Préprocesseur

Table of Contents

1. Concept

Le préprocesseur Red est un dialecte de Red, permettant la transformation du code source en utilisant une surcouche spécifique du langage Red habituel. Les transformations sont accomplies en insérant des mots-clés du préprocesseur (appelés `directives`) au sein du code source Red. Ces directives seront traitées:

- quand le code source Red est compilé,
- quand le code source Red est exécuté par la fonction `do` avec un fichier en argument,
- quand la fonction `expand-directives` est appelée sur une valeur de type bloc.

Le préprocesseur est invoqué après la phase de chargement (par LOAD), si bien qu'il traite les valeurs Red, et non le code source sous forme textuelle.

Catégories de directives:

- directives conditionnelles: incluent du code suivant le résultat d'une expression.
- directives de contrôle: contrôlent le comportement du préprocesseur.
- macros: transforment le code en utilisant des fonctions, permettant des transformations plus complexes.

Les directives sont identifiées par des valeurs spécifiques de type `issue!` (débutant par le symbole `#`).

Quand une directive est traitée, elle est remplacée par la valeur de retour qu'elle renvoie (certaines directives ne renvoient rien, et elles sont juste supprimées). C'est ainsi que se fait la transformation du code source.

NOTE	Red/System a son propre préprocesseur, qui est similaire, mais de bas-niveau et appliqué au code source sous forme textuelle.

1.1. Objet config

Dans les expressions conditionnelles et dans les macros, un objet `config` implicite est fourni, pour donner accès aux paramètres utilisés pour compiler le code source. Ces paramètres sont souvent utilisés pour l'inclusion conditionnelle de code. La liste exhaustive des paramètres se trouve ici.

Exemple:

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

NOTE	Lorsqu'on utilise le préprocesseur au moment de l'exécution (à partir de l'interpréteur de Red), l'objet `config` est également disponible, et reflétera les options utilisées pour compiler l'exécutable Red en cours d'utilisation.

1.2. Contexte d'exécution

Toutes les expressions utilisées dans les directives sont liées à un contexte dédié pour éviter la fuite de mots dans le contexte global, qui causerait des effets de bord non désirés. Ce contexte est déployé avec chaque set-word utilisé dans les expressions conditionnelles, les macros et les directives `#do`.

TIP

- Il est possible d'afficher le contenu de ce contexte caché en utilisant l'expression suivante:

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

- Lorsqu'il est utilisé au moment de l'exécution, le contexte caché peut aussi être interrogé directement en utilisant:

```
probe preprocessor/exec
```

1.3. Note d'implémentation

Actuellement, les expressions dans les directives utilisées au moment de la compilation sont évaluées en utilisant l'interpréteur Rebol2, qui exécute le code de la chaîne d'outils. Cela est temporaire et le basculement vers un évaluateur Red devrait se faire dès que possible. Dans l'intervalle, assurez-vous que le code de vos expressions et macros peut être exécuté avec Red aussi, pour la compatibilité avec l'interpréteur Red lors de l'exécution. (et de la compilation à l'avenir).

2. Macros

Le préprocesseur Red supporte la définition de fonctions macros (appelées simplement **macros**) pour accomplir des transformations plus complexes. Les macros permettent une forme efficace de métaprogrammation, où le coût d'exécution est supporté au moment de la compilation, plutôt qu'à l'exécution. Red a également de fortes capacités de métaprogrammation à l'exécution, mais, dans le but de permettre au code source Red de tourner aussi bien avec le compilateur qu'avec l'interpréteur, les macros peuvent également être résolues à l'exécution.

NOTE Il n'y a pas de macros pour la phase de lecture (reader). Etant donné qu'il est déjà si simple de prétraiter des sources sous forme de texte en utilisant le dialecte Parse, un tel support serait actuellement redondant.

Le préprocesseur supporte deux types de macros:

2.1. Macros nommées

Ce type de macro est de plus haut-niveau, et le préprocesseur se charge d'aller chercher les arguments et de remplacer la valeur renvoyée pour l'utilisateur. La forme typique est:

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

Après la définition d'une telle macro, chaque fois que le mot `name` est rencontré dans le code source, il déclenchera cette macro en respectant les étapes suivantes:

1. Recherche des valeurs des arguments.
2. Invocation de la fonction macro avec les arguments.
3. Remplacement de l'appel de macro et de ses arguments par la dernière valeur de la fonction.
4. Reprise du prétraitement à partir de la valeur remplacée (ce qui permet l'évaluation d'une macro récursive).

NOTE	Il n'est pas possible actuellement de spécifier des types pour les arguments.

