIRONMENT VARIABLES**

None.

▼ **STDOUT**

Not used.

▼ **STDERR**

The standard error shall be used only for diagnostic messages.

▼ **OUTPUT FILES**

None.

▼ **EXIT STATUS**

- 0 - All name operands were successfully unset.
- >0 - At least one name could not be unset.

▼ **CONSEQUENCES OF ERRORS**

Default.

▼ **Implementation**

We will come back to this part when we begin implementing Minishell.

▼ **Appendix A: Shell Grammar**

The following grammar defines the Shell Command Language. This formal syntax shall take precedence over the preceding text syntax description.

▼ **Shell Grammar Lexical Conventions**

In order for the shell to properly understand user input, we first need to recognize the input language at the character level. This means breaking up the input into individual characters. Once we have all the characters, we can group them together based on their immediate context. For example, if we see a quotation mark, we know that the characters after that mark are part of a quoted string.

Once we have identified these groups, we can use a set of rules to determine what "tokens" are present in the input. Tokens are groups of characters that represent a meaningful unit of information, such as a command, argument, or operator.

These rules are used to identify tokens that need to be parsed at a deeper level. Parsing is the process of analyzing the structure of a sentence or

expression to determine its meaning. In the context of the shell, parsing involves identifying the individual components of a command and understanding how they relate to each other.

The rules for token recognition are explained in [Token Recognition](#). By following these rules, we can accurately understand user input and provide the correct output.

Tokens shall be classified by following these rules:

1. If the token is an operator, the token identifier for that operator shall result.
2. If the string consists solely of digits and the delimiter character is one of '<' or '>', the token identifier `IO_NUMBER` shall be returned.
3. Otherwise, the token identifier `TOKEN` results.

Further explanation regarding `TOKEN` depends on the context. The `TOKEN` can be transformed into `WORD`, `NAME`, or `ASSIGNMENT_WORD` depending on the situation. The grammar below includes some productions with rule numbers. If a `TOKEN` appears and matches one of the annotated productions, the corresponding rule should be used to convert the `TOKEN` into an acceptable token identifier for that point in the grammar. The reduction process will then proceed based on the resulting token identifier. If more than one rule can apply, always choose the one with the highest rule number (which may refer to another rule).

The `word` Tokens shall have a word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

▼ Shell Grammar Rules

▼ 1. [Command Name]

After the token is formed, token `WORD` shall be returned.

▼ 2. [Redirection to or from filename]

The expansions specified in [Redirection](#) shall occur. As specified there, exactly one field can result (or the result is unspecified), and there are additional requirements on pathname expansion.

▼ 3. [Redirection from here-document]

Quote removal shall be applied to the word to determine the delimiter that is used to find the end of the here-document that begins after the next `<newline>`.

▼ 4. [Assignment preceding command name]

▼ [When the first word]

If the `TOKEN` does not contain the character '=', rule 1 is applied. Otherwise, 4b shall be applied.

▼ [Not the first word]

If the `TOKEN` contains an unquoted (as determined while applying rule 4 from Token Recognition) `<equals-sign>` character that is not part of an embedded parameter expansion construct (as determined while applying rule 5 from Token Recognition):

- If the `TOKEN` begins with '=', then rule 1 shall be applied.
- If all characters in the `TOKEN` preceding the first such `<equals-sign>` form a valid name, the token `ASSIGNMENT_WORD` shall be returned.
- Otherwise, it is unspecified whether rule 1 is applied or `ASSIGNMENT_WORD` is returned.

Otherwise, rule 1 shall be applied.

Assignment to the name within a returned `ASSIGNMENT_WORD` token shall occur as specified in Simple Commands

▼ Context-Free Grammar

We will define our grammar in context-free grammar form that is similar to BNF (Backus-Naur form).

Symbols come in two forms:

- `terminal` - A "terminal" is a letter from the grammar alphabet. You can think of it as a literal value. In the syntactic grammar we're defining, the terminals are individual lexeme-tokens coming from the scanner like `|`

or `cat -e`. Terminals are named with all uppercase words, with or without underscored. Ex. `PIPE_OP` or `WORD`.

