



# Red Programming Language

Guaracy Monteiro

<https://github.com/guaracy/red>

24 de novembro de 2015

## Sumário

<b>1</b>	<b>Introdução</b>	<b>4</b>
1.1	Objetivo . . . . .	4
1.2	A Linguagem . . . . .	4
1.3	Configuração . . . . .	4
<b>2</b>	<b>Sintaxe</b>	<b>5</b>
2.1	Delimitadores . . . . .	5
2.2	Sintaxe livre . . . . .	5
2.3	Comentários . . . . .	6
<b>3</b>	<b>REPL</b>	<b>7</b>
<b>4</b>	<b>Variáveis</b>	<b>9</b>
4.1	Nomenclatura . . . . .	9
4.2	Operação . . . . .	9
<b>5</b>	<b>Tipos de dados</b>	<b>11</b>
5.1	binary! . . . . .	12
5.2	bitset! . . . . .	12
5.3	block! . . . . .	12
5.4	char! . . . . .	13
5.5	datatype! . . . . .	14
5.6	error! . . . . .	14
5.7	file! . . . . .	14
5.8	float! . . . . .	14
5.9	function! . . . . .	14
5.10	get-path! . . . . .	15
5.11	get-word! . . . . .	15
5.12	hash! . . . . .	15
5.13	integer! . . . . .	15
5.14	issue! . . . . .	15
5.15	lit-path! . . . . .	15
5.16	lit-word! . . . . .	15
5.17	logic! . . . . .	16
5.18	map! . . . . .	16
5.19	native! . . . . .	16
5.20	none . . . . .	16
5.21	object! . . . . .	16
5.22	op! . . . . .	16

5.23	pair!	18
5.24	paren!	18
5.25	path!	19
5.26	percent!	19
5.27	point!	20
5.28	refinement!	20
5.29	routine!	20
5.30	set-path!	20
5.31	set-word!	20
5.32	string!	21
5.33	tuple!	21
5.34	typeset!	21
5.35	unset!	21
5.36	url!	22
5.37	vector!	22
5.38	word!	22
<b>6</b>	<b>Expressões</b>	<b>22</b>
<b>7</b>	<b>Funções</b>	<b>22</b>
<b>8</b>	<b>Escopo</b>	<b>22</b>
<b>9</b>	<b>Operadores</b>	<b>22</b>
<b>10</b>	<b>Controle de fluxo</b>	<b>22</b>
<b>11</b>	<b>Exceções</b>	<b>22</b>
<b>12</b>	<b>Pilha</b>	<b>22</b>
<b>13</b>	<b>Depuração</b>	<b>22</b>
<b>14</b>	<b>Estrutura do sistema</b>	<b>22</b>
<b>15</b>	<b>Palavras reservadas</b>	<b>22</b>
<b>16</b>	<b>VID</b>	<b>22</b>

## Listagens

1	Sintaxe livre . . . . .	6
---	-------------------------	---

## Lista de Tabelas

Rascunho

# 1 Introdução

## 1.1 Objetivo

O objetivo inicial não é o de ser um manual, livro ou algo do gênero sobre a linguagem. Apenas um local para que eu possa agrupar as informações e o conhecimento sobre a linguagem. Em segundo lugar, compartilhar a linguagem com quem estiver interessado. Vender e ficar rico está fora de cogitação. :D

## 1.2 A Linguagem

A linguagem **Red** é fortemente baseada em REBOL compartilhando, entre outros, a homoiconicidade, o grande número de tipos de dados, a mistura código+data como em Lisp. Como diferenças é possível citar a possibilidade de gerar executáveis em código nativo (não precisa ser na mesma plataforma de desenvolvimento) e a tipagem opcional para parâmetros nas funções.

