Preprocessor

Table of Contents

1. Concept

The Red preprocessor is a dialect of Red, enabling transformation of Red source code using a specific layer on top of regular Red language. Transformations are achieved by inlining preprocessor keywords (called `directives`) inside Red source code. These directives will be processed:

- when the Red source code is compiled
- when the Red source code is runned by `do` function with a file argument
- when `expand-directives` function is called on a block value

The preprocessor is invoked after the LOADing phase, so it processes Red values, and not the source code in text form.

Directive categories:

- conditional directives: include code depending on the result of an expression.
- control directives: control the behavior of the preprocessor.
- macros: transform code using functions, enables more complex transformations.

Directives are denoted by specific `issue!` values (starting with a `#` symbol).

When a directive is processed, it is replaced by the resulting value it returns (some directives do not return anything, so they are just removed). That is how transformations of source code is achieved.

NOTE Red/System has its own preprocessor, which is similar, but low-level and applied to the source code in text form.

1.1. Config object

In conditional expressions and in macros, an implicit `config` object is provided, to give access to the settings used to compile the source code. Those settings are often used for conditional inclusion of code. The exhaustive list of settings can be found here.

Example:

```
#if config/OS = 'Windows [#include %windows.red]
```

NOTE When using the preprocessor at runtime (from Red's interpreter), the `config` object is also available, and will reflect the options used for compiling the Red executable currently used to run the code.

1.2. Execution context

All the expressions used in directives are bound to a dedicated context to avoid leaking words in global context, causing unwanted side-effects. That context is extended with every set-word used in conditional expressions, macros and `#do` directives.

TIP

- It is possible to print out the content of that hidden context using the following expression:

```
#do [probe self]
```

- When used at runtime, the hidden context can also be accessed directly using:

```
probe preprocessor/exec
```

1.3. Implementation note

Currently, expressions in directives used at compile-time are evaluated using Rebol2 interpreter, as it is running the toolchain code. This is temporary and should be switched to a Red evaluator as soon as possible. In the meantime, ensure that your expressions and macros code can be run with Red too, for compatibility with Red interpreter at run-time (and at compile-time in the future).

2. Macros

The Red preprocessor supports defininig macro functions (called just **macros**) for achieving more complex transformations. Macros allow an efficient form of metaprogramming, where the execution cost is paid at compile-time, rather than run-time. Red already has strong metaprogramming abilities at run-time, but, for the sake of enabling Red source code to be run equally well by the compiler and interpreter, macros can also be resolved at run-time.

NOTE	There are no read-time (reader) macros. Given how simple it is already to preprocess sources in text form using Parse dialect, such support would be, for now, redundant.

The preprocessor supports two types of macros:

2.1. Named macros

This type of macro is higher-level, as the preprocessor takes care of arguments fetching and returned value replacing for the user. The typical form is:

```
#macro name: func [arg1 arg2... /local word1 word2...][...code...]
```

After defining such macro, each time the `name` word is encountered in the source code, it will trigger this macro following those steps:

1. Fetch the argument values.
2. Invoke the macro function with the arguments.
3. Replace the macro call and its arguments with the last value of the function.
4. Resume preprocessing from the replaced value (allowing recursive macro evaluation).

NOTE	Specifying types for the arguments is not currently supported.

Examples:

```
Red []
#macro make-KB: func [n][n * 1024]
print make-KB 64
```

will result in:

```
Red []
print 65536
```

Calling other macros, from within a macro:

```
Red []
#macro make-KB: func [n][n * 1024]
#macro make-MB: func [n][make-KB make-KB n]

print make-MB 1
```

will result in:

```
Red []
print 1048576
```

2.2. Pattern-matching macros

Instead of matching a word and fetching argument, this type of macros matches a pattern provided as a Parse dialect rule or keyword. Like for the named macros, the returned value is used as replacement for the matched pattern.

Though, there is also a lower-level version of this type of macros, which is triggered by the usage of the `[manual]` attribute. In such case, there are no implicit actions, but full control is given to the user. No automatic replacement takes place, it is up to the macro function to apply the desired transformations and set the resuming point of the processing.

The typical form of pattern-matching macros is:

```
#macro <rule> func [<attribute> start end /local word1 word2...][...code...]
```

The `<rule>` part can be:

- a `lit-word!` value: for matching a specfic word.
- a `word!` value: a Parse keyword, like a datatype name or `skip` for matching **all** values.
- a `block!` value: a Parse dialect rule.