- **non-terminal** - A "non-terminal" is a named reference to another rule in the grammar. It means "play that rule and insert whatever it produces here".
- **production** - A production consists of a nonterminal called the head or left side of the production an arrow and a sequence of terminals and/or nonterminals called the body or right side of the production.

For example

```
production := production tokens
```

- We define the end of the rule with the symbol `;`
- We allow a series of productions separated by a `|` (pipe).
- We use the postfix operator `*` to allow the previous symbol or group to be repeated zero or more times.
- We use the postfix operator `+` to require the preceding production to appear at least once.
- We use a postfix operator `?` for optional production. The thing before it can appear zero or one time, but not more.

```
/* Terminal Symbols */
ALPHA  := 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
        | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
        | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
        | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' |
        ;

DIGIT   := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
        ;

EQUALS  := '='
        ;

PIPE_OP := '|'
```



```

        ;
    GREAT    := '>'
        ;
    LESS     := '<'
        ;
    DGREAT   := '>>'
        ;
    DLESS    := '<<'
        ;
    LPAREN   := '('
        ;
    RPAREN   := ')'
        ;
    USCORE   := '_'
        ;
    AND_IF   := '&&'
        ;
    OR_IF    := '||'
        ;
    WORD      := (ALPHA | DIGIT | USCORE)+
        ;
    NAME      := (ALPHA | USCORE) (ALPHA | DIGIT | USCORE)*
        ;
f
/* Starting non-terminal */
program := list
        ;
list    := pipeline list'
        ;
list'   := (AND_IF | OR_IF) list` | NULL
        ;

/* Command Pipelines */
pipeline    := command pipeline'
        ;
pipeline'   := PIPE_OP command pipeline` | NULL

```

```

;
command      := cmd_prefix (compound_command | simple_
;
compound_command := subshell ccmd_suffix
;
subshell      := LPAREN list RPAREN
;
simple_command := cmd_word cmd_suffix
;
cmd_word      := WORD | assignment_word
;
cmd_prefix    := io_redirect*
;
ccmd_suffix   := io_redirect*
;
cmd_suffix    := (WORD | io_redirect)*
;
assignment_word := NAME EQUALS WORD
;
io_redirect    := io_file | io_here
;
io_file        := LESS    filename
                | GREAT  filename
                | DGREAT  filename
;
filename       := WORD
;
io_here        := DLESS  here_end
;
here_end       := WORD
;

```

▼ Appendix B: Definitions

▼ Shell Variables

Variables shall be initialized from the environment and can be given new values with variable assignment commands. If a variable is initialized from the environment, it shall be marked for export immediately. New variables can be defined and initialized with variable assignments.

The following variables shall affect the execution of the shell:

▼ **HOME**

The pathname of the user's home directory. The contents of *HOME* are used in tilde expansion.

▼ **IFS**

A string treated as a list of characters that is used for field splitting, and expansion of the `*` special parameter. If the value of *IFS* includes any bytes that do not form part of a valid character, the results are unspecified.

If *IFS* is not set, it shall behave as normal for an unset variable, except that field splitting by the shell shall be performed as if the value of *IFS* is `<space> <tab> <newline>`.

▼ **PATH**

A string formatted as described in [Environment Variables](#), used to effect command interpretation.

▼ **PPID**

Set by the shell to the decimal value of its parent process ID during initialization of the shell. In a sub shell, *PPID* shall be set to the same value as that of the parent of the current shell. For example `echo $PPID` and `(echo $PPID)` would produce the same value.

▼ **PS1**

Each time an interactive shell is ready to read a command, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value shall be `"$ "`.

For users who have specific additional implementation defined privileges, the default may be another, implementation-defined value.

The shell shall replace each instance of the character '!' with the history file number of the next command to be typed.

▼ **PS2**

Each time the user enters a <newline> prior to completing a command line in an interactive shell, the value of this variable shall be subjected to parameter expansion and written to standard error.

The default value is "> ".

▼ **PS4**

When an execution trace is being performed in an interactive shell, before each line in the execution trace, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is "+ ".

