# Parallel Processor User Manual

Daniel Starke

SQLITE LICENSE

UNICODE DATA LICENSE

COPYRIGHT AND PERMISSION NOTICE

## Table of Contents

# 1 Introduction

The main purpose of this program is to execute programs/commands that expect input files and return output files as parallel as possible. The program uses script files to describe the commands that are to be executed to obtain the expected output files from the given input files. Dependency chains can be defined to restrict the parallel execution of these commands. It is also possible to automatically check the exit code of the executed commands.

The only way to operate this application is via console.

# 2 Getting Started

The Parallel Processor operates on files thus a basic use case could be the following example (a Linux like command-line environment is expected).

Let us imagine we have a set of images and documents and want to create a website that lists all those files which a little thumbnail. We have already prepared an application to resample images via command-line and one to create a thumbnail image from a PDF document. A frames thumbnail image is created of each previous output before the final website is created. The application which takes a list of those files to create a website out of it is also available.

The goal is to utilize our multi-core system to create the thumbnail images in parallel.

The first step would be to set up our basic script environment.

```
@enable environment-variables
@enable variable-checking
@enable command-checking
@enable full-recursive-match
@enable remove-remains

db = "pp.db"
log = "log.txt"
progress = "stderr"

img = "(?<path>images)/(?<file>.*\.(?i:jpg|png|gif))"
doc = "(?<path>documents)/(?<file>.*\.(?i:pdf))"
tmp = "temporary"
out = "thumbs"

```

Our database for remaining file deletion is “pp.db” and logs are written to “log.txt”. The current progress is shown on the standard error console.
The input images are located in images/ and are filtered by file extension for jpg, png and gif. Likewise, the documents are located in documents/ and only PDF files are taken. All temporaries are created in temporaries and the output for the thumbnails goes to thumbs.

The next step would be the definition of our file creation rules.

```
process : CreateFolders {
   none {
         destination[1] = "{tmp}/images"
         destination[2] = "{tmp}/documents"
         destination[3] = "{out}/images"
         destination[4] = "{out}/documents"
         mkdir -p "{destination[1]}"
         mkdir -p "{destination[2]}"
         mkdir -p "{destination[3]}"
         mkdir -p "{destination[4]}"
   }
}

```

```
process : CreateImageThumbnail {
   foreach ".*" {
         ~destination = "{tmp}/(?<path>{path})/(?<file>{file})"
         ResizeImage --size 120x80 --input "{?}" --output "{destination}"
   }
}

process : CreatePdfThumbnail {
   foreach ".*" {
         ~destination = "{tmp}/(?<path>{path})/(?<file>{file}.png)"
         PdfToThumbnail --width 120 --height 80 --input "{?}" --output "{destination}" --type png
   }
}

process : CreateFramedImage {
   foreach ".*" {
         destination = "(?<path>{out}/{path})/(?<file>{file})"
         DecorateImage --type frame --border-size=3 --input "{?}" --output "{destination}"
         echo "{path}/{file}" > "{destination}.src"
   }
}

process : CreateWebsite {
   all ".*" {
         ~destination[1] = "input-list.txt"
         destination[2] = "index.html"
         for file in {@*}; do echo "$file -> `cat \\"$\{file\}.src\\"`"; done > {destination[1]}
         WebsiteFromThumbs --input-list "{destination[1]}" --output "{destination[2]}"
   }
}

```

`CreateFolders` will create all necessary folders. `CreateImageThumbnail` creates the thumbnails out of the image files. `CreatePdfThumbnail` will perform the same for the PDF files using a different application. `CreateFramedImage` uses yet another application to create a thumbnail with a frame from all previously created thumbnails. And `CreateWebsite` takes in a list with all thumbnails and a link to their original file. Intermediate files marked with `~` are deleted at the end.

Finally we would define the execution rule in which order the steps shall be performed.

```
execution : default {
   CreateFolders
   > (CreateImageThumbnail("{img}") | CreatePdfThumbnail("{doc}"))
   > CreateFramedImage
   > CreateWebsite
}
```

The first step is to create all folders. The next step executes the image and PDF thumbnail creation in parallel followed by the creation of the framed thumbnail images after all thumbnails have been created. The last step takes all input files and creates the desired website.