## 1.3 Configuração

Para criar um ambiente de desenvolvimento não são necessários poderes especiais. A primeira coisa a fazer é baixar de [www.red-lang.org](http://www.red-lang.org) a versão para o seu sistema operacional e colocar no local que ficar mais conveniente. Note que para o Linux, a versão disponível é de 32 bits. Para rodar em uma instalação de 64 bits é necessário instalar algumas bibliotecas para suportar a versão. As formas mais comuns de executar o programa são:

- Apenas executar o programa **red** e entrará no REPL (read-eval-print loop), isto é, um ambiente interativo onde você vai digitando e executando as instruções.
- Executando **red <arquivo.red>** o script existente no arquivo será executado.
- Executando **red -c <arquivo.red>** o script existente no arquivo será compilado e irá gerar um executável para a plataforma atual.
- Executando **red -c -t <plataforma><arquivo.red>**, o script será compilado para a plataforma especificada. Assim você pode estar no Linux e gerar executáveis, por exemplo, para Linux, Windows, Android e OSX, sem a necessidade de trocar de ambiente. As plataformas disponíveis são:

- **MSDOS** : Windows, x86, aplicações console (+ GUI)
- **Windows** : Windows, x86, GUI applications
- **Linux** : GNU/Linux, x86
- **Linux-ARM** : GNU/Linux, ARMv5, armel (soft-float)
- **RPi** : GNU/Linux, ARMv5, armhf (hard-float)
- **Darwin** : MacOSX Intel, apenas aplicações console
- **Syllable** : Syllable OS, x86
- **FreeBSD** : FreeBSD, x86
- **Android** : Android, ARMv5
- **Android-x86** : Android, x86

## 2 Sintaxe

Antes de começar qualquer coisa, aprender um pouco da sintaxe é importante. Até porque você deve estar acostumado com aquelas linguagens complicadas onde é necessário separar algumas coisas com vírgula, outras com ponto e vírgula, outras com chaves, outras com colchetes, etc., etc., etc.. Então vamos lá:

### 2.1 Delimitadores

Basicamente são três os delimitadores. Para string, blocos e caminho.

- **Strings** : utiliza-se aspas ( "string" ) para strings que não possuam quebra de linha no interior ou chaves ( {string até aqui} ) caso a string tenha mais de uma linha.
- **Blocos** : os blocos são delimitados por colchetes ( [ ] ) e não possuem limite.
- **Caminhos** : são delimitados (ou concatenados) com a barra invertida ( \ )

### 2.2 Sintaxe livre

O delimitador padrão é o espaço e, a única restrição é separar os tokens por um ou mais espaços. Os códigos abaixo são todos válidos e possuem a mesma avaliação:

```
1 while [a > 0][print "loop" a: a - 1]
2
3 while [a > 0]
4   [print "loop" a: a - 1]
5
6 while [
7   a > 0
8 ][
9   print "loop"
10  a: a - 1
11 ]
12
13 while [a > 0][
14   print "loop"
15   a: a - 1
16 ]
17
18 while [a > 0][
19   print "loop"
20   a: a - 1]
```

Listagem 1: Sintaxe livre

Note que, se você entrar com **a < 0** ou **a-1** (sem espaços) causará um erro. Ou melhor, poderá causar um erro já que serão consideradas como palavras (variáveis) e poderão existir e conter um valor válido.

## 2.3 Comentários

Existem dois tipos de comentários (trechos que são ignorados pelo programa):

- O comentário que inicia com ponto e vírgula ( ; ) e vai até o final da linha e pode ser utilizado em qualquer parte do programa e
- o comentário com mais de uma linha que inicia com **comment {** e termina com um fecha chave ( **}** ) pode ser utilizado em qualquer parte do programa menos no meio de uma expressão.