▼ **PWD**

Set by the shell and by `cd` utility. In the shell the value shall be initialized from the environment as follows. If a value for `PWD` is passed to the shell in the environment when it is executed, the value is an absolute pathname of the current working directory that is no longer than {PATH_MAX} bytes including the terminating umm byte, and the value from the environment. Otherwise, if a value for `PWD` is passed to the shell in the environment when it is executed, the value is a absolute pathname of the current working directory, and the value does not contain any components that are dot or dot-dot, then it is unspecified whether the shell sets `PWD` to the value from the environment or sets `PWD` to the pathname that would be output by `pwd -p` . Assignments to this variable may be ignored. If an application sets or unsets the value `PWD`, the behaviors of `cd` and `pwd` are unspecified.

▼ **Name**

In the shell command language, a word consisting solely of underscores, digits, and alphabetic from the Portable Character Set. The first character of a name is not a digit.

▼ **Portable Character Set**

Conforming implementations shall support one or more coded character sets. Each supported locale shall include the *portable character set*, which is the set of symbolic names for characters in Portable Character Set.

Symbolic Name(s)	Glyph	UCS	Description
<NUL>		<U0000>	NULL (NUL)
<alert>, <BEL>		<U0007>	BELL
<backspace>, <BS>		<U0008>	BACKSPACE
<tab>, <HT>		<U0009>	CHARACTER TABULATION
<newline>, <LF>		<U000A>	LINE FEED (LF)
<vertical-tab>, <VT>		<U000B>	LINE TABULATION
<form-feed>, <FF>		<U000C>	FORM FEED (FF)
<carriage-return>, <CR>		<U000D>	CARRIAGE RETURN (CR)
<space>		<U0020>	SPACE
<exclamation-mark>	!	<U0021>	EXCLAMATION MARK
<quotation-mark>	"	<U0022>	QUOTATION MARK
<number-sign>	#	<U0023>	NUMBER SIGN
<dollar-sign>	\$	<U0024>	DOLLAR SIGN
<percent-sign>	%	<U0025>	PERCENT SIGN
<ampersand>	&	<U0026>	AMPERSAND
<apostrophe>	'	<U0027>	APOSTROPHE
<left-parenthesis>	(<U0028>	LEFT PARENTHESIS
<right-parenthesis>)	<U0029>	RIGHT PARENTHESIS
<asterisk>	*	<U002A>	ASTERISK
<plus-sign>	+	<U002B>	PLUS SIGN
<comma>	,	<U002C>	COMMA

<hyphen-minus>, <hyphen>	-	<U002D>	HYPHEN-MINUS
<full-stop>, <period>	.	<U002E>	FULL STOP
<slash>, <solidus>	/	<U002F>	SOLIDUS
<zero>	0	<U0030>	DIGIT ZERO
<one>	1	<U0031>	DIGIT ONE
<two>	2	<U0032>	DIGIT TWO
<three>	3	<U0033>	DIGIT THREE
<four>	4	<U0034>	DIGIT FOUR
<five>	5	<U0035>	DIGIT FIVE
<six>	6	<U0036>	DIGIT SIX
<seven>	7	<U0037>	DIGIT SEVEN
<eight>	8	<U0038>	DIGIT EIGHT
<nine>	9	<U0039>	DIGIT NINE
<colon>	:	<U003A>	COLON
<semicolon>	;	<U003B>	SEMICOLON
<less-than-sign>	<	<U003C>	LESS-THAN SIGN
<equals-sign>	=	<U003D>	EQUALS SIGN
<greater-than-sign>	>	<U003E>	GREATER-THAN SIGN
<question-mark>	?	<U003F>	QUESTION MARK
<commercial-at>	@	<U0040>	COMMERCIAL AT
<A>	A	<U0041>	LATIN CAPITAL LETTER A
	B	<U0042>	LATIN CAPITAL LETTER B
<C>	C	<U0043>	LATIN CAPITAL LETTER C
<D>	D	<U0044>	LATIN CAPITAL LETTER D