# 3 Command-line Options

```
pp -bCf?hj?n [<target> ...] [<variable>=<value> ...]


 -b, --build
  Forces all parts to be executed.
 -C, --change-directory <directory>
  Change to this directory beforehand.
 -f, --file <file>
  Defines the path to the script file.
  The default value is "process.parallel".
 -h, --help
  Prints out this description.
 -j, --jobs <number>
  Execute with the given number of threads. It is possible to define this value
  in percent of available cores. The default is the number of virtual cores.
 --license
  Displays the licenses for this program.
 -n, --print-only
  Only prints the commands that would had been executed.
 -v, --verbosity <enumeration>
  Sets the verbosity level. Default is WARN. Setting this overwrites all
  verbosity settings in the script making this the only verbosity level.
  Possible values are ERROR, WARN, INFO and DEBUG.

target
  Execute these targets in sequence.
  The default target is "default".

variable=value
  Set these additional environment variables to the given value.
```

# 4 Syntax

## 4.1 Pragma

Pragma directives are available to change the way in which the script is parsed. They always start with an at mark `@` and there are the following types available.

1. Pragmas to enable or disable options.
2. Set the used shell.
3. To import other script files in place only once.
4. To include other script files in place.
5. To set the verbosity level.
6. To output details during pre-procession.
7. To conditionally include parts of the script.

### 4.1.1 Enable / Disable

Enabling or disabling an option will enforce the given option until the next change of it. The following options can be enabled or disabled.

1. environment-variables default: enabled
   Loads the environment variables into the global variable scope. This makes it possible to access all environment variables like any other variable. Variables given via command-line and pre-defined variables are also affected by this.
2. variable-checking default: enabled
   Forces access to missing variables to fail the program execution. Not enabling this option will only output a warning when trying to access a variable that was not defined beforehand.
3. command-checking default: enabled
   Enable this to force a check of the exit code of the programs executed. This is valid for all process blocks defined after the line in which this pragma is set. This will force the transition to abort if a program returned an exit code different from zero. Not doing so will ignore the exit codes.
4. nested-variables default: enabled
   Setting this option enables variable assignment within a process block. Not setting this option will interpret all variable assignments except the destination variable assignment in process blocks as commands.
5. full-recursive-match default: disabled
   This forces the regular expression path matching to match against every possible sub-path instead of just comparing the path elements against the regular expression. It can be used for example to find all files with a specific file extension in a given path. Due to its nature this variant is much slower than the default element-wise matching for directories with many elements.

6. remove-temporaries default: enabled
   Removes all temporaries (destination variable prefixed with ~) that got their direct dependent created successfully.
7. clean-up-incompletes default: enabled
   Automatically delete all destination files of transitions which were not completely performed (e.g. due to execution abortion) if set.
8. remove-remains default: enabled
   Removes all excessive files that are still remaining from the last execution if activated.

Examples:

```
@enable environment-variables
```

Inserts all environment variables into the global variable scope.

```
@disable environment-variables
```

Hides all environment variables from the global variable scope.

### 4.1.2 Shell

This can be used either to define a shell or to use an already defined shell. All commands following the enabled shell are executed by that shell. The predefined shell “default” is used in case no shell is enforced.

Example on how to define a shell:

```
@shell : newShell {
      path = "{ComSpec}"
      commandLine = "{ComSpec} /c {?}"
      replace = / /^ /
      raw = "true"
}
```

The here defined shell is called “newShell” for further reference and uses all possible parameters. These parameters are possible:

| **Variable Name** | **Mandatory** | **Meaning** |
|---|---|---|
| path | yes | Sets the path to the defined shell. The path will be resolved in the same way the system shell would resolve the path if no absolute path is given. Variable substitution is performed as in global variable scope. |
| commandLine | yes | Defines the command line the shell shall be called with. This includes the executable name. The variable ? is replaced with the actual command. |
| replace | no | Performs a find & replace operation on the command string before substitution of the variable ? in commandLine. Multiple replace variables can be defined by using the array syntax. In this case the replacement is done in the sorted order of the index whereas no index will be performed first. |

| Variable Name | Mandatory | Meaning |
|---|---|---|
| | | The syntax for the replace value differs from the normal variable assignment and is defined as:<br>Replace = Char, RegEx, Char, Format, Char<br>Here Char is always the same character and can be any character. RegEx is a regular expression as defined in Perl. Format is a format string as defined in Perl. |
| outputEncoding | no | Assume this character encoding for the command output. Possible encoding types are “UTF-8” and “UTF-16”. This value defaults to “UTF-8”. |
| raw | no | This parameter is only evaluated on Windows platforms. The default value is “false”. Setting this value to “true” disables special value quoting and passes the content of comamndLine as is to the shell. This might be needed to handle special command line parsings like in cmd.exe. |

Example on how to use a predefined shell:

```
@shell newShell
```

The already defined “newShell” is used from here on.

