プリプロセッサ

目次

1. コンセプト 1
  1.1. Configオブジェクト 2
  1.2. 実行コンテキスト 2
  1.3. 実装上の注意 2
2. マクロ 3
  2.1. 名前付きマクロ 3
  2.2. パターンマッチマクロ 4
3. ディレクティブ 5
  3.1. #if 5
  3.2. #either 6
  3.3. #switch 7
  3.4. #case 7
  3.5. #include 8
  3.6. #do 8
  3.7. #macro 9
  3.8. #local 11
  3.9. #reset 11
  3.10. #process 12
  3.11. #trace 12
4. ランタイムAPI 13
  4.1. expand-directives 13

1. コンセプト

RedプリプロセッサはRedのダイアレクトであり、Red言語の最も上に存在する特別なレイヤーにおいてRedのソースコードを変換することを可能にします。変換はRedソースコード中の、ディレクティブ（`directives`）と呼ばれるインラインプリプロセッサ用のキーワードに基づいて行われます。directiveは次のタイミングで処理されます。

- Redのソースコードがコンパイルされる時
- `do` 関数にRedのソースコードファイルが渡された時
- blockの値に対して `expand-directives` 関数が呼ばれた時

プリプロセッサはLOADフェーズの後に実行されます。そのため、ソースコードを文字列的に扱うのではなく、Redの各種の値として処理します。

ディレクティブは以下のカテゴリーに分けられます。

- 条件分岐ディレクティブ: 式の結果に応じたコードを挿入します。
- 制御ディレクティブ: プリプロセッサの振る舞いを制御します。

- [illegible] [illegible]

[illegible] `#` [illegible] issue! [illegible]

[illegible]

NOTE Red/System[illegible] [illegible] [illegible]

1.1. Config[illegible]

[illegible] `config` [illegible] [illegible] [illegible]

[illegible]

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

NOTE: [illegible]Red[illegible] `config` [illegible] `config` [illegible]Red[illegible]

1.2. [illegible]

[illegible]set-word[illegible] `#do` [illegible]

Tips:

- [illegible]

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

- [illegible]

  ```
  probe preprocessor/exec
  ```

1.3. [illegible]

[illegible]Rebol2[illegible]Red[illegible]Red[illegible]Red[illegible]

2. マクロ

Redのプリプロセッサはマクロ関数（単に マクロ と呼ばれることが多いです）をサポートしています。マクロを使うことで、より複雑な変換が行え、効率的なメタプログラミングを可能にします。実行コストをコンパイル時に発生させ、実行時のコストを回避できます。Redはマクロが実装される以前から実行時の強力なメタプログラミング性能を持っていましたが、Redのソースコードがコンパイラ上でもインタプリタ上でも同様に動作するため、マクロは実行時にも解決されるようになっています。

NOTE:ソースの読み取り時のマクロは存在しません。文字列としてソースを取り扱う方法はすでにParseダイアレクトが存在するため、そのような機能のサポートは現在のところは冗長だと捉えられています。

プリプロセッサは２つのタイプのマクロをサポートしています。

2.1. 名前付きマクロ

このタイプのマクロはハイレベルで動作し、引数や戻り値の取り扱いまでプリプロセッサ側で処理してくれるため、ユーザーが意識する必要はありません。典型的なフォーマットは以下のようなものです。

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

上記のようなマクロを定義した後、ソースコード中で `name` のwordが現れるたびに、以下のステップに従ってこのマクロが呼び出されます。

1. 引数の値を取得
2. 取得した引数でマクロの関数を実行
3. 関数内で最後に評価された値で、マクロ呼び出しと引数部分を置き換え
4. 置き換えられた値を元にプリプロセッサの動作を再開（再帰的なマクロの評価も可能です）