<E>	E	<U0045>	LATIN CAPITAL LETTER E
<F>	F	<U0046>	LATIN CAPITAL LETTER F
<G>	G	<U0047>	LATIN CAPITAL LETTER G
<H>	H	<U0048>	LATIN CAPITAL LETTER H
<I>	I	<U0049>	LATIN CAPITAL LETTER I
<J>	J	<U004A>	LATIN CAPITAL LETTER J
<K>	K	<U004B>	LATIN CAPITAL LETTER K
<L>	L	<U004C>	LATIN CAPITAL LETTER L
<M>	M	<U004D>	LATIN CAPITAL LETTER M
<N>	N	<U004E>	LATIN CAPITAL LETTER N
<O>	O	<U004F>	LATIN CAPITAL LETTER O
<P>	P	<U0050>	LATIN CAPITAL LETTER P
<Q>	Q	<U0051>	LATIN CAPITAL LETTER Q
<R>	R	<U0052>	LATIN CAPITAL LETTER R
<S>	S	<U0053>	LATIN CAPITAL LETTER S
<T>	T	<U0054>	LATIN CAPITAL LETTER T
<U>	U	<U0055>	LATIN CAPITAL LETTER U

<V>	V	<U0056>	LATIN CAPITAL LETTER V
<W>	W	<U0057>	LATIN CAPITAL LETTER W
<X>	X	<U0058>	LATIN CAPITAL LETTER X
<Y>	Y	<U0059>	LATIN CAPITAL LETTER Y
<Z>	Z	<U005A>	LATIN CAPITAL LETTER Z
<left-square-bracket>	[<U005B>	LEFT SQUARE BRACKET
<backslash>, <reverse-solidus>	\	<U005C>	REVERSE SOLIDUS
<right-square-bracket>]	<U005D>	RIGHT SQUARE BRACKET
<circumflex-accent>, <circumflex>	^	<U005E>	CIRCUMFLEX ACCENT
<low-line>, <underscore>	_	<U005F>	LOW LINE
<grave-accent>	`	<U0060>	GRAVE ACCENT
<a>	a	<U0061>	LATIN SMALL LETTER A
	b	<U0062>	LATIN SMALL LETTER B
<c>	c	<U0063>	LATIN SMALL LETTER C
<d>	d	<U0064>	LATIN SMALL LETTER D
<e>	e	<U0065>	LATIN SMALL LETTER E
<f>	f	<U0066>	LATIN SMALL LETTER F
<g>	g	<U0067>	LATIN SMALL LETTER G

<h>	h	<U0068>	LATIN SMALL LETTER H
<i>	i	<U0069>	LATIN SMALL LETTER I
<j>	j	<U006A>	LATIN SMALL LETTER J
<k>	k	<U006B>	LATIN SMALL LETTER K
<l>	l	<U006C>	LATIN SMALL LETTER L
<m>	m	<U006D>	LATIN SMALL LETTER M
<n>	n	<U006E>	LATIN SMALL LETTER N
<o>	o	<U006F>	LATIN SMALL LETTER O
<p>	p	<U0070>	LATIN SMALL LETTER P
<q>	q	<U0071>	LATIN SMALL LETTER Q
<r>	r	<U0072>	LATIN SMALL LETTER R
<s>	s	<U0073>	LATIN SMALL LETTER S
<t>	t	<U0074>	LATIN SMALL LETTER T
<u>	u	<U0075>	LATIN SMALL LETTER U
<v>	v	<U0076>	LATIN SMALL LETTER V
<w>	w	<U0077>	LATIN SMALL LETTER W
<x>	x	<U0078>	LATIN SMALL LETTER X

<y>	y	<U0079>	LATIN SMALL LETTER Y
<z>	z	<U007A>	LATIN SMALL LETTER Z
<left-brace>, <left-curly-bracket>	{	<U007B>	LEFT CURLY BRACKET
<vertical-line>		<U007C>	VERTICAL LINE
<right-brace>, <right-curly-bracket>	}	<U007D>	RIGHT CURLY BRACKET
<tilde>	~	<U007E>	TILDE

Portable character set defines the characters in the portable character set and the corresponding symbolic character names used to identify each character in a character set description file.

▼ Pattern Matching Notation

The pattern matching notation described in this section is used to specify patterns for matching strings in the shell. Historically, pattern matching notations is related to, but slightly difference from, the regular expression notation. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation, modified to account for the differences.