### 4.1.3 Import

Imports an other script file in place of the pragma line. This pragma expects a string value with the path to the import file. The difference to include is that imports are only included once whereas included are included as often as defined.

Note: An include can be imported again as those are two different constructs.

Example:

```
@import “path/to/import.file”
```

### 4.1.4 Include

#### 4.1.4.1 Global Scope

Include an other script file in place of the pragma line. This pragma expects a string value with the path to the include file.

Example:

```
@include “path/to/include.file”
```

#### 4.1.4.2 Process Scope

Include an other process in place of the pragma line. This pragma expects the ID of a previously defined process. All process blocks are included.

Example:

```
process : A {
      all “.*” {
            echo “Hello World!”
      }
}
process : B {
      @include A
}
```

Process `A` and `B` will perform the same commands.

#### 4.1.4.3 Execution Scope

Include all process nodes of another execution target in place of the pragma line. This pragma expects the ID of a previously defined execution target and works similar as in process scope but for process nodes in execution targets.

### 4.1.5 Verbosity

Sets a different default verbosity at pre-processor level. The command-line option overwrites this setting and defaults to what is set here or WARN if nothing was defined.

Possible values are:

- ERROR
- WARN
- INFO
- DEBUG

Example:

```
@verbosity WARN
```

Enables output of verbosity level ERROR and WARN.

### 4.1.6 Error, Warn, Info and Debug

Pass a variable name, string value or asterisk to output its value on the standard error console if the verbosity is not INFO. For INFO the output is written to the standard output console.

Examples:

```
@debug PATH
```

Outputs the content of PATH. Variable-checking is not applied to this.

```
@debug “Path = {PATH}”
```

Outputs the content of Path with the prefix “Path = “. Variable-checking is applied to this.

```
@debug *
```

Outputs all known variables within the current context. Variable-checking is not applied to this.

Use @error, @warn, @info or @debug to decide on which verbosity level this output should occur. Using @error will abort the program execution after error message output.

### 4.1.7 If Statement

An if-statement can be made on global scope or within execution blocks to control which expressions shall be included in the script and which not.

Example:

```
@if LOG is set and LOG is not file
      log = "{LOG}"
@else
      log = "default-log-file.log"
@end
```

This writes the log output to the file given by the variable LOG only if the file does not exist. It uses the given default log file in all other cases.

Note: Non-true blocks of the if-statement are not syntax checked.

See the following table for possible boolean checks. Boolean checks can be combined to a more complex boolean expression with `and` or `or` whereas `and` binds stronger than `or`. A boolean expression can be prefixed with `not` to invert its value and parentheses can be used were needed.

| Syntax | Operand | Meaning |
|---|---|---|
| "is" ["not"] \| "=" \| "!=" | set | Checks if the variable is set. |
| | file | Check if the given file exists. |
| | directory | Check if the given directory exists. |
| | regex | Check if the given string is a valid regular expression. |
| | <string literal> or <variable> | Check if the given strings are equal. |
| "is" ["not"] like \| "~" \| "!~" | <string literal> | Checks if the given string matches the regular expression provided as operand. |

If only a variable or string literal is given it will evaluate to true if the variable is set which means that a single string literal will always evaluate to true.

