Helpin' Red

Table of contents

Helpin'Red-pt

Tutoriais e exemplos para a linguagem de programação Red

por Ungaretti. Ainda evoluindo...

Versão 0.75 Built: 11/18/2018 3:49 PM (MM/DD/YYYY)

Você pode fazer o download do conteúdo deste site em PDF e MS-Word.
Veja a página de download.

Sugestões, colaborações e correções são bem-vindas! Use https://gitter.im/red/docs @ungaretti, ou mande uma mensagem privada lá para @ungaretti.

Downloads

Arquivo:	Tamanho:	**MD5 Hash** (veja abaixo como checar)
REDEDITOR *** **Novo!** Execute o script clicando em "play"	3428778	C9660E2CF7DC267758D2F4B107358D68
helpin.red-pt no formato PDF	-	-
helpin.red-pt no formato MS Word	-	-
helpin.red-pt HelpNDoc project	-	-

** Rededitor é um zip com executáveis (Notepad++ and Red), então pode dar problemas com firewalls e anti-virus. O hash e tamanho são para o arquivo zip.

Eu certamente não coloco malware nos meus arquivos, mas quem sabe o que um hacker pode fazer, então eu adiciono o tamanho e o hash MD5 do help app e do Notepad++ zip. Eu sei que o MD5 não é o mais seguro, mas é pequeno, e com o tamanho do arquivo deve deixar você seguro que o arquivo que você está baixando é o mesmo que eu criei. Os arquivos pdf e word não precisam de hash para segurança, e não é possível colocar o hash no o arquivo do projeto Helpndoc, pois ele mudaria assim que eu o colocasse nesta página!

Para saber o hash MD5 e o tamanho do arquivo, use o script abaixo. Ele abre um seletor de arquivos gráfico, então é bem fácil de usar.

```
Red []
a: request-file
prin "Hash= " print checksum a 'MD5
prin "Tamanho= " print size? a
```

Você pode até digitar no console:

```
>> b: request-file                                ; o seletor de arquivos
gráfico abre aqui
== %/C/Users/André/Documents/mytestfile.txt

>> print checksum b 'MD5
#{E054964EFB5ECAA5BF89164B988A62F7}
```

```
>> print size? b
2574
```

Introdução

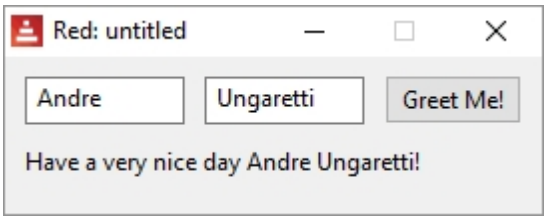
Sobre Red

- Red é uma linguagem de programação que cabe em um único executável com cerca de 1MB. Sem instalação, sem setup
- Red é gratuito e open-source.
- Red é interpretado, mas você pode compilar seu código para gerar executáveis standalone.
- Red executa alguma compilação antes da interpretação, então é bem rápido.
- Red é simples, sem "bloat".
- Red está em desenvolvimento, mas o objetivo é:
    - ser multi-plataforma;
    - ter ferramentas gráficas para todos os sistemas operacionais;
    - ser uma linguagem de programação *full-stack* ou seja, do mais baixo ao mais alto nível.
- Red é a evolução open-source do Rebol. Se você quer conhecer algumas das características do Red que ainda não estão disponíveis, você deve baixar e testar o Rebol, mas Red é o futuro.
- Red está sendo desenvolvido por um grupo liderado por Nenad Rakocevic.
- Recentemente, Red levantou fundos substanciais com uma ICO e foi criada a *Red Foundation* em Paris, França, então, o Red está aqui para ficar.

Uma amostra de Red:

```
Red [needs: view]
view [
    f1: field "First name"
    f2: field "Last name"
    button "Greet Me!" [
        t1/text: rejoin ["Have a very nice day " f1/text " " f2/text
"!"]
    ]
    return
    t1: text "" 200
```

```
]
```



Se você achou interessante, dê uma olhada em Short Red Examples, por Nick Antonaccio.

Sobre este trabalho:

É uma evolução do Red Language Notebook.
Eu usei HelpNDoc para desenvolver uma interface mais útil e completa.

Notas:

- Eu uso Windows, então todo esse trabalho é feito em cima deste sistema.
- Eu não sou um programador experiente em Red, aliás, eu nem sou um programador.
- Esta não é uma referência completa de Red (ainda?).
- Muitas vezes eu não uso a melhor formatação para os scripts, dê uma olhada em Red's coding style guide.
- Eu tento fazer meu trabalho original, mas parte foi copiada da documentação oficial do Red ou baseada em exemplos que encontrei em:
    - red-by-example.org por Arie van Wingerden e Mike Parr
    - mycode4fun.com.uk por Alan Brack
    - redprogramming.com por Nick Antonaccio

  Ainda, muita coisa foi obtida da comunidade, em gitter.im/red/home.

Obrigado a todos!!!

- Se você não encontrar alguma coisa na documentação do Red, você sempre pode pesquisar em www.rebol.com.

Convenções e notações

1- Sintaxe colorida

Eu acho que a sintaxe colorida ajuda muito os principiantes, pois existem tantas palavras pré-definidas em Red e o seu código é muito conciso. Sempre que possível, eu copio e colo a sintexe colorida do Notepad++[1].

```
Red []
a: "Hello"
b: 123
c: [33 "fox"]
print c
```

[1] - Para compiar e colar a sintaxe colorida do Notepad++ Eu uso um plugin chamado NppExport.

O console é representado por um fundo cinza:

```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

Quando os exemplos são dados como commandos digitados no console, eu destaco em negrito aquilo que é digitado pelo usuário. Isso evita confusão, pois algumas vezes você pode copiar e colar o texto dos exemplos e isso pode não funcionar direito.

Eu também adiciono uma linha em branco entre os comandos, para deixar mais legível, e às vezes coloco um fundo colorido para descacar coisas importantes. Isso tudo é adicionado durante a edição, portanto, cuidado ao copiar e colar.