`start` and `end` arguments are references delimiting the matched pattern in the source code. The return value needs to be a reference to the resuming position.

`<attribute>` can be `[manual]`, which triggers the low-level manual mode for the macro.

Examples:

```
Red []

#macro integer! func [s e][s/1 + 1]
print 1 + 2
```

will result in:

```
Red []
print 2 + 3
```

Using **manual** mode, the same macro would be written as:

```
Red []

#macro integer! func [[manual] s e][s/1: s/1 + 1 next s]
print 1 + 2
```

Using a block rule to create a variable-arity function:

```
Red []
#macro ['max some [integer!]] func [s e][
    first maximum-of copy/part next s e
]
print max 4 2 3 8 1
```

will result in:

```
Red []
print 8
```

3. Directives

3.1. #if

Syntax

```
#if <expr> [<body>]

<expr> : expression whose last value will be used as a condition.
<body> : code to be included if <expr> is true.
```

Description

Include a block of code if the conditional expression is true. If the `<body>` block is included, it will be also passed to the preprocessor.

Examples

```
Red []

#if config/OS = 'Windows [print "OS is Windows"]
```

will result in the following code if run on Windows:

```
Red []

print "OS is Windows"
```

and otherwise, will result in just:

```
Red []
```

It is also possible to define your own words using `#do` directive, which can be used in conditional expressions later:

```
Red []

#do [debug?: yes]

#if debug? [print "running in debug mode"]
```

will result in:

```
Red []

print "running in debug mode"
```

3.2. #either

Syntax

```
#either <expr> [<true>][<false>]

<expr>  : expression whose last value will be used as a condition.
<true>  : code to be included if <expr> is true.
<false> : code to be included if <expr> is false.
```

Description

Choose a block of code to include depending on a conditional expression. The included block will be also passed to the preprocessor.

Example

```
Red []

print #either config/OS = 'Windows ["Windows"]["Unix"]
```

will result in the following code if run on Windows:

```
Red []

print "Windows"
```

and otherwise, will result in:

```
Red []

print "Unix"
```

3.3. #switch

Syntax

```
#switch <expr> [<value1> [<case1>] <value2> [<case2>] ...]
#switch <expr> [<value1> [<case1>] <value2> [<case2>] ... #default [<default>]]

<valueN>  : value to match.
<caseN>   : code to be included if last tested value matched.
<default> : code to be included if no other value matched.
```

Description

Choose a block of code to include among several choices, depending on a value. The included block will be also passed to the preprocessor.

Example

```
Red []

print #switch config/OS [
    Windows ["Windows"]
    Linux   ["Unix"]
    macOS   ["macOS"]
]
```

will result in the following code if run on Windows:

```
Red []

print "Windows"
```

3.4. #case

Syntax

```
#case [<expr1> [<case1>] <expr2> [<case2>] ...]

<exprN> : conditional expression.
<caseN> : code to be included if last conditional expression was true.
```

Description

Choose a block of code to include among several choices, depending on a value. The included block will be also passed to the preprocessor.

Example

```
Red []

#do [level: 2]

print #case [
    level = 1  ["Easy"]
    level >= 2 ["Medium"]
    level >= 4 ["Hard"]
]
```

will result in:

```
Red []

print "Medium"
```

3.5. #include

Syntax

```
#include <file>

<file> : Red file to be included (file!).
```

Description

When evaluated at compile-time, read and include the argument file contents at the current position. The file can contain a path, absolute or relative to the current script. Directives from included file cannot see values set in directives set in including file:

```
Red [File: %main.red]
#do [a: true]
#include %incl.red
```

```
Red [File: %incl.red]
#either a [print "a"] [print "not a"]   ;-- this will result in "a has no value"
preprocessor error.
```

When run by the Red interpreter, this directive is just replaced by a `do`, and no file inclusion occurs.

3.6. #do

Syntax

```
#do [<body>]
#do keep [<body>]

<body> : any Red code.
```

Description

Evaluate the body block in the hidden execution context. If `keep` is used, replace the directive and argument with the result of evaluating `body`.

Example

```
Red []

#do [a: 1]

print ["2 + 3 =" #do keep [2 + 3]]

#if a < 0 [print "negative"]
```

will result in:

```
Red []

print ["2 + 3 =" 5]
```

3.7. #macro

Syntax

```
#macro <name> func <spec> <body>
#macro <pattern> func <spec> <body>

<name>    : name of the macro function (set-word!).
<pattern> : matching rule for triggering the macro (block!, word!, lit-word!).
<spec>    : specification block for the macro function.
<body>    : body block of the macro function.
```

Description

Create a macro function.