### 3 REPL

Em vez de criar um script em um editor, executar e/ou compilar, acredito que o mais interessante no início seja digitar e ver o resultado. Para tal, basta usar o REPL (read-eval-print-loop). Como o nome já diz, ele lê uma entrada efetuada pelo usuário, efetua uma avaliação, mostra o resultado e fica esperando uma nova entrada. Para iniciar, basta executar **red** sem nenhum argumento e deverá aparecer algo como:

```
---== Red 0.5.4 ===--
Type HELP for starting information.

red>>
```

Para sair digite **q** ou **quit** e pressione enter. Digitando **help** e enter, serão apresentadas algumas opções para auxílio. Lembre-se que o

Apesar de não necessitar a digitação de **Red** [ ] que aparecem nas listagens para efeitos de salientar a sintaxe do script, se entrar no REPL não terá problema nenhum. O REPL entende que a entrada de uma nova linha seja a indicação para que ele avalie a entrada. Faz-se necessário que o comando seja digitado em uma linha a menos que ele termine com a abertura de um bloco [. Neste caso, ele mudará o prompt de **red >>** para uma abertura de colchetes [ indicando que está esperando o fechamento para avaliar a expressão.

```
red>> a: 5
== 5
red>> while [a > 0]
*** Script error: while is missing its body argument
*** Where: while
red>> while [a > 0] [
[   print
[   "loop"
[   a: a - 1
[   ]
loop
loop
loop
loop
loop
loop
red>>
```



Utilizando as setas para cima e para baixo é possível navegar no histórico para a execução de expressões informadas anteriormente. Se você digitar algo e pressionar tab, o REPL irá mostrar uma relação das possíveis funções que podem ser entradas, inclusive as definidas pelo usuário. Se for digitado a e tab, teremos algo como:

```
red>> action! any any-type! all absolute add and~ append at
any-object! any-string! any-word! any-function! any-block!
arcsine arccosine arctangent arctangent2 as-pair any-path!
a attempt action? ask a-an acos asin atan aqua any-block?
any-function? any-object? any-path? any-string? any-word?
atan2 and about
red>> a
```

Se você digita **help** ou **?** seguido de uma função, será mostrado um resumo da função informando como ela é utilizada, uma breve descrição da função, os argumentos e alguns refinamentos. Para insert, temos:

```
red>> help insert

USAGE:
    insert series value /part length /only /dup count

DESCRIPTION:
    Inserts value(s) at series index; returns series head.
    insert is of type: action!

ARGUMENTS:
    series [series! bitset! map!]
    value [any-type!]

REFINEMENTS:
    /part => Limit the number of values inserted.
           length [number! series!]
    /only => Insert block types as single values (overrides /part).
    /dup  => Duplicate the inserted values.
```

```
count [number!]  
  
red>>
```

## 4 Variáveis

Toda a linguagem possui alguma forma de armazenar um determinado valor em algum lugar. No caso de **Red** variáveis (ou palavras) podem armazenar (ao associar) dados ou código.

### 4.1 Nomenclatura

### 4.2 Operação

A associação ou atribuição é feita seguindo a variável com dois pontos ( : ). Por exemplo, `nome: "Fulano de Tal"` irá atribuir `"Fulano de Tal"` a variável `nome`. Para avaliar a variável e retornar o seu valor, basta informar o nome da variável. No caso de termos `cliente: nome`, indica que iremos atribuir o conteúdo da variável `nome` para a variável `cliente`. Existem casos onde não é possível a construção acima como no caso onde o conteúdo da variável é uma função. Para tal, precede-se o nome da variável com dois pontos e será retornado o conteúdo mas não será avaliado. Finalmente podemos tratar a variável como símbolo e retornará o nome da variável. Por exemplo:

```
red>> a: 25  
== 25  
red>> b: [1 2 3 4 5]  
== [1 2 3 4 5]  
red>> c: 26  
== 26  
red>> sum: func [a b][a + b]  
== func [a b][a + b]  
red>> sum 1 2  
== 3  
red>> a: 25  
== 25  
red>> b: [1 2 3 4 5]  
== [1 2 3 4 5]  
red>> sum: func ["Soma dois números" a b][a + b]
```

```
== func ["Soma dois números" a b][a + b]
red>> type? b
== block!
red>> type? a
== integer!
red>> sum a c
== 51
red>> sum a b
*** Script error: block type is not allowed here
*** Where: +
red>>
red>> x: sum
*** Script error: sum is missing its a argument
*** Where: sum
red>> x: :sum
== func ["Soma dois números" a b][a + b]
red>> x 2 3
== 5
red>> type? x
*** Script error: x is missing its a argument
*** Where: x
red>> type? :x
== function!
red>>
```

