Bash Syntax Cryptips

```
 ^ ^
(O,O)
(   ) LINUX F1
-"-"--------------
     ClusterBR

Document version v1.0.0
```

*The minimalistic style, backward compatibility, multiple ways to achieve the same result, and extensive use of special characters, along with the language's inherent concepts of streaming, piping, and redirecting, can make Bash scripting prone to cryptic and obscure syntax. This document summarizes some key features of **Bash scripting** syntax.*

1. **Command Substitution**: Runs command and returns its output, the backtick version is legacy, and $(...) is the preferred modern syntax.

```
$(command) or `command`
```

2. **Arithmetic Expansion**: Evaluates an arithmetic expression and returns the result.

```
$((expression))
```

3. **Parameter Expansion**: Allows manipulation of variables, including extracting substrings, modifying values, and applying operations, using the ${} syntax.

```
${var:2:4}
```

4. **Brace Expansion**: Expands into a b c, allowing multiple arguments to be generated from a list or sequence.

```
echo {a,b,c}
```

5. **Parameter Expansion (with default values)**: Returns default if var is unset or null. It's a way of providing default values to variables.

```
${var:-default}
```

6. **Conditional Evaluation with 1-pair brackets**: A logical evaluation operator, supporting basic tests like file conditions, string comparisons, and numeric tests.

```
[ expression ]
```

7. **Conditional Evaluation 2-pair brackets**: Logical test operator that allows for pattern matching, logical operations, and more than single square bracket tests.

```
[[ expression ]]
```

8. **String Manipulation in Parameter Expansion**: Removes the shortest matching prefix (#) or suffix (%) of a string variable.

```
${var#pattern} or ${var%pattern}
```

9. **Process Substitution**: Allows the output or input of a command to be treated as a file.

```
<(command) or >(command)
```

10. **Here-document**: Allows multi-line input to be provided directly to a command.

```
cat <<EOF
Content here
EOF
```

11. **Here-string**: Passes a string directly to the command's standard input, replacing the need for echo.

```
command <<< "string"
```

12. **Array Access**: Accesses the element at the given index of an array. arr[@] and arr[*] refer to the entire array.

```
${arr[index]}
```

13. **Positional Parameters**: Represents the command-line arguments passed to a script (e.g., $1 is the first argument).

```
$1, $2, ..., $n
```

14. **File Descriptors and Redirection**: Redirects standard error (`2`) to standard output (`1`) or sends output to `/dev/null` to discard it.

```
2>&1 or 1>/dev/null
```

15. **Loop with File Descriptors**: Reads input from a custom file descriptor (here, `3`), allowing more advanced input handling.

```
2>&1 or 1>/dev/null
```

16. **Using `eval` for Command Execution**: Executes the string contained in `$command` as a Bash command. It's dangerous if the input is untrusted.

```
eval "$command"
```

17. In-Place Editing with `sed`: Edits files in place (i.e., modifies the file directly rather than printing the output to stdout).

```
sed -i 's/foo/bar/' file.txt
```

18. **Test with `-e`, `-f`, `-d` flags**: Tests if a file exists (`-e`), is a regular file (`-f`), or is a directory (`-d`).

```
if [ -e file ]; then ... fi
```

19. `&&` **and** `||` **for Command Chaining**: Executes command2 only if command1 succeeds (`&&`), and executes command3 if command1 fails (`||`).

```
command1 && command2 || command3
```

20. **Nested Loops with for**: A nested for loop that iterates over multiple ranges or sets.

```
for i in {1..3}; do for j in {a..c}; do echo "$i$j"; done; done
```

21. `let` **for Arithmetic Operations**: Performs arithmetic operations, modifying variables in place (though `$((...))` is preferred).

```
let "a = b + c"
```

22. **Redirecting Output to Multiple Locations**: tee writes output to both standard output and a file at the same time.

```
command | tee file.txt
```

23. **Using** `shift` **to Move Positional Parameters**: Shifts the positional parameters left, essentially removing `$1` and making `$2` the new `$1`.

```
shift
```

24. `Trap for Signal Handling`: Defines a set of commands to run when a specific signal (like `SIGINT` for interrupt) is received.

```
trap 'commands' SIGINT
```

25. **`$?` to Check Last Command Exit Status**: Checks the exit status of the last command. `0` means success, `non-zero` means failure.

```
if [ $? -eq 0 ]; then ... fi
```

26. **Command Grouping with `{}`**: Groups commands and redirects their combined output to a file.

```
{ command1; command2; } > output.txt
```

27. **Pipefail to Capture All Failures**: Ensures that a pipeline returns the exit status of the last failing command rather than the last successful one.

```
set -o pipefail
```

28. **Using `declare` for Local Variables**: Declares variables with specific attributes, like `-i for integer types` or `-r for readonly`, `-a for indexed arrays`, `-A for associative array`, `-f for functions`, etc.

```
declare -i count=0
```

29. **Process Substitution for Comparing Files**: Uses process substitution to compare the output of two commands as if they were files.

```
diff <(command1) <(command2)
```

30. **Using `find` with `exec`**: Finds files and executes a command on each result. The {} is replaced by the found filename.

```
find . -name "*.txt" -exec rm {} \;
```

31. **`xargs` for Command Execution**: Passes input from stdin as arguments to a command, often used for processing output from other commands.

```
    echo "a b c" | xargs -n 1 echo
```