```
>> a: make hash! [a 33 b 44 c 52]
== make hash! [a 33 b 44 c 52]
                        ;esta linha vazia não existe no console
>> select a [c]
== 52
                        ;nem esta
>> select a 'c
== 52                   ;comentários e fundos coloridos são
adicionados na edição
```

Começando

A primeira coisa, é claro, é baixar o executável do Red. Você pode achar o download aqui.

Quando você dá um duplo clique no executável, ele simplesmente abre o console (também chamado REPL) no seu desktop. Na primeira vez que você roda o executável, ele também faz algumas compilações, veja nota adiante.

Instruções para rodar scripts estão no capítulo "Hello world" - compilar e executar mas, primeiro, acho que você deve escolher um editor de texto.

Escolhendo um editor

Você pode simplesmente escrever seus scripts em um editor que salve o arquivo como um texto puro (p. ex. o Notepad), e então usar o Red para executá-lo através da linha de comando, mas isso não é muito amigável. Existem várias opções que podem deixar sua vida muito mais fácil.

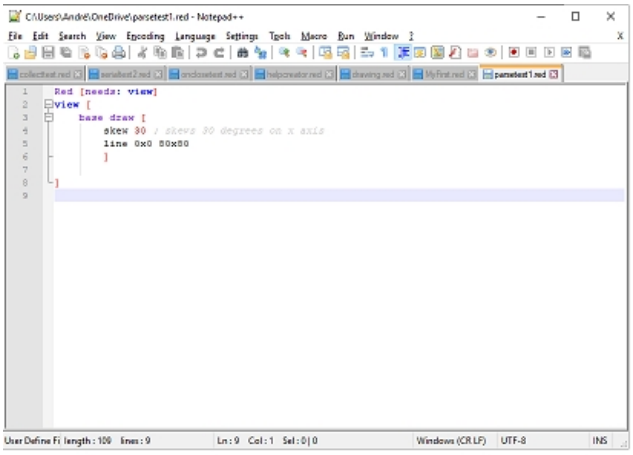
O website do Red sugere:

- Visual Studio Code com Red extension .
- Cloud9 editor, que roda no browser (setup instructions for Red).

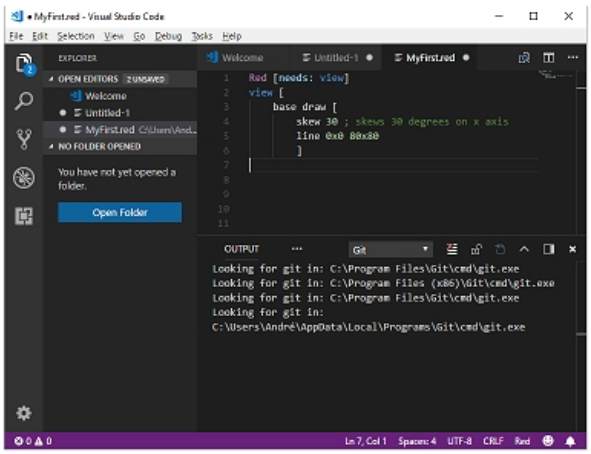
Eu acrescento o Notepad++ a estas sugestões, pois é um editor leve e muito popular. O Red se orgulha de ser um único arquivo sem instalação ou setup. Bem, se você aprecia estas qualidades do Red, você vai gostar de usar o Notepad++, especialmente se configurado como Rededitor.

Em todo este trabalho eu uso o Notepad++.

Eu também fiz um um capítulo sobre o setup do Visual Studio Code. É um editor mais sofisticado que tem muitas coisas que o Notepad++ não tem.



Notepad++



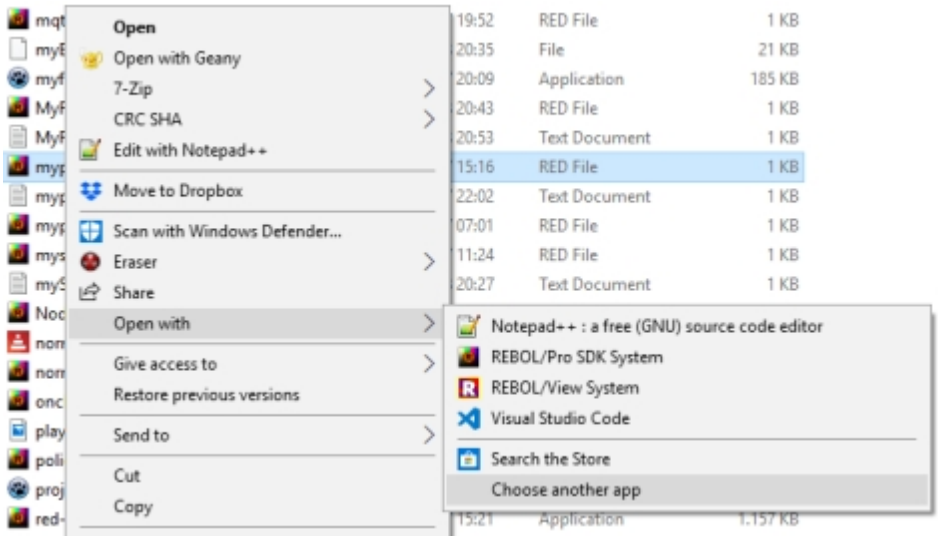
Visual Studio

Informações que podem ser úteis:

Na primeira vez que você roda o executável do Red, ele cria alguns arquivos em `C:\ProgramData\Red\` . Se você instalar um novo *release* de Red, eu aconselho você a apagar todos os arquivos que estão dentro desta pasta. Se você não fizer isso, a não ser que você especifique o *path* para o novo *release* cada vez que rodar um script, o Windows vai continuar usando o *release* velho como *default.*

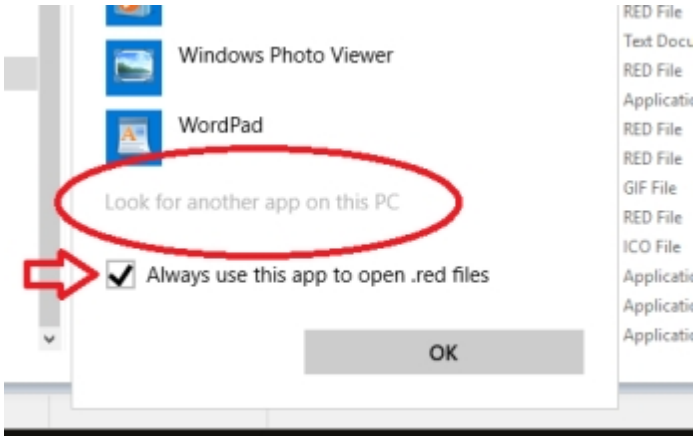
Um script de Red é um arquivo de texto puro. Ele pode ter qualquer extensão, mas é uma boa ideia dar a eles a extensão .red, já que, quando você usar editores de texto, você vai querer que eles reconheçam a linguagem que você está usando.

Você provavelmente vai querer também que o Windows associe a extensão .red ao arquivo executável Red. A maneira mais fácil de fazer isso é clicar com o botão direito em um arquivo com a extensão .red e escolher "*Open with/Choose another app"*:



Então navege até "*Look for another app on this PC"*, marque o checkbox "*Always use this*

app to open .red files" clique em "*Look for another app on this PC"* e selecione o executável Red. Todo o arquivo com a extensão .red vai ficar assim associada ao executável Red.



Rededitor

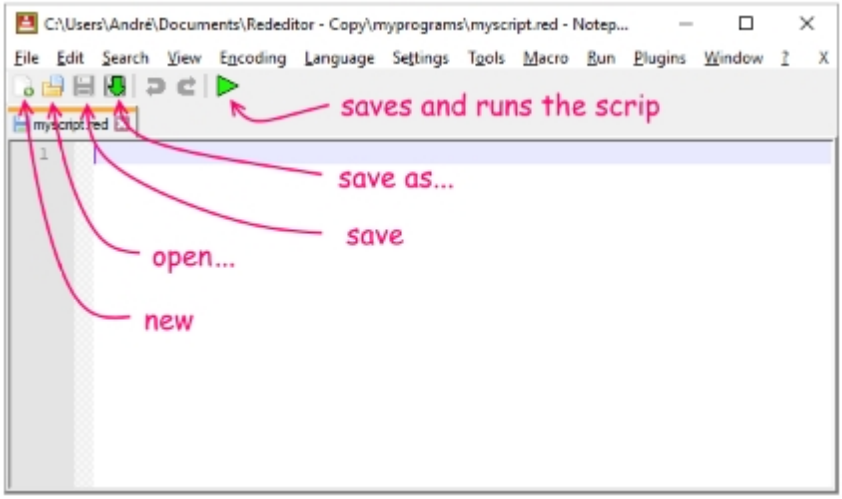
Tudo o que você precisa para começar com o Red, incluindo o próprio Red!
Basta pressionar o botão play para executar o seu script! *

*A primeira execução pode demorar um pouco enquanto o executável Red compila o console da GUI.

Depois de muita tentativa e erro com a configuração do Notepad ++, eu criei uma que é limpa, enxuta e permite que você salve e execute um script Red simplesmente pressionando o botão "play".

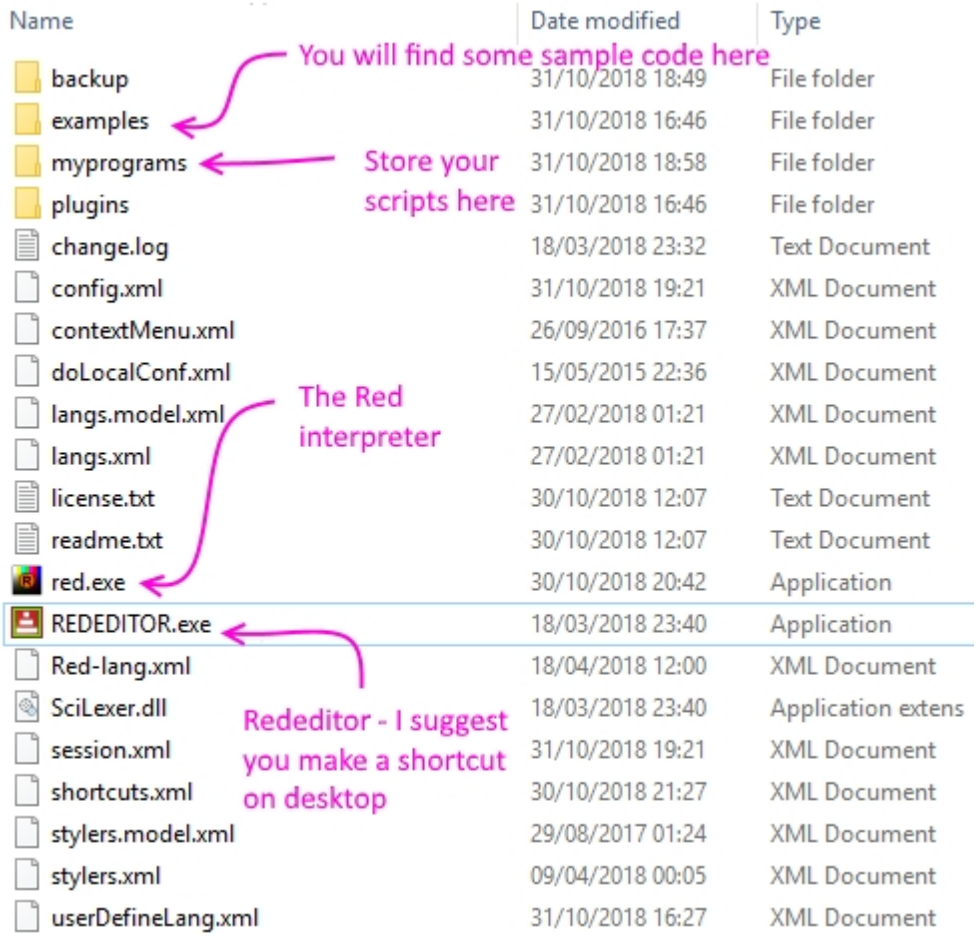
Ela tem todos os recursos interessantes do Notepad ++, além de realce de sintaxe para o Red e o pacote de plugins necessários. Tudo em um arquivo zip que já contém uma cópia do executável Red. Este zip extrai para uma pasta que é portátil e auto-suficiente, o que significa que você pode cloná-la apenas copiando e colando.