Exemples:

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

résultera en:

```
Red []
print 65536
```

Appel d'autres macros, depuis l'intérieur d'une macro:

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

print make-MB 1
```

résultera en:

```
Red []
print 1048576
```

2.2. Macros à reconnaissance de motif

Au lieu de reconnaître un mot et d'aller chercher un argument, ce type de macros reconnaît un motif fourni sous forme de règle du dialecte Parse ou de mot-clé. Comme pour les macros nommées, la valeur renvoyée est utilisée à la place du motif reconnu.

Cependant, il y a aussi une version bas-niveau de ce type de macros, qui est déclenchée par l'usage de l'attribut `[manual]`. Dans ce cas, il n'y a pas d'actions implicites, mais le contrôle total est donné à l'utilisateur. Aucun remplacement automatique n'a lieu, c'est à la fonction macro d'appliquer les transformations désirées et de fixer le point de reprise du traitement.

Cette forme typique de macro à reconnaissance de motif est:

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

La partie `<rule>` peut être:

- une valeur de type `lit-word!`: pour reconnaître un mot spécifique.
- une valeur de type `word!`: un mot-clé de Parse, comme un nom de type ou `skip` pour accepter **toutes** les valeurs.
- une valeur de type `block!`: une règle du dialecte Parse.

Les arguments `start` et `end` sont des références délimitant le motif reconnu dans le code source. La valeur renvoyée doit être une référence à la position de reprise.

`<attribute>` peut être `[manual]`, qui déclenche le mode manuel de bas-niveau pour la macro.

Exemples:

```
Red []

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

résultera en:

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

En utilisant le mode **manual** la même macro s'écrirait:

```
Red []

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

Utilisation d'une règle bloc pour créer une fonction d'arité variable:

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

résultera en:

```
Red []
print 8
```

3. Directives

3.1. #if

Syntaxe

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

<expr> : expression dont la dernière valeur sera utilisée comme condition.
<body> : code à inclure si <expr> est vraie.
```

Description

Inclut un bloc de code si l'expression conditionnelle est vraie. Si le bloc `<body>` est inclus, il sera également passé au préprocesseur.

Exemples

```
Red []

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

résultera dans le code suivant en cas d'exécution sous Windows:

```
Red []

print "L'OS est Windows"
```

et dans le cas contraire, résultera juste en:

```
Red []
```

Il est également possible de définir vos propres mots en utilisant la directive `#do`, qui peut être utilisée plus tard dans des expressions conditionnelles:

```
Red []

#do [debug?: yes]

#if debug? [print "exécution en mode de débogage"]
```

résultera en:

```
Red []

print "exécution en mode de débogage"
```

3.2. #either

Syntaxe

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

<expr>  : expression dont la dernière valeur sera utilisée comme condition.
<true>  : code à inclure si <expr> est vraie.
<false> : code à inclure si <expr> est fausse.
```

Description

Choisit un bloc de code à inclure suivant une expression conditionnelle. Le bloc inclus sera aussi passé au préprocesseur.

Exemple

```
Red []

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

résultera dans le code suivant en cas d'exécution sous Windows:

```
Red []

print "Windows"
```

et dans le cas contraire, résultera en:

```
Red []

print "Unix"
```

3.3. #switch

Syntaxe

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

<valueN>  : valeur à reconnaître.
<caseN>   : code à inclure si la dernière valeur testée a été reconnue.
<default> : code à inclure si aucune autre valeur n'a été reconnue.
```

Description

Choisit un bloc de code à inclure parmi différents choix, en fonction d'une valeur. Le bloc inclus sera aussi passé au préprocesseur.

Exemple

```
Red []

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

résultera dans le code suivant en cas d'exécution sous Windows:

```
Red []

print "Windows"
```

3.4. #case

Syntaxe

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

<exprN> : expression conditionnelle.
<caseN> : code à inclure si la dernière expression conditionnelle était vraie.
```

Description

Choisit un bloc de code à inclure parmi différents choix, en fonction d'une valeur. Le bloc inclus sera aussi passé au préprocesseur.

Exemple

```
Red []

#do [level: 2]

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

résultera en:

```
Red []

print "Medium"
```

3.5. #include

Syntaxe

```
#include <file>

<file> : Fichier Red à inclure (file!).
```

Description

Lors de l'évaluation au moment de la compilation, lit et inclut le contenu du fichier argument à la position courante. Le fichier peut contenir un chemin, absolu ou relatif au script courant. Les directives du fichier inclus ne penvent pas voir les valeurs définies dans les fichiers qui l'incluent:

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

```
Red [File: %incl.red]
#either a [print "a"] [print "not a"]   ;-- ceci provoquera l'erreur de préprocesseur
"a has no value".
```

Lors de l'évaluation par l'interpréteur Red, cette directive est juste remplacée par un `do`, et aucune inclusion de fichier n'a lieu.

3.6. #do

Syntaxe

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

<body> : tout code Red.
```

Description

Evalue le bloc corps (`body`) dans le contexte d'exécution caché. Si `keep` est utilisé, remplace la directive et l'argument par le résultat de l'évaluation de `body`.

