Preprocessor

Table of Contents

1. Koncept

Preprocesor je dialekt Redu, umožňující transformaci zdrojového kódu s použitím specielní vrstvy nad normálním jazykem. Transformací je dosahováno vkládáním klíčových slov preprocesoru (zvaných `directives`) do zdrojového kódu. Tato direktiva jsou zpracována když je:

- zdrojový kód kompilován
- zdrojový kód spuštěn funkcí `do`, jejímž argumentem je soubor
- volána funkce `expand-directives` pro blokovou hodnotu

Preprocesor je invokován po fázi LOAD, takže zpracovává hodnoty Redu a nikoli zdrojový kód v textové formě.

Kategorie direktiv:

- podmínková direktiva: vloží kód v závislosti na výsledku výrazu.

- řídící direktiva: řídí chování preprocesoru.
- makra: transformují kód s použitím funkcí; umožňují složitější transformace.

Direktiva jsou označena specifickými hodotami typu `issue!` (počínaje symbolem `#`).

Je-li direktivum prováděno, je nahrazeno jeho výslednou vratnou hodnotou (některá direktiva nevrací nic, jsou tedy pouze odstraněna). Takto je postupně transformován celý zdrojový soubor.

NOTE	Red/System má svůj vlastní preprocesor, jenž je podobný ale `low-level` a je aplikován na zdrojový kód v textové formě.

1.1. Konfigurační objekt

V podmínkových výrazech a makrech se používá implicitní objekt `config`, umožňující přístup k nastavením, použitým pro kompilaci zdrojového kódu. Tato nastavení se často používají pro podmínečnou inkluzi kódu. Vyčerpávající seznam nastavení lze nalézt zde.

Example:

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

NOTE	Při použití preprocesoru za běhu programu (at runtime - z interpreta Redu) je rovněž dostupný objekt `config`, jenž reflektuje volby, použité pro kompilaci prováděčky Redu, aktuálně použité pro běh kódu.

1.2. Prováděcí kontext

Všechny výrazy, použité v direktivách, jsou vázány na dedikovaný kontext aby se předešlo upouštění (leaking) slov v globálním kontextu, jež může mít nechtěné vedlejší účinky. Tento kontext je rozšířen o každé zadané slovo (set-word), použité v podmínkových výrazech, makrech a v direktivě `#do`.

TIP	• Obsah takového skrytého kontextu je možné vytisknout tímto výrazem:
	`#do [probe self]`
	• Při použití při běhu programu (runtime) lze ke skrytému kontextu přistoupit přímo použitím výrazu:
	`probe preprocessor/exec`

1.3. Implementační poznámka

V současné době jsou výrazy v direktivách, použitých při kompilaci, vyhodnocovány interpretem Rebol2, neboť ten spouští "toolchain code". Toto je dočasné a mělo by být co nejdříve převedeno na vyhodnocování přímo Redem.

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. Makra

Preprocesor Redu podporuje definování maker pro realizaci složitějších transformací. Makra umožňují účinou formu metaprogramování, kdy se náklady na provedení uplatňují spíše při kompilaci než při běhu programu. Red již vykazuje silné metaprogramovací schopnosti při běhu programu. Aby však bylo možné zpracovávat zdrojový kód stejně dobře kompilátorem i interpretem, mohou být makra řešena také při běhu programu.

NOTE	Neexistují žádná "read-time (reader)" makra. Vzhledem k tomu, jak jednoduché již je předzpracovávat zdroje v textové formě s použitím dialektu Parse, byla by taková podpora (aktuálně) maker nadbytečná.

Preprocesor podporuje dva typy maker:

2.1. Pojmenovaná makra

Tento typ maker je vyšší úrovně, neboť preprocesor zajišťuje příjem argumentů a jejich náhradu vratnými hodnotami. Typická forma je:

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

Poté, co je takové makro definováno, je spuštěno při každém výskytu slova `name` s postupnou realizací těchto kroků:

1. Přijmutí argumentů.
2. Invokace funkce makra pro přijaté argumenty.
3. Náhrada invokace poslední hodnotou funkce.
4. Pokračování v předzpracování (preprocessing) od nahrazené hodnoty (umožňující rekurzivní vyhodnocení makra).

NOTE	Určení typů argumentů není aktuálně podporováno.