## 4.2 Variables

### 4.2.1 Pre-Defined Variables

Some variables are set as environment variables by the Parallel Processor before reading the specific script file. They are available like any other environment variable. See the following tables for possible variables.

| Variable Name | Meaning |
|---|---|
| PP_PATH | Full path to the Parallel Processor executable. |
| PP_VERSION | Version of the Parallel Processor. |
| PP_OS | “windows” for Windows platforms and “unix” for every other platform. |
| PP_TIME | Time when the Parallel Processor was started as HH:MM:SS. |
| PP_DATE | Date when the Parallel Processor was started as yyyy-mm-dd. |
| PP_SCRIPT | Full path to the current script file. |
| PP_TARGETS | Comma separated list of the targets requested for execution. |
| PP_THREAD | ID of the execution thread. The actual value is platform dependent. It contains only characters that are valid for file names. This variable is only defined within process blocks. |
| PP_THREADS | Number of requested execution threads. |

### 4.2.2 Special Variables

There are some variables that are treated special depending on their context. The following table illustrates these.

| Variable Name | Scope | Meaning |
|---|---|---|
| db | global | Defines the database file. By default the main script file name with the appended extension .db is used if no file is given. The database file is only created if needed. |
| log | global | Defines the log file. The standard output console will be used if this variable is not set. Set this to a writable file or use one of the following special names:<br>stdout – write to the standard output console<br>stderr – write to the standard error console<br>The path to the log file needs to be existent. The application will not attempt to create the path, only the file itself at its given location. Add a + to the beginning of the log file path to append the file rather than overwriting it. |
| progress | global | Defines the progress file. No progress output will be generated if this variable is not set. Set this to a writable file or use one of the following special names:<br>stdout – write to the standard output console<br>stderr – write to the standard error console<br>The path to the progress file needs to be existent. The application will not attempt to create the path, only the file itself at its given location. |
| progressFormat | global | Set a different format for the progress output string. See chapter Progress Format String for further details. |
| ? | process block | Refers to the given input file for each iteration. |
| * | process block | Refers to all input files separated by space. |
| @* | process block | Refers to all input files separated by spaces and each of them enclosed in quotes. |

| Variable Name | Scope | Meaning |
|---|---|---|
| destination | process block | Set this variable to define the path of the output element. |
| destination[0] | process block | Set this variable with any index to define the path of an output element. Multiple output elements can be set by using different indexes. |
| 1 | process block | Refers to the first field of the given input file for each iteration. Use a different number to access a different field. Fields may also be named and referred by its name. |

## Progress Format String

The default is defined as

`%dY-%dM-%dD %lH:%lM:%lS: %c / %t commands executed, %p%%, ETA %re\n`.

The following format tags are available:

| Tag | Replacement |
|---|---|
| **replacement prefixes** | |
| %d | last update date |
| %l | last update time stamp |
| %r | recent average |
| %g | global average |
| **direct replacements** | |
| %% | % |
| %c | current state value |
| %C | current state value formatted as human readable SI binary size |
| %t | final state value |
| %T | final state value formatted as human readable SI binary size |
| %p | percentage with leading spaces |
| %P | percentage without leading spaces |
| %q | percentage with leading spaces and once decimal comma place |
| %Q | percentage without leading spaces and once decimal comma place |
| **last update date suffixes** | |
| %dy | year part |
| %dY | year part with 4 digits |
| %dm | month part |
| %dM | month part with 2 digits |
| %dd | day of month part |
| %dD | day of month part with 2 digits |
| **last update time stamp suffixes** | |
| %lh | hours part |
| %lH | hours part with 2 digits |
| %lm | minutes part |
| %lM | minutes part with 2 digits |
| %ls | seconds part |
| %lS | seconds part with 2 digits |
| %lf | milliseconds part |
| %lF | milliseconds part with 3 digits |
| **recent and global suffixes ('x' stands for 'r' or 'g')** | |
| %xf | milliseconds part |
| %xF | total number of milliseconds (rounded down) |

| Tag | Replacement |
|---|---|
| %xs | seconds part |
| %xS | total number of seconds (rounded down) |
| %xm | minutes part |
| %xM | total number of minutes (rounded down) |
| %xh | hours part |
| %xH | total number of hours (rounded down) |
| %xd | days part (rounded down) |
| %xe | with two units precision |
| %xE | as %d %h:%m:%s.%f whereas the %d part is omitted if zero |
| %xa | average speed with dynamic time unit |
| %xA | average speed formatted as human readable SI binary size |

### 4.2.3 Removal

Using the following syntax to remove a variable from all scopes.

```
unset MyVariable
```

Multiple variables can be passed by separating them via comma.

Example:

```
unset MyVariable, AnotherVariable, More
```

### 4.2.4 Assignment

Use the following syntax to assign a string value to a variable.

```
MyVariable = “this string contains variable {x}”
```

References to other variables inside the string value are automatically resolved while script parsing. Some references may remain until program execution. These are not resolved at parse time but at run time. An example for this are references to `?` within process blocks.

