プリプロセッサ

目次

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。

- マクロ: 関数を使ったより複雑なコード変換を行います。

ディレクティブは `#` シンボルで始まる `issue!` 型の値で表現されます。

ディレクティブが処理されると、そのディレクティブが返す戻り値でコードが置き換わります。いくつかのディレクティブは値を返さないですが、その場合は単に削除されます。ソースコードの変換はこのように行われます。

> **NOTE** Red/Systemは独自の プリプロセッサ を持っています。これはディレクティブと似ていますが、ローレベルで文字列的な取り扱いにより適用されます。

1.1. Configオブジェクト

条件分岐を行うディレクティブやマクロのでは暗黙的に提供される `config` オブジェクトが使用可能です。これはコンパイル時の設定へのアクセスを可能にし、条件に従ってコードを挿入したい場合によく用いられます。参照できる情報の完全なリストは こちら にあります。

例:

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

NOTE: プリプロセッサが実行時に（つまり、Redのインタプリタ上で）呼び出された場合でも、 `config` オブジェクトは参照可能です。この場合、 `config` オブジェクトにはコードを実行しているRedのバイナリのコンパイルオプションが設定されています。

1.2. 実行コンテキスト

ディレクティブの中で使用される式は全て、現在のコンテキストにのみ反映され、グローバルコンテキストにリークして予期しない副作用を起こすことはありません。コンテキストは式の中でset-word、マクロ、 `#do` ディレクティブが使用されるたびに拡張されます。

Tips:

- 以下のコードにより、隠蔽されたコンテキストの内容を表示することができます。

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

- 実行時に使用される場合、隠蔽コンテキストへのアクセスは以下のように行います。

  ```
  probe preprocessor/exec
  ```

1.3. 実装上の注意

現在のところ、コンパイル時に呼び出されるディレクティブ中の式はRebol2のインタプリタで評価されています。ツールチェインのコードを実行する場合と同じです。これは一時的なもので、できるだけ早くRedのエバリュエータに変更される予定です。それまでの間は、Redのランタイム（また、将来のコンパイル時の）における動作の互換性のため、使用する式やマクロがRedでも実行できるようにしておいてください。

2. [illegible]

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

NOTE:[illegible]Parse[illegible]

[illegible]

2.1. [illegible]

[illegible]

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

[illegible] `name` [illegible]word[illegible]

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]

NOTE:[illegible]

[illegible]

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

[illegible]

```
Red []
print 65536
```

[illegible]

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

print make-MB 1
```

この結果は以下のように変換されます。

```
Red []
print 1048576
```

2.2. パターンマッチマクロ

このタイプのマクロはwordと引数でマッチングされるのではなく、Parseダイアレクトのルールとキーワードで定義されたパターンによってマッチングを行います。名前付きマクロと同様、戻り値がマッチしたパターンを置換するために使われます。

もう一つ、よりローレベルのマクロも存在し、こちらは `[manual]` 属性を使用することで起動されます。この場合、暗黙的な処理はありませんが、ユーザーが全てをコントロールすることができます。自動的な置換は行われず、必要な変換を適用するのもコードの評価を再開する位置を決めるのも定義したマクロ関数次第です。

典型的なパターンマッチマクロは以下のような形式です。

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

`<rule>` は以下のものを使用することができます。

- lit-word!型の値: 指定したwordでマッチングされます。
- word!の値: Parseダイレクトで使用されるキーワードです。たとえばデータ型の名前や 全ての値にマッチする「skip」などです。
- block!型の値: Parseダイアレクトのルール。

引数 `start` と `end` は分割されたソースコードに対する参照です。戻り値は再開位置への参照である必要があります。

`<attribute>` には `[manual]` を指定することができます。これにより、そのマクロはローレベルのマニュアルモードで実行されます。

例:

```
Red []

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

この結果は以下のように変換されます。

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

[illegible]

```
Red []

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

block[illegible]

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

[illegible]

```
Red []
print 8
```

3. [illegible]

3.1. #if

[illegible]

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

<expr> : [illegible]
<body> : if <expr> [illegible]true[illegible]
```

[illegible]

[illegible]true[illegible] `<body>` [illegible]

[illegible]

```
Red []

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

Windowsで実行した場合、以下のように変換されます。

```
Red []

print "OS is Windows"
```

Windowsでない場合、以下の結果になります。

```
Red []
```

#do ディレクティブを使うと自由にwordを定義でき、式の中で使用することができます。

```
Red []

#do [debug?: yes]

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

この結果は以下のように変換されます。

```
Red []

print "running in debug mode"
```

3.2. #either

構文

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

<expr>  : 式（最後の値が条件分岐に用いられます）
<true>  : if <expr> がtrueだった場合に実行されるコード
<false> : if <expr> がfalseだった場合に実行されるコード
```

説明

条件式によって挿入するコードブロックを選択します。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 []

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

Windowsで実行した場合、以下のように変換されます。

```
Red []

print "Windows"
```

Windowsでない場合、以下の結果になります。

```
Red []

print "Unix"
```

3.3. #switch

構文

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

<valueN>  : マッチさせる値
<caseN>   : 最後に評価された値がマッチした時に挿入するコード
<default> : マッチする値がなかった場合に挿入するコード
```

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 []

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

Windowsで実行した場合、以下のように変換されます。

```
Red []

print "Windows"
```

3.4. #case

構文

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

<exprN> : 条件式
<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。

パターンマッチマクロには以下のデータを渡せます。

- block: Parseダイアレクトで使用するマッチパターンを指定します
- word: Parseダイレクトで使用されるキーワードです。たとえばデータ型の名前や 全ての値にマッチする「skip」などです。
- lit-word: 指定したwordでマッチングされます。

例

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

この結果は以下のように変換されます。

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

パターンマッチマクロの例:

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

print 9 + 16 = 25
```

この結果は以下のように変換されます。

```
Red []
print true
```

マニュアルモードのパターンマッチマクロの例

```
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              ;-- マッチしたパターンに渡される位置を返します。
]

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

□□

```
#reset
```

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□word□□□□□□□□□□□□□□□

3.10. #process

□□

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

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Red□□□
□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `#process off` □□□□□□□ `#process on` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□

```
Red []

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

□□□□□□□□□□□□□□□□

```
Red []

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

3.11. #trace

□□

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

□□

[illegible]Red[illegible]

4. □□□□□API

Red[illegible] `do` □□□ `file!` [illegible] [illegible] `do` [illegible] `do load %file`

4.1. expand-directives

□□

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

<body> : [illegible]Red□□□
```

□□

[illegible] `/clean` [illegible]

□

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

□□□□□Windows[illegible]

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