For a named macro, the specification block can declare as many arguments as needed. The body needs to return a value that will be used to replace the macro call and its arguments. Returning an empty block will just remove the macro call and its arguments.

For a pattern-matching macro, the specification block must declare only **two** arguments, the starting reference and ending reference of the matched pattern. By convention, the arguments names are: `func [start end]` or `func [s e]` as short form. By default, the body needs to return a value that will be used to replace the matched pattern. Returning an empty block will just remove the matched pattern.

A **manual** mode is also available for pattern-matching macros. It can be set by putting a `[manual]` attribute in the function's **spec** block: `func [[manual] start end]`. Such manual mode requires the macro to return the resuming position (instead of a replacement value). If it needs to **reprocess** a replaced pattern, then `start` is the value to return. If it needs to **skip** the matched pattern, then `end` is the value to return. Other positions can also be returned, depending on the transformation achieved by the macro, and the desire to partially or fully reprocess the replaced value(s).

A pattern-matching macro accepts:

- a block: specifies a pattern to match using the Parse dialect.
- a word: specifies a valid Parse dialect word (like a datatype name, or `skip` to match all values).
- a lit-word: specifies a specific literal word to match.

Examples

```
Red []
#macro pow2: func [n][to integer! n ** 2]
print pow2 10
print pow2 3 + pow2 4 = pow2 5
```

will result in:

```
Red []
print 100
print 9 + 16 = 25
```

Pattern-matching macro example:

```
Red []
#macro [number! '+ number! '= number!] func [s e][
    do copy/part s e
]

print 9 + 16 = 25
```

will result in:

```
Red []
print true
```

A pattern-matching macro in manual mode:

```
Red []
#macro ['sqrt number!] func [[manual] s e][
    if negative? s/2 [
        print [
            "*** SQRT Error: no negative number allowed" lf
            "*** At:" copy/part s e
        ]
        halt
    ]
    e             ;-- returns position passed the matched pattern
]

print sqrt 9
print sqrt -4
```

will result in:

```
*** SQRT Error: no negative number allowed
*** At: sqrt -4
(halted)
```

3.8. #local

Syntax

```
#local [<body>]

<body> : arbitrary Red code containing local macros definitions.
```

Description

Create a local context for macros. All macros defined in that context will be discarded on exit. Therefore, the local macros also need to be locally applied. This directive can be used recursively (`#local` is a valid directive in `<body>`).

Example

```
Red []
print 1.0
#local [
    #macro float! func [s e][to integer! s/1]
    print [1.23 2.54 123.789]
]
print 2.0
```

will result in:

```
Red []
print 1.0
print [1 3 124]
print 2.0
```

3.9. #reset

Syntax

```
#reset
```

Description

Reset the hidden context, emptying it from all previously defined words and removing all previously defined macros.

3.10. #process

Syntax

```
#process [on | off]
```

Description

Enable or disable the preprocessor (it is enabled by default). This is an escape mechanism to avoid processing parts of Red files where directives are used literally and not meant for the preprocessor (for example, if used in a dialect with a different meaning).

Implementation constraint: when enabling the preprocessor again after disabling it earlier, the `#process off` directive needs to be at same (or higher) level of nesting in the code.

Example

```
Red []

print "Conditional directives:"
#process off
foreach d [#if #either #switch #case][probe d]
#process on
```

will result in:

```
Red []

print "Conditional directives:"
foreach d [#if #either #switch #case][probe d]
```

3.11. #trace

Syntax

```
#trace [on | off]
```

Description

Enable or disable the debugging output of evaluated expressions and macros on screen. There are no specific constraints on where this directive can be used in the Red sources.

4. Runtime API

The Red preprocessor can also work at run-time, in order to be able to evaluate source code using preprocessor directives also from the interpreter. It will be invoked automatically when using `do` on a `file!` value. Note that the following form can be used to `do` a file without invoking the preprocessor: `do load %file`.

4.1. expand-directives

Syntax

```
expand-directives [<body>]
expand-directives/clean [<body>]

<body> : arbitrary Red code containing preprocessor directives.
```

Description

Invoke the preprocessor on a block value. The argument block will be modified and used as returned value. If `/clean` refinement is used, the preprocessor state is reset, so all the macros previously defined are erased.

Example

```
expand-directives [print #either config/OS = 'Windows ["Windows"]["Unix"]]
```

will return on Windows platform:

```
[print "Windows"]
```