▼ Patterns Matching a Single Characters

The following patterns matching a single character shall match a single character: ordinary characters, special pattern characters, and pattern bracket expressions. The pattern bracket expression also shall match a single collating element. A <backslash> character shall escape the following character. The escaping <backslash> shall be discarded. If a pattern ends with an unescaped <backslash>, it is unspecified whether the pattern does not match anything or the pattern is treated as invalid.

An ordinary character is a pattern that shall match itself. It can be any character in the supported character set except for NUL, those special shell character in Quoting that require quoting, and the following three special pattern characters. Matching shall be based on the bit pattern used for encoding the character, not on graphic representation of the

character. If any character (ordinary, shell special, or pattern special is quoted, that pattern shall match the character itself. The shell special characters always require quoting.

When unquoted and outside a bracket expression, the following three characters shall have special meaning in the specification of patterns:

- * - An <asterisk> is a pattern that shall match multiple characters, as described in Patterns Matching Multiple Characters.

When pattern matching is used where shell quote removal is not performed (such as in the argument to the `find` - name primary when `find` is being called using one of the `exec` functions, or in the pattern argument to the `rematch()` function), special characters can be escaped to remove their special meaning by preceding them with a <backslash> character. This escaping <backslash> is discarded. The sequence “\\” represents one literal <backslash>. All of the requirements and effects of quoting on ordinary, shell special, and special pattern characters shall apply to escaping in this context.

▼ Patterns Matching Multiple Characters

The following rules are used to construct patterns matching multiple characters from patterns matching a single character:

1. The <asterisk> ('*') is a pattern that shall match any string, including the null string.
2. The concatenation of patterns matching a single character is a valid pattern that shall match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.
3. The concatenation of one or more patterns matching a single character with one or more <asterisk> characters is a valid pattern. In such patterns, each <asterisk> shall match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

▼ Patterns Used for Filename Expansion

The rules described so far in Patterns Matching a Single Character and Patterns Matching Multiple Characters are qualified by the following rules that apply when pattern matching notation is used for filename expansion:

1. The <slash> character in a pathname shall be explicitly matched by using one or more <slash> characters in the pattern; it shall neither be matched by the <asterisk> special character nor by a bracket expression. <slash> characters in the pattern shall be identified before bracket expressions; thus, a <slash> cannot be included in a pattern bracket expression used for filename expansion. If a <slash> character is found following an unescaped.
2. If a filename begins with a <period> ('.'), the <period> shall be explicitly matched by using a <period> as the first character of the pattern or immediately following a <slash> character. The leading <period> shall not be matched by:
 - The <asterisk> or <question-mark> special characters.
3. Specified patterns shall be matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character shall require read permission in the directory containing that component. Any component, except the last, that does not contain a pattern character shall require search permission. For example, given the pattern:

```
/foo/bar/x*/bam
```

search permission is needed for directories `/` and `foo`, search and read permissions are needed for directory `bar`, and search permission is needed for each `x*` directory. If the pattern matches any existing filename or pathnames, the pattern shall be replaced with those filename and pathnames, sorted according to the collating sequence in effect in the current locale. If this collating sequence does not have a total ordering of all characters, any filenames or pathnames that collate equally should be further

compared byte-by-byte using the collating sequence for the POSIX locale.

▼ Environment Variables

Environment variables defined in this chapter affect the operation of multiple utilities, functions, and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the ENVIRONMENT VARIABLES section of the utility descriptions for information on environment variable usage.

The value of an environment variable is a string of characters. For a C-language program, an array of strings called the environment shall be made available when a process begins. The array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form *name=value*; *names* shall not contain the character '='. For values to be portable across systems conforming to POSIX.1-2017, the value shall be composed of characters from the portable character set (except NUL and as indicated below). There is no meaning associated with the order of strings in the environment. If more than one string in an environment of a process has the same *name*, the consequences are undefined.

Environment variable names consist solely of uppercase letters, digits, and the <underscore> ('_') from the characters defined in Portable Character Set and do not begin with a digit. Other characters may be permitted by an implementation; applications shall tolerate the presence of such names. Uppercase and lowercase letters shall retain their unique identities and shall not be folded together. The name space of environment variable names containing lowercase letters is reserved for applications. Applications can define any environment variables with names from this name space without modifying the behavior of the standard utilities.