NOTE:引数の型を指定することは現在のところサポートされていません。

例

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

この結果は以下のように変換されます。

```
Red []
print 65536
```

マクロの中から他のマクロを呼び出す例です。

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

```
print make-MB 1
```

□□□□□□□□□□□□□□□□□□

```
Red []
print 1048576
```

2.2. □□□□□□□□□□

□□□□□□□□□□word□□□□□□□□□□□□□□□□□□Parse□□

□□□□□□□□□□□□□□□□□□□□□□□□□ `[manual]` □□

□□□□□□□□□□□□□□□□□□□□□□□□□□

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

`<rule>` □□□□□□□□□□□□□□□□□□□

- lit-word!□□□□□□□□word□□□□□□□□□□□
- word!□□□Parse□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□ □skip□□□□□□
- block!□□□□Parse□□□□□□□□□□□

□□ `start` □ `end` □□

`<attribute>` □□ `[manual]` □□□

□□

```
Red []

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

□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
Red []

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

block□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□

```
Red []
print 8
```

3. □□□□□□□

3.1. #if

□□

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

<expr> : □□□□□□□□□□□□□□□□□□□□
<body> : if <expr> □true□□□□□□□□□□□□□□□
```

□□

□□□□true□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `<body>` □□□□□□□□□□□□□□□□□□□□□

□

```
Red []

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

Windows□□□□□□□□□□□□□□□□□□□□

```
Red []

```

```
print "OS is Windows"
```

Windows□□□□□□□□□□□□□□□□□

```
Red []
```

#do □□□□□□□□□□□□□□word□□□□□□□□□□□□□□□□□□□□□□

```
Red []

#do [debug?: yes]

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

□□□□□□□□□□□□□□□□□□

```
Red []

print "running in debug mode"
```

3.2. #either

□□

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

<expr>  : □□□□□□□□□□□□□□□□□□
<true>  : if <expr> □true□□□□□□□□□□□□□
<false> : if <expr> □false□□□□□□□□□□□□□
```

□□

□□

□

```
Red []

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

Windows□□□□□□□□□□□□□□□□□□□□□

```
Red []
```

```
print "Windows"
```

Windows[illegible]

```
Red []

print "Unix"
```

3.3. #switch

[illegible]

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

<valueN>   : [illegible]
<caseN>    : [illegible]
<default> : [illegible]
```

[illegible]

[illegible]

[illegible]

```
Red []

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

Windows[illegible]

```
Red []

print "Windows"
```

3.4. #case

[illegible]

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

<exprN> : [illegible]
```

```
<caseN> : 条件式がtrueだった場合に挿入されるコード
```

説明

値に応じて、複数の選択肢の中から挿入するコードを選択します。挿入されたブロックは再度プリプロセッサに渡されます。

例

```
Red []

#do [level: 2]

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

この結果は以下のように変換されます。

```
Red []

print "Medium"
```

3.5. #include

構文

```
#include <file>

<file> : 挿入するRedのファイル（file!）
```

説明

コンパイル時に評価された場合、引数のファイルの中身を読み込んで現在の位置に挿入します。ファイルは現在のスクリプトからの絶対パスまたは相対パスを使うことができます。Redインタプリタで実行された場合、このディレクティブは単に `do` に置換され、ファイルの挿入は行われません。

3.6. #do

構文

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

<body> : 任意のRedのコード
```

説明

暗黙的な実行コンテキスト上で、bodyブロックを評価します。 `keep` が使用された場合、 `body` を評価した結果でディレクティブと引数が置換されます。

例

```
Red []

#do [a: 1]

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

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

この結果は以下のように変換されます。

```
Red []

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

3.7. #macro

構文

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

<name>    : マクロ関数の名前（set-word!）
<pattern> : マクロを実行するマッチングルール（block!, word!, lit-word!）
<spec>    : マクロ関数のスペックブロック
<body>    : マクロ関数のボディブロック
```

説明

マクロ関数を作成します。

名前付きマクロでは、specブロックでは任意の数の引数を宣言できます。bodyブロックはマクロ呼び出しと引数を置換するための戻り値を返す必要があります。空のブロックを返した場合、マクロ呼び出しと引数は単に削除されます。