Podemos ver que **Red** não é uma linguagem tipada, isto é, as variáveis podem conter qualquer valor mas, depois de assumirem um valor, possuirão um tipo correspondente que será utilizado para validar a avaliação das expressões. No caso de funções, é possível especificar o tipo de parâmetro que será aceito. Nada impede que a função trate os diversos tipos para retornar resultados válidos. No exemplo, **sum a + b** retornou um erro pois não foi possível adicionar um inteiro em um bloco (no caso pode ser considerado como uma lista). Bastiria que a função, por exemplo, adicionasse o número em cada um dos elementos. Da mesma forma, para atribuir a função **sum** para a variável **x** foi necessário utilizar o formato **x: :sum** para que a função não fosse avaliada. A construção **x: sum 2 3** seria validada e retornaria o valor **5**. Portanto:

Formato	Significado
<b>var</b>	Avalia a variável e retorna o resultado.
<b>var:</b>	Atribui um valor a uma variável.
<b>:var</b>	Retorna o valor de uma variável sem avaliá-lo.
<b>'var</b>	Trata a variável como um símbolo e retorna o valor sem avaliá-lo.

## 5 Tipos de dados

Este é um tópico onde REBOL é muito rico e, por consequência, **Red** também. A variedade de tipos é interessante para evitar a necessidade de criar código para trabalhar com os diversos dados requeridos pelo programa. Por exemplo, se voce deseja trabalhar com percentuais em determinada tarefa qual a melhor forma? Informar o decimal correspondente (e.g. 0,1 para 10%) e efetuar diretamente o cálculo ou informar o percentual e efetuar o cálculo no estilo *valor × percentual ÷ 100*? Ficaria algo assim:

```
red>> p: 15%
== 15%
red>> v: 150
== 150
red>> p * v
== 22.5
red>> type? p
== percent!
red>>
```

No momento, estão disponíveis os seguintes tipos (alguns não serão utilizados em condições normais e, mais importante, alguns ainda não foram definidos como **data** e **hora**. Por enquanto temos:

**action!**, **binary!**, **bitset!**, **block!**, **char!**, **datatype!**, **error!**, **file!**, **float!**, **function!**, **get-path!**, **get-word!**, **hash!**, **integer!**, **issue!**, **lit-path!**, **lit-word!**, **logic!**, **map!**, **native!**, **none!**, **object!**, **op!**, **pair!**, **paren!**, **path!**, **percent!**, **point!**, **refinement!**, **routine!**, **set-path!**, **set-word!**, **string!**, **tuple!**, **typeset!**, **unset!**, **url!**, **vector!**, **word!**. Alguns não importam para a programação normal, sendo mais relevantes internamente.

Para você descobrir o tipo de uma variável basta entrar com **type?** e uma variável ou um valor e será retornado o tipo atual. Para comparar com um tipo específico, basta utilizar o tipo trocando a exclamação por uma interrogação. Por exemplo, se quiser verificar se uma variável possui um valor inteiro, basta digitar **integer?** variável ou valor e retornara um verdadeiro ou falso.

Para converter um tipo de dado em outro, utilizamos **to**, o tipo de dado que desejamos e o valor a ser convertido.

```
red>> to integer! 2.33
== 2
```

```
red>> to integer! "34"
== 34
red>> to integer! 123%
== 1
red>> to integer! #"A"
== 65
red>> to integer! 2x3
*** Script error: to does not allow pair for its type argument
*** Where: to
red>>
```

Internamente, algumas conversões são executadas automaticamente para que um determinado cálculo seja efetuado. Converter um valor inteiro para ponto flutuante, por exemplo.