Note: Other applications may have difficulty dealing with environment variable names that start with a digit. For this reason, use of such names is

not recommended anywhere.

The *values* that the environment variables may be assigned are not restricted except that they are considered to end with a null byte and the total space used to store the environment and the arguments to the process is limited to {ARG_MAX} bytes.

It is unwise to conflict with certain variables that are frequently exported by widely used command interpreters and applications:

ARFLAGS	IFS	MAILPATH	PS1
CC	LANG	MAILRC	PS2
CDPATH	LC_ALL	MAKEFLAGS	PS3
CFLAGS	LC_COLLATE	MAKESHELL	PS4
CHARSET	LC_CTYPE	MANPATH	PWD
COLUMNS	LC_MESSAGES	MBOX	RANDOM
DATMSK	LC_MONETARY	MORE	SECONDS
DEAD	LC_NUMERIC	MSGVERB	SHELL
EDITOR	LC_TIME	NLSPATH	TERM
ENV	LDFLAGS	NPROC	TERMCAP
EXINIT	LEX	OLDPWD	TERMINFO
FC	LFLAGS	OPTARG	TMPDIR
FCEDIT	LINENO	OPTERR	TZ
FFLAGS	LINES	OPTIND	USER
GET	LISTER	PAGER	VISUAL
GFLAGS	LOGNAME	PATH	YACC
HISTFILE	LPDEST	PPID	YFLAGS
HISTORY	MAIL	PRINTER	
HISTSIZE	MAILCHECK	PROCLANG	
HOME	MAILER	PROJECTDIR	

If the variables in the following two sections are present in the environment during the execution of an application or utility, they shall be given the meaning described below. Some are placed into the environment by the

implementation at the time the user logs in; all can be added or changed by the user or any ancestor of the current process.

▼ Internationalization Variables

This section describes environment variables that are relevant to the operation of internationalized interfaces described in POSIX.1-2017.

Users may use the following environment variables to announce specific localization requirements to applications. The descriptions of the internationalization environment variables describe the resulting behavior only when the application locale is initialized in this way. The use of the internationalization variables by utilities described in the Shell and Utilities volume of POSIX.1-2017 is described in the ENVIRONMENT VARIABLES section for those utilities in addition to the global effects described in this section.

▼ *LANG*

This variable shall determine the locale category for native language, local customs, and coded character set in the absence of all *LC_ALL* and other *LC_** environment variables. This can be used by applications to determine the language to use for error messages and instructions, collating sequences, date formats, and so on.

▼ *LC_ALL*

This variable shall determine the values for all locale categories. The value of the *LC_ALL* environment variable has precedence over any of the other environment variables starting with *LC_** and *LANG* environment variable.

▼ *LC_COLLATE*

This variable shall determine the locale category for character collation. It determines collation information for regular expressions and sorting, including equivalence classes and multi-character collating elements, in various utilities. Additional semantics of this variable, if any, are implementation-defined.

▼ *LC_CTYPE*

This variable shall determine the locale category for character handling functions, such as `to_lower()`, `to_upper()`, and `is_alpha()`. This environment variable determines the interpretation of sequences of bytes of text data as characters (for example, single as opposed to multi-byte characters), the classification of characters (for example, alpha, digit, graph), and the behavior of character classes. Additional semantics of this variable, if any, are implementation-defined.

▼ *LC_MESSAGES*

This variable shall determine the locale category for processing affirmative and negative responses and the language and cultural conventions in which messages should be written. Additional semantics of this variable, if any, are implementation-defined.

▼ *LC_MONETARY*

This variable shall determine the locale category for monetary-related numeric formatting information. Additional semantics of this variable, if any, are implementation-defined.

▼ *LC_NUMERIC*

This variable shall determine the locale category for numeric formatting (for example, thousands separator and radix character) information in various utilities as well as the formatted I/O operations in `printf()` and `scanf()` and the string conversion functions in `strtod()`. Additional semantics of this variable, if any, are implementation-defined.

▼ *LC_TIME*

This variable shall determine the locale category for date and time formatting information. It affects the behavior of the time functions in `strptime()`. Additional semantics of this variable, if any, are implementation-defined.