Příklady

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

má za výsledek:

```
Red []
print 65536
```

Volání jiných maker z makra:

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

print make-MB 1
```

má za výsledek:

```
Red []
print 1048576
```

2.2. Makra s porovnávanými vzory

Místo porovnávání slova s přijímaným argumentem, tento typ makra porovnává vzor poskytnutý jako pravidlo nebo klíčové slovo dialektu Parse. Stejně jako u pojmenovaných maker se vracená hodnota používá jako náhrada za shodující se vzor.

K tomuto typu makra existuje také verze na nižší úrovni, která je spouštěna použitím atributu `[manual]`. V takovém případě nexistuje žádná implicitní akce; plná kontrola je přenechána uživateli. Nekoná se žádná automatická náhrada a provedení požadovaných transformací, včetně nastavení pokračovacího bodu závisí na funkci makra.

Typická forma maker typu **pattern matching** je tato:

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

Částí `<rule>` může být:

- hodnota typu `lit-word!`: pro vyhledání určitého slova.
- hodnota typu `word!`: klíčové slovo dialektu Parse, jako je název datového typu nebo `skip` pro **všechny** hodnoty.
- hodnota typu `block!`: pravidlo dialektu Parse.

Argumenty `start` a `end` jsou reference, vymezující porovnávané hodnoty ve zdrojovém kódu. Vratná hodnota musí být referencí na počáteční pozici.

Příklady

```
Red []

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

will result in:

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

Použití blokového pravidla pro vytvoření funkce s proměnnou aritou:

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

will result in:

```
    Red []
    print 8
```

3. Direktiva

3.1. #if

Skladba

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

<expr> : výtaz, jehož poslední hodnota bude použita jako podmínka.
<body> : vložený kód při splnění podmínky <expr>.
```

Popis

Vloží blok kódu, je-li podmínkový výraz pravdivý. Je-li blok `<body>` vložen, bude rovněž postoupen

preprocesoru.

Příklady

```
Red []

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

bude mít za následek následující kód při běhu na Windows:

```
Red []

print "OS is Windows"
```

pokud ne, vrací pouze:

```
Red []
```

Je také možné definovat vlastní slovo použitím direktivy `#do`, jež může být použito v podmínkových výrazech později:

```
Red []

#do [debug?: yes]

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

bude mít za následek:

```
Red []

print "running in debug mode"
```

3.2. #either

Skladba

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

<expr>  : výraz, jehož poslední hodnota bude použita jako podmínka.
<true>  : vkládaný kód při splnění podmínky <expr>.
<false> : vkládaný kód při nesplnění podmínky <expr>.
```

Popis

Výběr vkládaného bloku s kódem v závislosti na splnění podmínkového výrazu. Vkládaný blok bude rovněž předán preprocesoru.

Příklad

```
Red []

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

bude mít za následek následující kód při běhu na Wincows:

```
Red []

print "Windows"
```

v opačném případě bude výsledkem:

```
Red []

print "Unix"
```

3.3. #switch

Skladba

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

<valueN>  : porovnávaná hodnota.
<caseN>   : vložený kód při shodě poslední testované hodnoty
<default> : vložený kód, nedojde-li ke shodě u žádné hodnoty.
```

Popis

Výběr vkládaného bloku kódu mezi několika možnostmi v závislosti na hodnotě. Vložený blok bude rovněž předán preprocesoru.

Příklad

```
Red []

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

bude mít za následek následující kód při běhu na Windows:

```
Red []

print "Windows"
```

3.4. #case

Skladba

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

<exprN> : podmínkový výraz.
<caseN> : příslušná varianta vkládaného kódu
```

Popis

Výběr bloku kódu, který odpovídá splněné podmínce. Vložený blok bude rovněž předán preprocesoru.

Příklad

```
Red []

#do [level: 2]

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

bude mít za následek:

```
Red []

print "Medium"
```

3.5. #include

Skladba

```
#include <file>

<file> : Vkládaný soubor Redu (file!).
```

Popis

Při vyhodnocení při kompilaci načíst obsah souboru s argumenty a vložit jej v aktuální pozici do skriptu. Soubor může obsahovat absolutní nebo relativní cestu vzhledem k aktuálnímu skriptu. Direktivy z připojovaného souboru nemají přístup k hodnotám direktiv rozšiřovaného souboru.

```
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.
```

Při realizaci v interpretu Redu je tato direktiva pouze nahrazena příkazem `do` a k žádné inkluzi nedojde.

3.6. #do

Skladba

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

<body> : jakýkoliv kód Redu.
```

Popis

Vyhodnotí tělo bloku ve skrytém prováděcím kontextu. Při použití slova `keep` nahradí direktivu a argument výsledkem vyhodnoceného těla (`body`).