## 5.1 binary!

Contém um conteúdo binário. É informada digitando-se `base#{valores}`. Se a base não for informada, é assumido que os valores estão no formato hexadecimal. Para entrar com valores binários digitamos 2 e podemos utilizar a base 64. Os valores informados devem ser compatíveis com a base informada para não ocorrer um erro.

```
red>> type? #{cafe}
== binary!
red>> type? 2#{11001111}
== binary!
red>> type? 64#{Lm87GGv}
== binary!
red>> a: 64#{Lm87GGv}
== #{2E6F3B}
red>> a/1
== 46
red>> a/2
== 111
red>> a/3
== 59
red>> a/1: 128
== 128
red>> a
```

```
== #{806F3B}  
red>> length? #{cafe}  
== 2  
red>>
```

## 5.2 bitset!

TBD

## 5.3 block!

Blocos são grupos de código e/ou dados e/ou outros blocos e/ou qualquer coisa. São delimitados por colchetes `[ ]`. Imagine, em outras linguagens, uma matriz que pode conter qualquer coisa. Podemos efetuar operações em um bloco ou mesmo entre blocos.

```
red>> a: [2 4 6 8 1 3 5 7 6 2]  
== [2 4 6 8 1 3 5 7 6 2]  
red>> type? a  
== block!  
red>> first a  
== 2  
red>> a/4  
== 8  
red>> sort a  
== [1 2 2 3 4 5 6 6 7 8]  
red>> unique sort a  
== [1 2 3 4 5 6 7 8]  
red>> last a  
== 8  
red>> a  
== [1 2 2 3 4 5 6 6 7 8]  
red>> print ["um" "dois" lf "três" "quatro"]  
um dois  
três quatro  
red>> either 1 > 2 [  
[ print "1 é maior que dois." ]  
[ print "1 é menor que dois." ]  
1 é menor que dois.  
red>>
```

## 5.4 char!

Representa um caractere qualquer. O caractere é escrito entre aspas precedido de `#`. Para representarmos o caractere `A` usamos `#"A"`. Se você esquecer o `#`, será tratado como uma string. Para representar caracteres com valores menor que 32 (espaço), prefixamos com um sinal circunflexo. Para o caractere 27 (escape) temos `#"^["`. O ambiente já define alguns caracteres para facilitar a utilização em algumas ocasiões.

```
red>> ? char!
      comma      :  #", "
      cr         :  #"^M"
      dot        :  #". "
      escape     :  #"^["
      lf         :  #"^/"
      newline    :  #"^/"
      null       :  #"^@"
      slash      :  #"/ "
      sp         :  #" "
      space      :  #" "
      tab        :  #"^-"

red>> type? "A"
== string!
red>> type? #"A"
== char!
red>> char? #"^@"
== true
red>> to integer! #"^@"
== 0
red>> to integer! null
== 0
red>> char? null
== true
red>>
```

## 5.5 datatype!

Uso interno.

```
type? error! = datatype!
```

## 5.6 error!

TBD.

## 5.7 file!

TBD

## 5.8 float!

Valores numéricos não inteiros. Você pode usar o ponto ou a vírgula para separar a parte decimal. Os valores `123.45` e `123,45` são iguais. Também é possível utilizar `e` (maiúsculo ou minúsculo) para informar um expoente. Por exemplo `1.2345e3 = 1234.5`

## 5.9 function!

Indicam que a variável contém uma função. Você poderá utilizar `help função` para verificar a utilização da mesma e/ou digitar `source função` para ver o código fonte da função. Outras informações na seção sobre funções ([7](#))

```
red>> dobro: function[x][x * 2]
== func [x][x * 2]
red>> dobro 3
== 6
red>> type? :dobro
== function!
red>>
```

## 5.10 get-path!

TBD

## 5.11 get-word!

TBD



## 5.12 hash!

TBD

## 5.13 integer!

TBD

## 5.14 issue!

Usado para identificar sequencias diversas como telefone, cep, etc.. Inicia com **#** e segue até o próximo delimitador. Facilita a visualização de determinados valores. Para trabalharmos com um cep, por exemplo, fica mais legível entrarmos com **#25477-122** do que um valor inteiro como **25477122** ou ponto flutuante do tipo **25477.122**.