Exemple

```
Red []

#do [a: 1]

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

#if a < 0 [print "négatif"]
```

résultera en:

```
Red []

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

3.7. #macro

Syntaxe

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

<name>    : nom de la fonction macro (set-word!).
<pattern> : règle de reconnaissance pour le déclenchement de la macro (block!, word!,
lit-word!).
<spec>    : bloc de spécifications pour la fonction macro.
<body>    : bloc corps de la fonction macro.
```

Description

Crée une fonction macro.

Pour une macro nommée, le bloc de spécifications peut déclarer autant d'arguments que nécessaire. Le corps doit renvoyer une valeur qui sera utilisée pour remplacer l'appel de la macro et ses arguments. Le renvoi d'un bloc vide supprimera juste l'appel de la macro et ses arguments.

Pour une macro à reconnaissance de motif, le bloc de spécification doit déclarer seulement **deux** arguments, la référence de début et la référence de fin du motif reconnu. Par convention, les arguments sont nommés: `func [start end]` ou `func [s e]` en forme abrégée. Par défaut, le corps doit renvoyer une valeur qui sera utilisée pour remplacer le motif reconnu. Le renvoi d'un bloc vide supprimera juste le motif reconnu.

Un mode **manual** est également disponible pour les macros à reconnaissance de motif. Il peut être activé en mettant un attribut `[manual]` dans le bloc **spec** de la fonction: `func [[manual] start end]`. Un tel mode manuel requiert que la macro renvoie la position de reprise (au lieu d'une valeur de remplacement). Si elle doit **retraiter** un motif remplacé, alors la valeur à renvoyer est `start`. Si elle doit **sauter** le motif reconnu, alors la valeur à renvoyer est `end`. D'autres positions peuvent également être renvoyées, suivant la transformation effectuée par la macro, et le souhait de retraiter partiellement ou totalement la ou les valeurs remplacée(s).

Une macro à reconnaissance de motif accepte:

- un bloc: spécifie un motif à reconnaître en utilisant le dialecte Parse.
- un word: spécifie un mot valide du dialecte Parse (comme un nom de type de données, ou `skip` pour accepter toutes les valeurs).
- un lit-word: spécifie a un mot littéral particulier à reconnaître.

Exemples

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

résultera en:

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

Exemple de macro à reconnaissance de motif:

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

print 9 + 16 = 25
```

résultera en:

```
Red []
print true
```

Une macro à reconnaissance de motif en mode manuel:

```
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             ;-- renvoie la position après le motif reconnu
]

print sqrt 9
print sqrt -4
```

résultera en:

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

3.8. #local

Syntaxe

```
#local [<body>]

<body> : code Red quelconque contenant des définitions de macros locales.
```

Description

Crée un contexte local pour des macros. Toutes les macros définies dans ce contexte seront jetées à l'issue. Par conséquent, les macros locales doivent aussi être appliquées localement. Cette directive peut être utilisée récursivement. (`#local` est une directive valide dans `<body>`).

Exemple

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

résultera en:

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

3.9. #reset

Syntaxe

```
#reset
```

Description

Réinitialise le contexte caché, le vidant de tous les mots précédemments définis et supprimant

toutes les macros précédemment définies.

3.10. #process

Syntaxe

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

Description

Active ou désactive le préprocesseur (il est activé par défaut). Ceci est un mécanisme d'échappement pour éviter de traiter des parties de fichiers Red où les directives sont utilisées littéralement et ne sont pas destinées au préprocesseur (par exemple, si elles sont utilisées dans un dialecte avec un sens différent).

Contrainte d'implémentation: lorsqu'on active de nouveau le préprocesseur après l'avoir désactivé auparavant, la directive `#process off` doit être au même niveau (ou à un niveau supérieur) dans l'imbrication du code.

Exemple

```
Red []

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

résultera en:

```
Red []

print "Directives conditionnelles:"
foreach d [#if #either #switch #case][probe d]
```

3.11. #trace

Syntaxe

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

Description

Active ou désactive la sortie de débogage à l'écran des expressions évaluées et des macros. Il n'y a pas de contraintes spécifiques sur le lieu où cette directive peut être utilisée dans les sources Red.

4. API d'exécution

Le préprocesseur peut également travailler au moment de l'exécution, ce qui permet d'évaluer du code source en utilisant les directives du préprocesseur aussi à partir de l'interpréteur. Il sera invoqué automatiquement lorsqu'on utilise `do` sur une valeur de type `file!`. Notez que la forme suivante peut être utilisée pour effectuer `do` sur un fichier sans invoquer le préprocesseur: `do load %file`.

4.1. expand-directives

Syntaxe

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

<body> : code Red quelconque contenant des directives du préprocesseur.
```

Description

Invoque le préprocesseur sur une valeur de type bloc. Le bloc argument sera modifié et utilisé comme valeur renvoyée. Si le raffinement `/clean` est utilisé, l'état du préprocesseur est réinitialisé, si bien que toutes les macros précédemment définies sont effacées.

Exemple

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

renverra sur la plateforme Windows:

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