Variable references can also contain functions. These functions are given by a colon operator `:` and function name. Any order and number of functions may be given. All are separated by colon operator.

Example:

```
ParentFolder = “{Folder:directory:filename}”
```

This will output `b` if Folder is set to `/a/b/c`.

### 4.2.5 Functions

Variable functions can be used when referring to the content of a variable within a string value. The functions can be given in any order and quantity after the referred variable name. All functions are separated by colon operator `:` from each other and the variable name. The following table shows which functions are available and how they operate by examples.

| Function | Input | Output | Works on `*` and `@*` |
|---|---|---|---|
| directory | /root/parent/file.ext | /root/parent | no |
| filename | /root/parent/file.ext | file.ext | no |
| file | /root/parent/file.ext | file | no |
| extension | /root/parent/file.ext | .ext | no |
| exists | /mnt | true | no |
| exists | /something-strange | false | no |
| rexists | .*\.ext | true | no |
| $/a(b)c/a<\1>c/ | abc | a<b>c | yes |
| 3,2 | 123456 | 45 | yes |
| esc | \root\parent\file.ext | \\root\\parent\\file.ext | yes |
| unix | \root\parent\file.ext | /root/parent/file.ext | yes |
| win | /root/parent/file.ext | \root\parent\file.ext | yes |
| native | \root\parent\file.ext<br>/root/parent/file.ext | /root/parent/file.ext<br>\root\parent\file.ext | yes |
| upper | file.ext | FILE.EXT | yes |
| lower | FILE.EXT | file.ext | yes |

The string replacement function has a special syntax which starts with a `$`. See the replace variable in chapter 4.1.2 which has the same format for its value like this replacement function.

The sub-string command expects one or two parameters. The first is the start position, which can be negative in which case it is subtracted from the length of the string. The second parameter is the length in characters.

The function `rexists` works similar to `exists` but match the path by regular expression rather than as it. It is affected by the pragma option full-recursive-match.

The `native` function is a platform specific alias for `unix` on Unix systems and `win` on Windows systems.

Variable functions can not be applied to variables that are made up from any variable which is not known at parse time, like `?`, `*`, `@*` or `PP_THREAD`.

### 4.2.6 Scopes

Scopes exist to handle different level of locality. The following table gives an overview of the available variable scopes and their precedence. The top-most is the most local scope which gets prioritized in case the same variable was defined in more global scopes as well. Scopes other than within process blocks are within the first global scope.

| Scope | Variable defined via |
|---|---|
| process block | assignment; initialized with the variables passed as process parameters |
| process block | named captures in process input filter |

| Scope | Variable defined via |
|---|---|
| process block | named and unnamed captures in process input file description |
| global | variable assignment |
| global | environment variables; overwritten by variable assignment on command-line<br>Note: This scope can be enabled or disabled via pragma commands. |

## 4.3 Process

Process templates can be used to define the steps that should be performed for each level. These consist of process blocks. Each process template is given an ID which can be used within an execution chain to apply the specific process template. Defining a process with an already used process ID overwrites the previously defined process. The currently defined process is used in statements accessing it.

### 4.3.1 Process Block

There are three types of process blocks, `all`, `foreach` and `none`. A process block always defines the type. For type `all` and `foreach` an input filter may be provided. This input filter uses regular expressions to define a subset of input files or a reference to a file which contains a list of input files for further procession. All files are selected if no input filter is given. It is possible to invert the meaning of the input filter by adding a `!` in front of it. A process block may consist of destination files and any number of commands. The destination file may be created based on the input file and will be passed to the next item in the process chain as defined in the execution chain. All operations defined in a single process block are executed in sequence by a single thread using the default command shell. The exit code of each command may be checked based on the variable-checking state.

Defining a destination file in the following will just pass the input file to the next process within the execution chain.

```
destination = “{?}”
```

It is possible to access capture groups from the input file description in the same way as variables. Capture groups of the input variables are made available as variables within the local scope in fact. Define the capture groups of the output path by enclosing a part in brackets `( )`. To name these capture groups use the PERL regular expression syntax for named capture groups. The same is applied to input filters, except for that unnamed captures are not propagated as variables. Defining the same capture name within the input file description and input filter gives only access to the capture group of the input filter.