## 5.15 lit-path!

TBD

## 5.16 lit-word!

TBD

## 5.17 logic!

São variáveis cujo conteúdo se restringe a falso e verdadeiro. Diferente de algumas linguagens que consideram 0 como falso e qualquer outro valor como verdadeiro, em **Red** isto não é possível. De qualquer forma, para melhorar o entendimento do programa, outros valores também são definidos como falso e verdadeiro (nada que não possa ser feito na maioria das linguagens). Por exemplo:

```
red>> ? logic!
false      : false
no         : false
off        : false
on         : true
true       : true
yes        : true
red>>
```

## 5.18 map!

TBD

## 5.19 native!

TBD

## 5.20 none

Basicamente, informa que uma variável não contém nenhum valor. Não é equivalente a **zero** ou **false** ou **"** e só pode ser comparada com **none**.

## 5.21 object!

TBD

## 5.22 op!

Você poderia perguntar: Tipo de variável operador? Não seria função? Bem, podemos ver um operador do tipo soma como uma função. Por exemplo, **a + b** poderia ser visto como algo do tipo **add a b** (Se você conhece Lisp, a expressão **(+ a b)** faz todo o sentido). Uma lista dos operadores atualmente definidos:

```
red>> ? op!
% => Returns what is left over when one value is divided by another
* => Returns the product of two values
** => Returns a number raised to a given power (exponent)
+ => Returns the sum of the two values
- => Returns the difference between two values
/ => Returns the quotient of two values
// => Compute a nonnegative remainder of A divided by B
< => Returns true if the first value is less than the second
<<
<= => Returns true if the first value is less than or equal to the second
<> => Returns true if two values are not equal
= => Returns true if two values are equal
== => Returns true if two values are equal, and also the same datatype
=? => Returns true if two values have the same identity
> => Returns true if the first value is greater than the second
>= => Returns true if the first value is greater than or equal to the second
>>
```

```

>>>
and      => Returns the first value ANDed with the second
or => Returns the first value ORed with the second
xor      => Returns the first value exclusive ORed with the second
red>>

```

Por exemplo, podemos ter um programa onde precisamos utilizar frequentemente se um valor *pertence* ( $\in$ ) a um determinado conjunto. Podemos facilmente criar uma função para a tarefa. Mas para a leitura poderia ficar mais fácil termos um operador e escrever  $v \in l$  do que uma função do tipo `pertence v l`. Você pode trocar o  $\in$  por um símbolo de acesso mais fácil pelo teclado como o Euro (Alt gr + e). Ficaria algo como:

```

red>> pertence: func [v l][
[      t: append copy [] v
[      t = intersect t l
[      ]
== func [v l][t: append copy [] v t = intersect t l]
red>> pertence 3 [1 3 5 7]
== true
red>> pertence 3 [2 4 6 8]
== false
red>> ∈: make op! :pertence
== make op! [[v l]]
red>> 3 ∈ [2 4 6 8 3]
== true
red>> 3 ∈ [1 2 4 6 99]
== false
red>>

```

Definimos uma função `pertence` e criamos o operador  $\in$ .

### 5.23 pair!

É uma estrutura para armazenar pares ordenados. Atualmente apenas inteiros. Para criar um par com os valores 4 para x e 5 para y entramos com: `pt: 4x5`. Para acessar o x podemos digitar `pt/x` ou `pt/1`. Para o y podemos digitar `pt/y` ou `pt/1`. Assim poderemos alterar os valores individualmente. Para colocar 6 no x basta entrar `pt/x:6`.

Também são permitidas operações sobre os pares. Um número afetar ambos os valores  $2 \times 5 + 2 \rightarrow 4 \times 7$ . Se o operador for um par, irá afetar os valores individualmente  $2 \times 5 + 2 \times 1 \rightarrow 4 \times 6$ .