パターンマッチマクロの場合、specブロックは 2個の引数を宣言する必要があり、それらがマッチしたパターンへの開始位置の参照と終了位置の参照になります。規約として、引数の名前は `func [start end]` とするか省略して `func [s e]` とします。デフォルトでは、マクロの本体はマッチしたパターンを置換する値を返す必要があります。空のブロックを返した場合、マッチしたパターンは削除されます。 パターンマッチングマクロではマニュアルモード も使用可能で、関数のspecブロックに `[manual]` 属性を設定することで実行されます。つまり、次のような形です。 `func [[manual] start end]` マニュアルモードのマクロは置換する値ではなく、再開位置を返す必要があります。もし置換されたパターンで 再評価 をしたいのであれば、 `start` を返してください。置換されたパターンは スキップ するのであれば、 `end` を返してください。必要手であれば他の位置を返すことも可能です。マクロによって変換された内容や、置換さあれた値を部分的に再評価したいか、全体を再評価したいかといったことを考慮して、返す位置を決めてください。

□□□□□□□□□□□□□□□□□□□□□□□□

- block□Parse□□□□□□□□□□□□□□□□□□□□□□□□
- word□Parse□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□ □skip□□□□□□
- lit-word□□□□□word□□□□□□□□□□□

□

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

□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□

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

print 9 + 16 = 25
```

□□□□□□□□□□□□□□□□□

```
Red []
print true
```

□□□□□□□□□□□□□□□□□□□□

```
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             ;-- □□□□□□□□□□□□□□□□□□□□□□
]

print sqrt 9
print sqrt -4
```

この結果は以下のような動作になります。

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

3.8. #local

構文

```
#local [<body>]

<body> : ローカルマクロの定義を含む任意のRedコード
```

説明

マクロのローカルコンテキストを生成します。ローカルコンテキスト中に定義された全てのマクロはコンテキストの終了時に破棄されます。ローカルなものとして定義したいマクロはローカルコンテキストの中で定義する必要があります。このディレクティブは再帰的に使用することができます（ `#local` は `body` 中で使用できる有効なディレクティブです）。

例

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

この結果は以下のように変換されます。

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

3.9. #reset

構文

```
#reset
```

説明

保持されているマクロコンテキストをリセットします。定義済みの全てのword、マクロは初期化（破棄）されます。

3.10. #process

構文

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

説明

プリプロセッサの有効・無効を切り替えます（初期状態は有効です）。これはRedのコード中で、ディレクティブがリテラルとして使用されていて、プリプロセッサとして処理をさせたくない場合のための、エスケープの仕組みです。たとえば、そのプリプロセッサが他の意味を持つダイアレクトで使われている時などに使用します。

実装上の制約: プリプロセッサを無効にした後、再度有効にするという場合、`#process off` ディレクティブは `#process on` ディレクティブと同じか、より上のスコープに位置している必要があります。

例

```
Red []

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

この結果は以下のように変換されます。

```
Red []

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

3.11. #trace

構文

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

説明

マクロの評価中のデバッグ出力を表示するかどうかを切り替えます。このディレクティブがRedのソースコードで使用できる箇所について特に制約はありません。

4. ランタイムAPI

Redのプリプロセッサは実行中も動作するようになっており、インタプリタでもプリプロセッサディレクティブを含むコードを実行することができます。`do` 関数に `file!` 型の値を渡した場合に、自動的にプリプロセッサの評価が行われます。もしプリプロセッサを実行させずに `do` 関数にファイルを渡したい場合、次の形式を使います: `do load %file`

4.1. expand-directives

構文

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

<body> : プリプロセッサディレクティブを含む任意のRedコード
```

説明

ブロックの中のプリプロセッサを実行します。引数のブロックは変更され、戻り値として返されます。`/clean` リファインメントを使用した場合、以前のプリプロセッサの状態がリセットされ、全ての以前の定義されたマクロは削除されます。

例

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

この結果はWindows上では以下のように変換されます。

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