Příklad

```
Red []

#do [a: 1]

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

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

vyústí v:

```
Red []

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

3.7. #macro

Skladba

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

<name>    : jméno funkce makra (set-word!).
<pattern> : srovnávací pravidlo pro spuštění makra (block!, word!, lit-word!).
<spec>    : blok specifikací pro funkci makra.
<body>    : blok s tělem funkce makra.
```

Popis

Vytvoření funkce makra.

U pojmenovaného makra může blok specifikací deklarovat libovolný počet argumentů. Tělo musí vrátit hodnotu, která se použije jako náhrada za volání makra s jeho argumenty. Vracený prázdný blok pouze odstraní makro a jeho argumenty.

U makra s porovnávacími vzory smí blok specifikací deklarovat pouze dva argumenty - počáteční a koncový odkaz porovnávacího vzoru. Podle zavedených zvyklostí těmito argumenty jsou: `func [start end]` nebo ve zkrácené formě `func [s e]`. Implicitně potřebuje tělo vrátit hodnotu, kterou se nahradí porovnávaný vzor. Vracení prázdného bloku pouze odebere porovnávaný vzor.

Manuální mód je rovněž k disposici pro p-m (pattern-matching) makra. Lze jej nastavit zadáním atributu `[manual]` ve specifikačním bloku funkce: `func [[manual] start end]`. Takovýto ruční mód vyžaduje aby makro vrátilo počáteční pozici (místo hodnoty pro přemístění). Je-li zapotřebí **přepracovat** přemístěný vzor, potom se vrací hodnota veličiny `start`. Potřebuje-li přeskočit (skip) porovnávaný vzor, potom se vrací hodnota `end`. Jiné pozice mohou být rovněž vráceny, v závislosti

na makrem dosažených transformacích a potřebě částečně či zcela přepracovat přemístěné hodnoty.

Makro s porovnávacími vzory přijímá:

- blok: určuje porovnávaný vzor s použitím dialektu Parse.
- slovo: určuje platné slovo dialektu Parse (jako jméno datového typu nebo `skip` pro shodu se všemi hodnotami).
- lit-word: určuje porovnávané literálové slovo.

Při použití lit-wordu pro porovnávání, působí makro jako nízko úrovňová verze pojmenovaného makra, bez automatické náhrady či ošetření argumentu, avšak s požadavkem vrátit se do pokračovací pozice.

Příklady

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

vyústí v:

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

Pattern-matching macro example:

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

print 9 + 16 = 25
```

vyústí v:

```
Red []
print true
```

3.8. #local

Skladba

```
#local [<body>]

<body> : libovolný kód Redu obsahující lokální definice maker
```

Popis

Vytvoří lokální kontex maker. Všechna makra, definovaná v tomto kontextu, budou při exitu odvržena. Lokální makra tedy potřebují být lokálně použita. Tato direktiva může být použita rekurzivně (`#local` je platná direktiva v `<body>`).

Příklad

```
Red []
print 1.0
#local [
    #macro float! func [s e][s/1: to integer! s/1 next s]
    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

Skladba

```
#reset
```

Popis

Resetovat skrytý kontext, uvolňujíce z něj všechna předtím definovaná slova a makra.

3.10. #process

Skladba

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

Popis

Povolit či zamezit použití preprocesoru (implicitně je povoleno). Toto je únikový (escape) mechanizmus, který má zabránit zpracování těch částí souborů Red, kde jsou direktiva použita doslovně (literally) a nejsou určena pro preprocesor (například při použití v dialektu s odlišným významem).

Omezení implementace: při opětovném povolení preprocesoru po jeho předchozím nepovolení, potřebuje být direktiva `#process off` na stejné (nebo vyšší) úrovni zanoření v kódu.

Příklad

```
Red []

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

vyústí v:

```
Red []

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

3.11. #trace

Skladba

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

Popis

Povolit nebo zamezit výstupu ladění vyhodnocovaných výrazů a maker na monitor. Nejsou žádná specifická omezení pro použití této direktivy ve zdrojových souborech Redu.

4. Runtime API

Preprocesor Redu umí také pracovat při běhu programu (run-time) a to proto aby byl schopen vyhodnotit zdrojový kód s použitím preprocesorových direktiv také z interpreta. Bude invokován automamticky při použitím funkce `do` pro hodnotu `file!`. Vězte, že následující forma `do` může být použita pro soubor i bez invokace preprocesoru: `do load %file`.

4.1. expand-directives

Skladba

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

<body> : libovolný kód Redu, obsahující direktiva preprocesoru.
```

Popis

Invokovat preprocesor pro hodnotu bloku. Blok argumentu bude modifikován a použit jako vratná hodnota. Je-li použito zjemnění `/keep`, je předchozí stav preprocesoru zachován se všemi jeho makry a slovy předtím definovanými. V opačném případě se preprocesor spustí s čistým štítem (clean state).

Příklad

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

na platformě Windows vrátí:

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