Example:

```
destination = “output/path/file_(?<index>{index})(?<ext>.txt)”
```

Use the named capture group `index` to define the output file path. Make the capture groups `index` and `ext` available as input for the next process within the execution chain.

The alternative syntax ?'index' is not supported.

Additionally to this the special variable `dependency` can be defined to add a dependency file for this process block which means that every resulting transition will depend on this file. An array index can be used in the same way as for the `destination` variable to define multiple files.

Example:

```
dependency[1] = “path/to/dependency/file.ext”
```

Note that dependency variables are evaluated after destination variables.

#### 4.3.2 all Type

This type executes all commands defined once regardless of the number of input files.

#### 4.3.3 foreach Type

This type executes all commands defined for each input files. All input files are processed in parallel.

#### 4.3.4 none Type

This type executes all commands.

### 4.4 Execution

Execution targets can be defined by ID. The ID can be passed to the program to select the targets which shall be executed. Each execution target can hold any number of execution chains. Defining an execution target with an already used execution target ID overwrites the previously defined execution target. The currently defined execution target is used in statements accessing it.

An execution chain consists of process templates, execution targets and operators. These operators define whether the output of a process is needed as input for another process or if multiple processes can be run in parallel before their output is passed to another process.

A direct dependency of an execution chain is defined with the greater than operator `>`.

```
Process1 > Process2
```

Perform `Process1` and use its output as input for `Process2`. `Process2` cannot start before `Process1` finishes all tasks.

The output of many processes can be combined by the pipe operator `|`. This also instructs the program to execute all these processes in parallel.

```
(Process1 | Process 2) > Process3
```

Execute `Process1` and `Process2` in parallel and use both outputs as input for `Process3` after both processes have been finished.

All execution chains within an execution target are executed in parallel.

One or more input file masks can be passed to a process template to overwrite its input file list. These input file masks are given as regular expression.

Example:

```
Process1(“/a/.*.jpg”) > Process2
```

This will process all JPEG files within `/a/` by using process template `Process1` and uses the created output files as input for `Process2`.

It is also possible to pass variables to a process template by assigning them just in place. Alternatively it is possible to pass predefined variables.

Example:

```
Process1(var = “value”, predefined) > Process2
```

This will pass the variables `var` with the value `value` and `predefined` the to the process template `Process1`. All references to this variable will be resolved within the commands of the process template.

The capture groups of the defined regular expression are also available within the process block. This is also true for named capture groups.

Alternatively it is possible to pass the path to a file which holds a list of input files. This is done by using a `@` at the beginning of the string.

Example:

```
Process1(“@/a/filelist”) > Process2
```

This will read the files stored in `/a/filelist` line by line and passes those as input to `Process1`. Non-existing files will be ignored from the list.

Passing a `+` as first process parameter will use the previous output files as input in addition to the new dependencies provided. Not doing so will only use the new dependencies by default.

## 4.5 Log Output Format

The log output format depends on whether the script was actually executed or the planned actions were just printed using the command-line argument `-n`. Both forms list the commands within the processes in the order in which they were called within the execution target. The command is prefixed with the execution flags which are coded in the following way.

| Flag | Name | Description |
|---|---|---|
| F | Forced | Execution of this command was forced. This can be done on command-line (`-b`) or within the script. |
| M | Missing | The output is missing and needs to be created by this command. |
| C | Changed | An input dependency changed and the output needs to be re-created. |

Additionally to this, the log output format after script execution also contains the start and stop time of each command in UTC, the execution duration and the exit code.