▼ *NLSPATH*

This variable shall contain a sequence of templates that the catopen()

function uses when attempting to locate message catalogs. Each template consists of an optional prefix, one or more conversion specifications, a pathname, and an optional suffix.

For example:

```
NLSPATH="/system/nlslib/%N.cat"
```

defines that catopen() should look for all message catalogs in the directory **/system/nlslib**, where the catalog name should be constructed from the *name* parameter passed to catopen() (%N), with the suffix **.cat**.

▼ Other Environment Variables

▼ COLUMNS

This variable shall represent a decimal integer >0 used to indicate the user's preferred width in column positions for the terminal screen or window; see Column Position. If this variable is unset or null, the implementation determines the number of columns, appropriate for the terminal or window, in an unspecified manner. When *COLUMNS* is set, any terminal-width information implied by *TERM* is overridden. Users and conforming applications should not set *COLUMNS* unless they wish to override the system selection and produce output unrelated to the terminal characteristics.

Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behavior, such as to display data in an area arbitrarily smaller than the terminal or window.

▼ DATEMSK

Indicates the pathname of the template file used by getdate().

▼ HOME

The system shall initialize this variable at the time of login to be a pathname of the user's home directory. See `<pwd.h>`.

▼ *LINES*

This variable shall represent a decimal integer >0 used to indicate the user's preferred number of lines on a page or the vertical screen or window size in lines. A line in this case is a vertical measure large enough to hold the tallest character in the character set being displayed. If this variable is unset or null, the implementation determines the number of lines, appropriate for the terminal or window (size, terminal baud rate, and so on), in an unspecified manner. When *LINES* is set, any terminal-height information implied by *TERM* is overridden. Users and conforming applications should not set *LINES* unless they wish to override the system selection and produce output unrelated to the terminal characteristics.

Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behavior, such as to display data in an area arbitrarily smaller than the terminal or window.

▼ *LOGNAME*

The system shall initialize this variable at the time of login to be the user's login name. See `<pwd.h>`. For a value of *LOGNAME* to be portable across implementations of POSIX.1-2017, the value should be composed of characters from the portable filename character set.

▼ *MSGVERB*

Describes which message components shall be used in writing messages by `fntmsg()`.

▼ *PATH*

This variable shall represent the sequence of path prefixes that certain functions and utilities apply in searching for an executable file known only by a filename. The prefixes shall be separated by a <colon> (':'). When a non-zero-length prefix is applied to this

filename, a <slash> shall be inserted between the prefix and the filename if the prefix did not end in <slash>. A zero-length prefix is a legacy feature that indicates the current working directory. It appears as two adjacent <colon> characters (":"), as an initial <colon> preceding the rest of the list, or as a trailing <colon> following the rest of the list. A strictly conforming application shall use an actual pathname (such as .) to represent the current working directory in *PATH*. The list shall be searched from beginning to end, applying the filename to each prefix, until an executable file with the specified name and appropriate execution permissions is found. If the pathname being sought contains a <slash>, the search through the path prefixes shall not be performed. If the pathname begins with a <slash>, the specified path is resolved (see *Pathname Resolution*). If *PATH* is unset or is set to null, the path search is implementation-defined.

Since <colon> is a separator in this context, directory names that might be used in *PATH* should not include a <colon> character.

▼ *PWD*

This variable shall represent an absolute pathname of the current working directory. It shall not contain any components that are dot or dot-dot. This value is set by `cd` or `minishell` during initialization.

▼ *SHELL*

This variable shall represent a pathname of the user's preferred command language interpreter.

▼ *TMPDIR*

This variable shall represent a pathname of a directory made available for programs that need a place to create temporary files.

▼ *TERM*

This variable shall represent the terminal type for which output is to be prepared. This information is used by utilities and application programs wishing to exploit special capabilities specific to a

terminal. The format and allowable values of this environment variable are unspecified.

▼ *TZ*

This variable shall represent timezone information. The contents of the environment variable named *TZ* shall be used by the *ctime()*, *ctime_r()*, *localtime()*, *localtime_r()*, *strftime()*, *mktime()*, functions, and by various utilities, to override the default timezone.