## 5.24 paren!

Basicamente é um bloco delimitado por parênteses e deve ser avaliado antes.

```
red>> a: [12 34 (44 + 66) 77]
== [12 34 (44 + 66) 77]
red>> type? a
== block!
red>> type? a/1
== integer!
red>> type? a/3
== paren!
red>> a/1
== 12
red>> a/3
== (44 + 66)
red>>
```

## 5.25 path!

Informa o caminho para algum lugar. Pode ser um subdiretório, um componente de um objeto, um caractere de uma string, etc.. É definido separando os elementos por uma barra. Por exemplo.

```
red>> caminho: 'a/b
== a/b
red>> type? caminho
== path!
red>> insert :caminho 'aqui
== a/b
red>> caminho
== aqui/a/b
red>> append :caminho 'la
== aqui/a/b/la
red>> find :caminho 'b
```

```

== b/1a
red>> caminho/2
== a
red>> a: 3x4
== 3x4
red>> a/x
== 3
red>> a/y
== 4
red>>

```

## 5.26 percent!

Uma forma fácil de trabalhar com percentuais. Para que um valor seja entendido como percentual, basta colocar o percentual como sufixo. O valor é dividido por 100 interna e automaticamente. Assim, para calcular o percentual de um determinado valor, basta multiplicá-lo pelo percentual  $120 * 10\% \rightarrow 12$  ou se desejamos saber de qual valor é um percentual, basta dividi-lo  $12 / 10\% \rightarrow 120$ .

Imediatamente pensamos que, para adicionar um percentual a um valor basta entrar com  $120 + 10\% \rightarrow 132$ . Sinto informar que, pelo menos no atual estágio de desenvolvimento, **não**. Mas calma, nem tudo está perdido. Colocando o tico e o teco para funcionar chegamos a uma solução, no mínimo, elegante. Podemos redefinir o operador `+` da seguinte forma:

```

red>> oldadd: :+
== make op! ["Returns the sum of the two values" value1 [number! cha...
red>> newadd: func [a b][
[   either percent? b [
[   return a oldadd (a * b)]]
[   return a oldadd b]]
== func [a b][either percent? b [return a oldadd (a * b)] [return a o...
red>> newadd 120 10%
== 132.0
red>> +: make op! :newadd
== make op! [[a b]]
red>> 120 + 10%
== 132.0
red>>

```

Pronto. É uma das formas de tratar com o problema. Da mesma forma, poderíamos dar um desconto (diminuir um percentual do valor) redefinindo o operador `-`.

### 5.27 point!

TBD

### 5.28 refinement!

TBD

### 5.29 routine!

TBD

### 5.30 set-path!

TBD

### 5.31 set-word!

TBD

### 5.32 string!

TBD

### 5.33 tuple!

Um grupo de três ou quatro valores entre 0 e 255 separados por ponto. Podem designar cores no formato `r.g.b` ou endereços de IP, por exemplo. Se voce entrar no REPL com `? tuple!` terá o retorno das variáveis do tipo (inicialmente algumas cores definidas para facilitar).

```
red>> ? tuple!
aqua      : 40.100.130
beige     : 255.228.196
black     : 0.0.0
blue      : 0.0.255
brick     : 178.34.34
brown     : 139.69.19
```

```
...

red>> blue
== 0.0.255
red>> blue/1
== 0
red>> blue/3
== 255
red>> black
0.0.0
red>> black + 1
== 1.1.1
red>> black + 0.2.0
== 0.2.0
```

### 5.34 typeset!

TBD

### 5.35 unset!

TBD

### 5.36 url!

TBD

### 5.37 vector!

TBD

### 5.38 word!

TBD

- 6 Expressões**
- 7 Funções**
- 8 Escopo**
- 9 Operadores**
- 10 Controle de fluxo**
- 11 Exceções**
- 12 Pilha**
- 13 Depuração**
- 14 Estrutura do sistema**
- 15 Palavras reservadas**
- 16 VID**