## 4.6 Extended Backus–Naur Form

Syntax description according to ISO/IEC 14977.

```
NonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

Digit = "0" | NonZeroDigit ;

LowerLetter = "a" | "b" | "c" | "d" | "e" | "f" | "g"  "h" | "i" | "j" | "k" |
"l" | "m" | "n"  "o" | "p" | q" | "r" | "s" | "t" | "u"  "v" | "w" | "x" | "y" |
"z" ;

UpperLetter = "A" | "B" | "C" | "D" | "E" | "F" | "G"  "H" | "I" | "J" | "K" |
"L" | "M" | "N"  "O" | "P" | Q" | "R" | "S" | "T" | "U"  "V" | "W" | "X" | "Y" |
"Z" ;

IdString = ("_" | LowerLetter | UpperLetter), { "_" | LowerLetter | UpperLetter
| Digit } ;

ValueString = ('"', { Character - EndOfLine }, '"') | ("'", { Character -
EndOfLine }, "'") ;

NonEmptyValueString = ('"', Character, { Character - EndOfLine }, '"')
| ("'", Character, { Character - EndOfLine }, "'") ;

VariableRemoval = "unset", IdString, { ",", IdString } ;

VariableAssignment = IdString, "=", ValueString ;

Pragma = "@", (
(("enable" | "disable"), ("environment-variables" | "variable-checking" |
"command-checking" | "full-recursive-match" | "remove-temporaries" | "clean-up-
incompletes" | "remove-remains"))
| ("shell", (IdString | (":", IdString, "{", { VariableAssignment }, "}")))
| ("verbosity", ("ERROR" | "WARN" | "INFO" | "DEBUG"))
| (("error" | "warn" | "info" | "debug"), (IdString | ValueString | "*"))
| ("import", NonEmptyValueString) )
| ("include", NonEmptyValueString) ) ;

BooleanCheck = (IdString | ValueString), [ ((("is", [ "not" ]) | ("=" | "!=")),
("set" | ValueString | "file" | "directory" | "regex" | "true" | "false" |
IdString)) | ((("is", [ "not" ], "like") | ("~" | "!~")), ValueString) ] ;

BooleanAnd = ("and" | "&&"), BooleanTerm ;

BooleanAndOrTerm = BooleanTerm, { BooleanAnd } ;

BooleanOr = ("or" | "||"), BooleanAndOrTerm ;

BooleanTerm = [ "not" | "!" ], ("(", BooleanExpression, ")") | BooleanCheck ;

BooleanExpression = BooleanAndOrTerm, { BooleanOr } ;
```

```
PragmaIfBlock = Grammar ;

PragmaIf = "@if", BooleanExpression, PragmaIfBlock, { "@else", "if",
BooleanExpression, PragmaIfBlock }, [ "@else", BooleanExpression,
PragmaIfBlock ], "@end" ;

ExecutionPragmaIfBlock = { ExecutionExpression } ;

ExecutionPragmaIf = "@if", BooleanExpression, ExecutionPragmaIfBlock, { "@else",
"if", BooleanExpression, ExecutionPragmaIfBlock }, [ "@else", BooleanExpression,
ExecutionPragmaIfBlock ], "@end" ;

InputFileFilter = [ "!" ], NonEmptyValueString ;

DestinationVariable = [ "~" ], "destination", [ "[", ("0" | (NonZeroDigit,
{Digit})), "]" ], "=", ValueString ;

DependencyVariable = "dependency", [ "[", ("0" | (NonZeroDigit, {Digit})),
"]" ], "=", ValueString ;

CommandLineStart = { Character - ("\", EndOfLine) } ;

CommandLineEnd = "\", EndOfLine ;

Command = CommandLineStart, { CommandLineEnd, CommandLineStart } ;

ProcessBlock = ((("foreach" | "all"), [ InputFileFilter ]) | "none"), "{",
{ DestinationVariable | DependencyVariable | VariableAssignment | Command }, "}"
;

ProcessPragamInclude = "@include", IdString ;

Process = "process", ":", IdString, "{", {ProcessBlock | ProcessPragamInclude },
"}" ;

ProcessParameterList = "+" | ([ "+", "," ], (IdString  | VariableAssignment |
NonEmptyValueString), { ",", (IdString  | VariableAssignment |
NonEmptyValueString) });

ExecutionElement = [ "!" ], IdString ;

ProcessElement = [ "!" ], IdString, [ "(", ProcessParameterList, ")" ] ;

Parallel = Term, "|", Term, { "|", Term } ;

Dependencies = Term, ">", Term, { ">", Term } ;

Term = ("(", Expression, ")") | ExecutionElement | ProcessElement ;

ExecutionPragamInclude = "@include", IdString ;
```

```
Expression = Dependencies | Parallel | Term ;

ExecutionExpression =  Expression | Term | ExecutionPragamInclude |
ExecutionPragmaIf ;


Execution = "execution", ":", IdString, "{", { ExecutionExpression }, "}" ;


Grammar = { Pragma | PragmaIf | VariableAssignment | VariableRemoval | Process |
Execution } ;
```