Eu chamei este pacote Rededitor. Você pode obtê-lo na página de downloads.



Eu até sugiro que você marque a caixa de seleção
Set t i ngs/ Pr ef er ences. . . / Hi de menu bar para ficar ainda mais enxuta e limpa (veja a primeira tela no topo). Você pode trazer a barra de menu de volta pressionando a tecla `alt` key or `F10`.

Após o download do zip, extraia a pasta "Rededitor". Dentro dela você encontrará:



Notas:

- Lembre-se de atualizar regularmente o interpretador Red com a última versão, renomeada para "red.exe".

- Licença Rededitor:

  Rededitor é apenas um Notepad ++ pré-configurado com 3 plugins: "Customize Toolbar", "NppExec" e "NppExport. Por favor, consulte o" license.txt "do Notepad ++ na pasta do Rededitor.

  Você pode fazer o que quiser com o Rededitor, desde que você respeite a licença do Notepad ++.

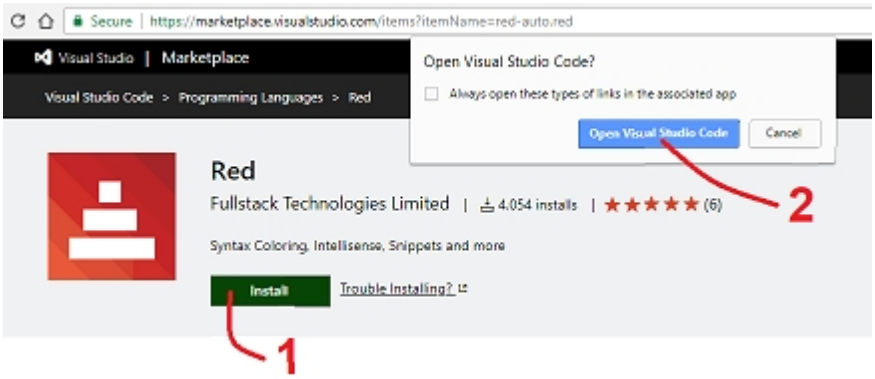
  A única mudança real feita no próprio programa (Notepad ++) foi o ícone da janela.

Setup - Visual Studio

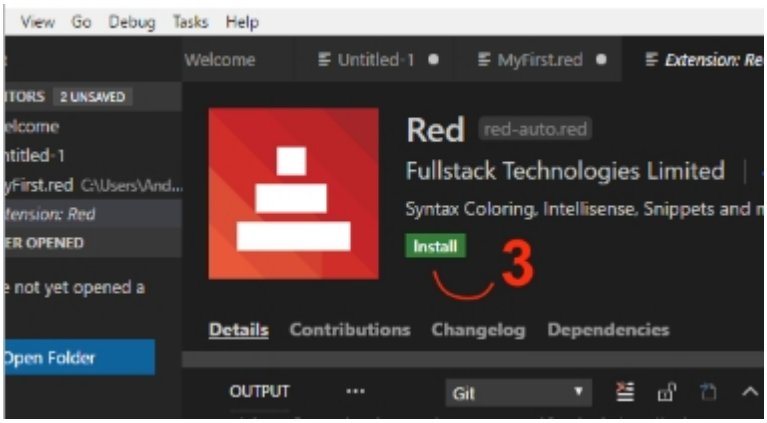
Instalar o Visual Studio com a extensão Red é bem fácil. Primeiro você tem que rodar o executável do Red pelo menos uma vez. Esta página diz que "For Windows user, need to run `red.exe --cli` first"), então, abra o command prompt e rode o Red com a opção `--cli` pelo menos uma vez antes de instalar o Visual Studio. Veja aqui como fazer isso.

Então faça o download do Visual Studio daqui e instale como qualquer outro software.

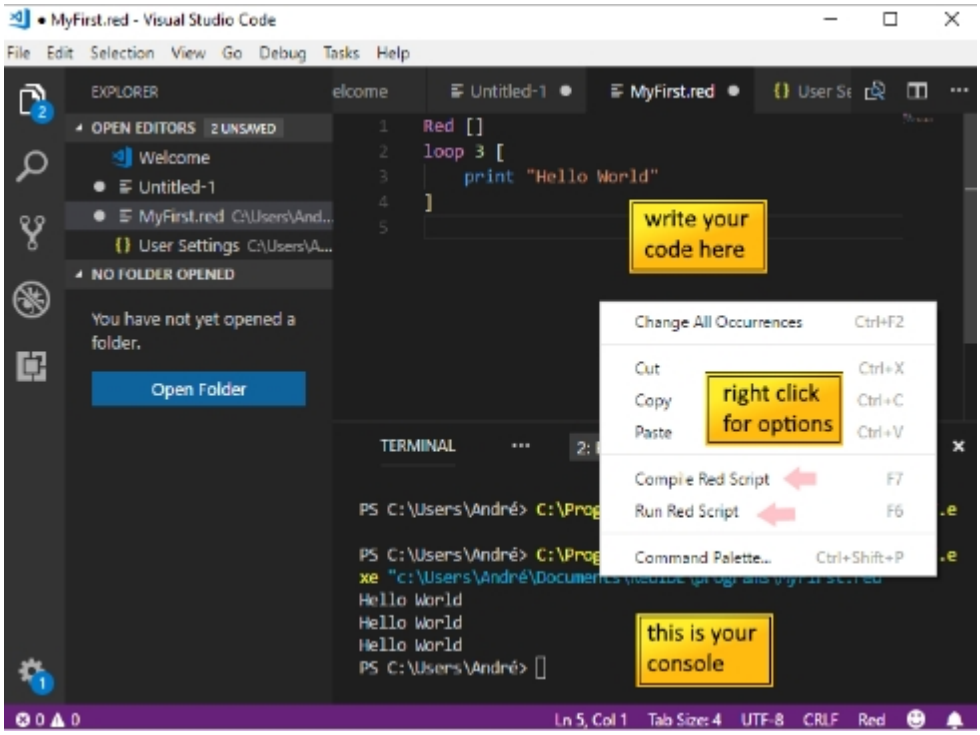
Então abra esta página (Red extension) e clique em Install. Você vai ver um prompt de "Open Visual Studio Code" . Clique nele também:

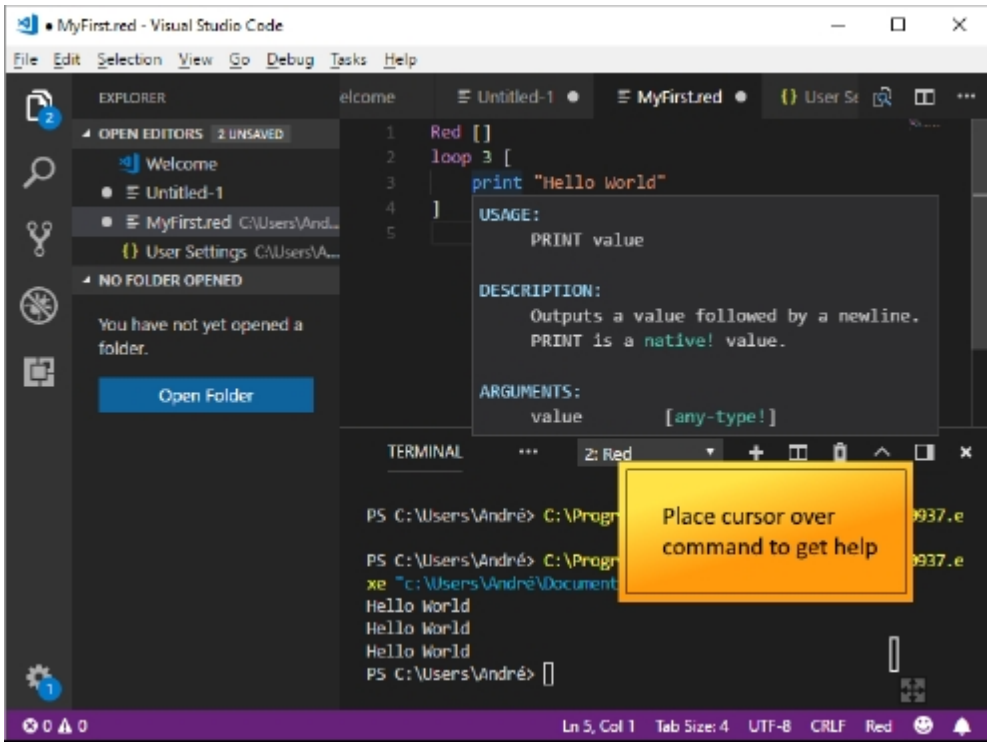


O Visual Studio vai abrir com um botão para instalar a "Red extension". Clique neste botão e... pronto! Eu tive que reiniciar o Visual Studio para que as mudanças tivessem efeito, talvez você tenha que fazer isso também.



Algumas dicas sobre como usar o Visual Studio:

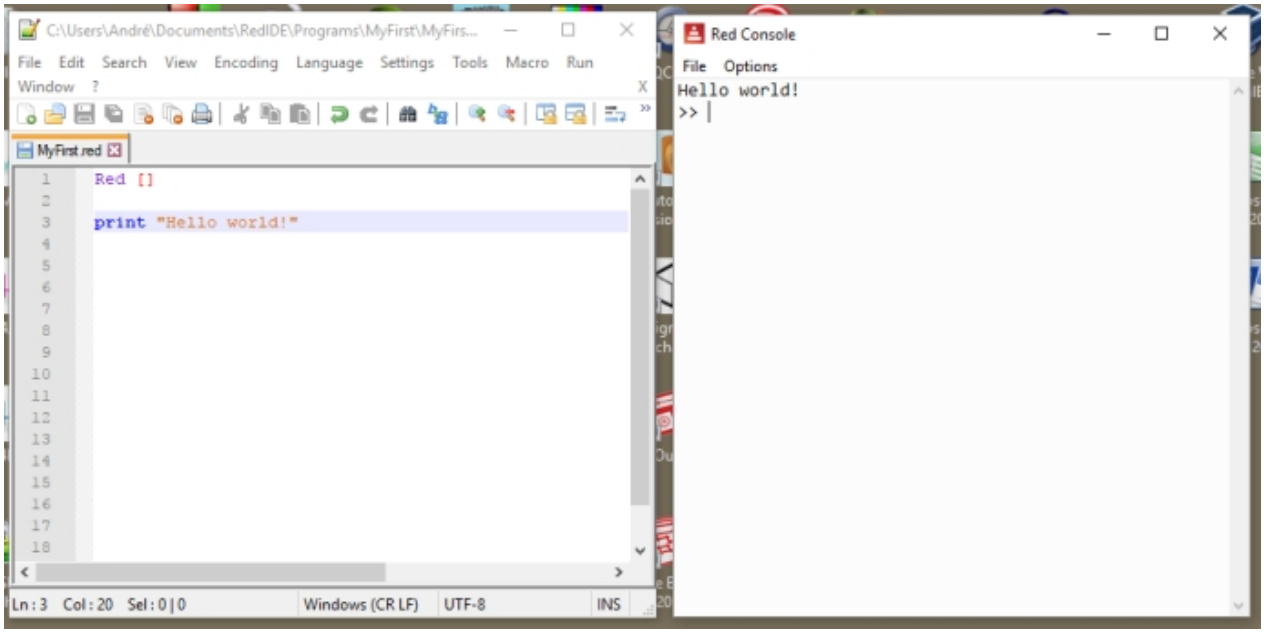

• MyFirst.red - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER
OPEN EDITORS 2 UNSAVED
Welcome
Untitled-1
MyFirst.red C:\Users\And...
User Settings C:\Users\A...
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Untitled-1
MyFirst.red
Red []
loop 3 [
print "Hello World"
]
write your code here
Change All Occurrences Ctrl+F2
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
right click for options
Compile Red Script F7
Run Red Script F6
Command Palette... Ctrl+Shift+P
TERMINAL
PS C:\Users\André> C:\Prog
PS C:\Users\André> C:\Prog
xe "c:\Users\André\Documents
Hello World
Hello World
Hello World
PS C:\Users\André>
this is your console
Ln 5, Col 1 Tab Size: 4 UTF-8 CRLF Red


• MyFirst.red - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER
OPEN EDITORS 2 UNSAVED
Welcome
Untitled-1
MyFirst.red C:\Users\And...
User Settings C:\Users\A...
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Untitled-1
MyFirst.red
Red []
loop 3 [
print "Hello World"
]
USAGE:
PRINT value
DESCRIPTION:
Outputs a value followed by a newline.
PRINT is a native! value.
ARGUMENTS:
value [any-type!]
TERMINAL
2: Red
Place cursor over command to get help
PS C:\Users\André> C:\Progr
PS C:\Users\André> C:\Progr
xe "c:\Users\André\Document
Hello World
Hello World
Hello World
PS C:\Users\André>
Ln 5, Col 1 Tab Size: 4 UTF-8 CRLF Red

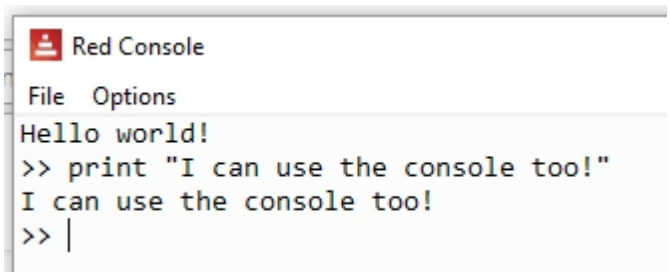
"Hello world" - compilar e executar

"Hello world" no console:

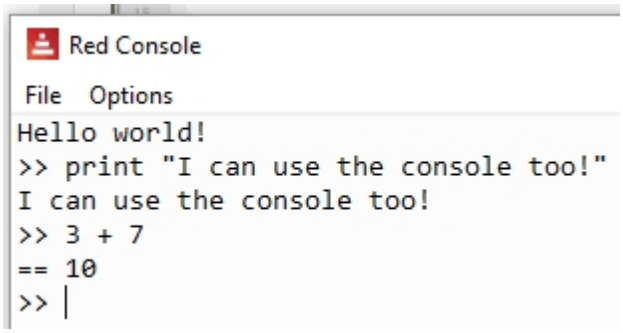
Escreva o código abaixo no Notepad++, salve como "MyFirst.red" na pasta "programs" dentro da pasta da sua IDE e execute (run/red-run). Você vai ter:



A janela a direita é o console, às vezes chamado de REPL. Clique neste console, digite `print "I can use the console too!"` e aperte *enter*:



Now type 3 + 7 and press enter:



Note que é preciso ter um espaço entre as palavras. Espaços são delimitadores no Red e sem eles você tem erros:

```
Hello world!
>> print "I can use the console too!"
I can use the console too!

>> 3 + 7
== 10

>> 3+7          ;sem espaços!!!!!
*** Syntax Error: invalid integer! at "3+7"
*** Where: do
*** Stack: load
```

Note que depois de 3+7 eu escrevi `;sem espaços!!!!!` . O Red ignora tudo que vem depois do ponto-e-vírgula, é assim que se faz **comentários** no corpo do programa.

De volta ao programa (também chamado *script*):

Linguagens interpretadas executam uma linha de código de cada vez. Programas para linguagens interpretadas são chamados "scripts". Red não é 100% interpretado, uma vez que realiza alguma compilação antes de executar, mas os programas para Red são chamados scripts de qualquer forma.

Na primeira linha nós temos `Red [ ]`. Como eu disse antes, todo script para Red tem que começar com `Red.` Não RED nem red, mas `Red`. Depois de `Red` nós temos os colchetes. Em Red, qualquer coisa dentro de colchetes é chamada de um "bloco " ("block"). Este primeiro bloco é destinado a conter informações sobre o programa. A maior parte desta informação é opcional, com algumas exceções, a mais relevante sendo a declaração de bibliotecas (mais sobre isso daqui a pouco).

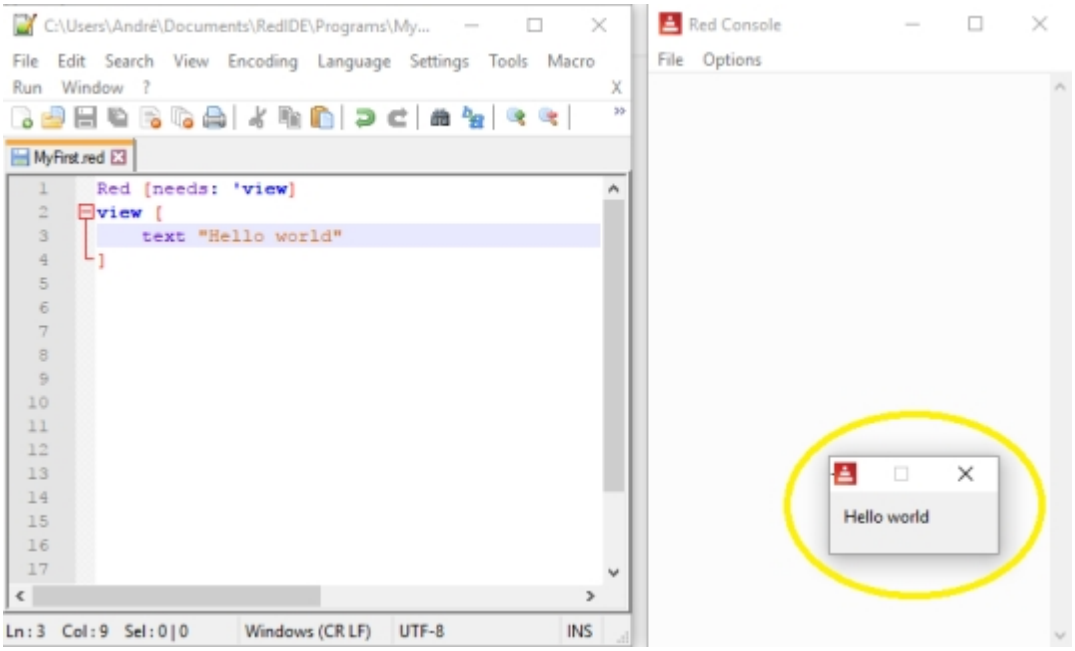
Um primeiro bloco bem completo poderia ser:

```
Red [
        title: "Hello World"
        author: "My name"
        version: 1.1
```

```
        purpose {
                To print a greeting to the planet.
                Notice that multi-line text goes
                inside curly brackets.
                }
        ]

print "Hello World!"
```

"Hello world" com interface gráfica - GUI:

Uma das características mais relevantes do Red é a sua capacidade de gerar interfaces gráficas com um código muito simples. Ele faz um uso bem inteligente das APIs do sistema operacional. Um "Hello world" com GUI seria:

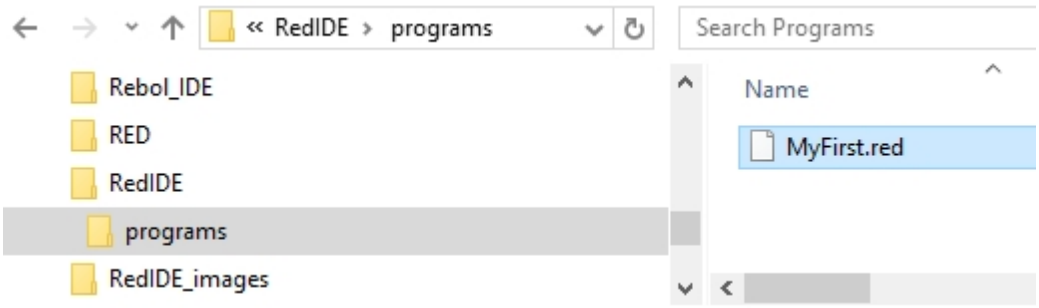


Note que eu escrevi `needs: 'view` no bloco de cabeçalho (o apóstrofe é opcional). Isso diz ao Red para carregar a biblioteca gráfica "view" . Isto não é estritamente necessário se você está usando o console gráfico, já que a biblioteca "view" é carregada automaticamente, mas é uma boa idéia deixar isso explícito sempre.

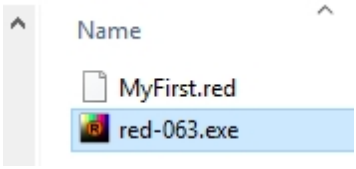
Compilando seu "Hello world" para um arquivo executável:

Voce pode criar um executável (.exe) do seu "Hello World".

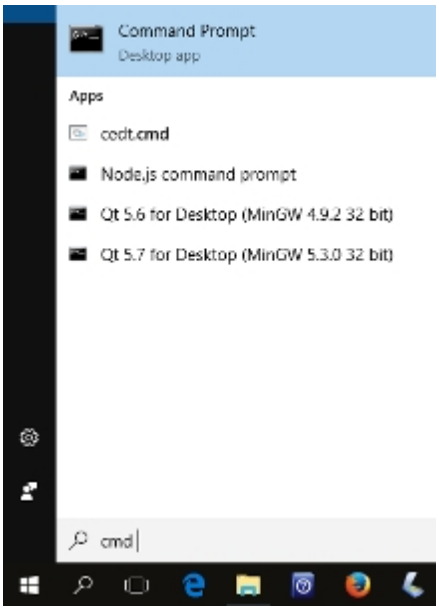
Se você salvou o seu "Hello world" (com GUI) como "MyFirst.red" na pasta "programs" você deve ter algo assim no seu computador:



Para deixar a coisa mais clara, faça uma cópia do seu executável Red e cole na mesma pasta onde está o seu programa. De outra forma, o resultado da sua compilação vai ficar na mesma parta da IDE Red, perdidos no meio de todos aqueles arquivos.



Abra o Command Prompt. Se você não sabe como, escreva "cmd" no campo de busca do Windows (lupa) e procure o ícone do Command Prompt:

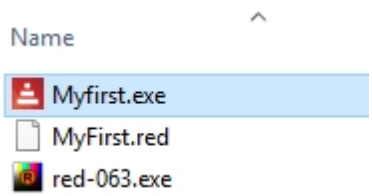


No Command prompt, digite o caminho do seu executável Red (o executável que você acabou de copiar na pasta "programs") seguido de `-r -t windows` e o nome do seu programa:

```
C:\Users\André\Documents\Rededitor\myprograms>red.exe -r -t windows Myfirst.red
```

Note: If you compile to windows, i believe you must always load the GUI library (use `needs: view`). If you just want a program that runs on CLI alone, use MSDOS as target.

O Red vai dar uma série de mensagens no Command Prompt e, depois de mais ou menos um minuto, você vai ter o executável na sua pasta "programs":



Dê um duplo clique nele e você deve ver o seu "Hello World" gráfico (GUI) na sua tela.

Mais algumas observações sobre compilação:

Eu notei que a versão compilada do programa pode não se comportar como a interpretada. Eu tive problemas com prints que eu deixei no código, assim, parece que chamar comandos do console no modo executável não é uma boa idéia. Eu também tive problemas com variáveis (palavras) de escopo global dentro de funções. O compilador parece não reconhecê-las como variáveis globais. Eu solucionei este último problema de duas maneiras::

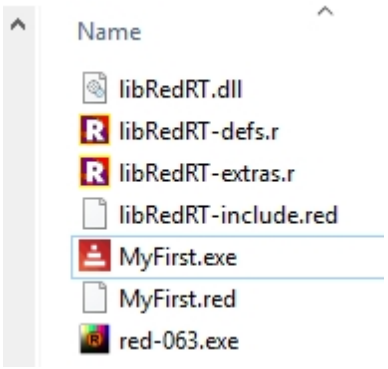
1. Eu "declarei" minhas variáveis, ou seja, eu dei um valor a elas no início do programa. Os valores não são importantes, já que vão mudar depois.

2. Eu suei a opção de compilação "-e" (que não é listada no github, parece que é experimental). -e significa "encap". Ela faz que você obtenha um executável simples, mas o código é internamente interpretado. compiler option (not listed in github, as it seems to be experimental).

Você pode também compilar o MyFirst.red usando apenas a opção `-c` (compile) :

```
C:\Users\André\Documents\Rededitor\myprograms> red.exe -c
Myfirst.red
```

Neste caso você vai obter os seguintes arquivos na sua pasta:



Os dois únicos arquivos que você precisa para rodar o programa são o **libRedRT.dll** e o **MyFirst.exe**.

Entretanto, neste caso, você vai notar que o Red mantém um Command Prompt aberto enquando roda o programa. Se você quiser evitar isso (quem não quer?) use a opção `-t` . Esta opção compila para uma plataforma específica:

```
C:\Users\André\Documents\Rededitor\myprograms> red.exe -c -t
```

```
windows Myfirst.red
```

O resultado vai ser aquele mesmo conjunto de arquivos, incluindo a dll, mas o Command Prompt não fica mais aberto durante a execução.

Você deveria poder compilar para todas as plataformas listadas abaixo, mas como o Red ainda está evoluindo, algumas ainda apresentam problemas. Por exemplo, a plataforma Android ainda não é suportada.

Do github do Red:

Cross-compilation targets:

```
MSDOS        : Windows, x86, console (+ GUI) applications
Windows      : Windows, x86, GUI applications
WindowsXP    : Windows, x86, GUI applications, no touch API
Linux        : GNU/Linux, x86
Linux-ARM    : GNU/Linux, ARMv5, armel (soft-float)
RPi          : GNU/Linux, ARMv5, armhf (hard-float)
Darwin       : macOS Intel, console-only applications
macOS        : macOS Intel, applications bundles
Syllable     : Syllable OS, x86
FreeBSD      : FreeBSD, x86
Android      : Android, ARMv5
Android-x86  : Android, x86
```

Compiler options:

```
-c, --compile                : Generate an executable in the working
                               folder, using libRedRT. (developement
mode)
-d, --debug, --debug-stabs   : Compile source file in debug mode. STABS
                               is supported for Linux targets.
-dlib, --dynamic-lib         : Generate a shared library from the source
                               file.
-h, --help                   : Output this help text.
-o <file>, --output <file>   : Specify a non-default [path/][name] for
                               the generated binary file.
-r, --release                : Compile in release mode, linking
everything
                               together (default: development mode).
-s, --show-expanded          : Output result of Red source code
expansion by
                               the preprocessor.
```

```
-t <ID>, --target <ID>         : Cross-compile to a different platform
                                 target than the current one (see targets
                                 table below).
-u, --update-libRedRT          : Rebuild libRedRT and compile the input
script
                                  (only for Red scripts with R/S code).
-v <level>, --verbose <level>  : Set compilation verbosity level, 1-3 for
                                 Red, 4-11 for Red/System.
-V, --version                  : Output Red's executable version in x.y.z
                                 format.
--config [...]                 : Provides compilation settings as a block
                                 of `name: value` pairs.
--cli                          : Run the command-line REPL instead of the
                                 graphical console.
--no-runtime                   : Do not include runtime during Red/System
                                 source compilation.
--red-only                     : Stop just after Red-level compilation.
                                 Use higher verbose level to see compiler
                                 output. (internal debugging purpose)
```

Também tem a opção `-e` . Veja a observação lá em cima.

Rodando o Red no console do Windows (cmd):

Para executar o Red no modo linha de comando, abra o cmd prompt, mude o diretório para a pasta onde você tem o seu executável do Red e digite o nome deste executável seguido de `--cli` . Note que são dois traços. Eu tenho `red-063.exe`, então:

```
 C:\Users\André\Documents\RedIDE>red-063.exe --cli
--== Red 0.6.3 ==--
Type HELP for starting information.
>>
```

Passando argumentos para um script Red:

Tudo que vem depois no nome do script na linha de comando é passado para o script como argumentos. Estes argumentos são guardados em *system/options/args* como um bloco.

Este script foi salvo com o nome de "arguments.red":

```
Red []
probe system/options/args
```

Executado da linha de comando (CLI):

```
C:\Users\André\Documents\RedIDE\programs>red-063.exe arguments.red foo
boo loo
```

O output do script no console do Red é:

```
["foo" "boo" "loo"]
>>
```

Ajuda do sistema

Red tem um ótimo help embutido no próprio programa. Tem uma grande quantidade de informações que você pode opter sobre a linguagem e sobre o seu código simplesmente digitando alguns comandos no console.

function! ? (ou help)

Dá informação sobre as palavras pré-definidas do Red e também sobre o seu próprio programa. Você também pode digitar `help`, mas `?` é, claro, mais curto. `?` por sí só dá informações sobre como usar o help.

```
>> ? now
USAGE:
     NOW

DESCRIPTION:
     Returns date and time.
     NOW is a native! value.

REFINEMENTS:
     /year        => Returns year only.
     /month       => Returns month only.
     /day         => Returns day of the month only.
     /time        => Returns time only.
     /zone        => Returns time zone offset from UCT (GMT) only.
     /date        => Returns date only.
     /weekday     => Returns day of the week as integer (Monday is day
1).
     /yearday     => Returns day of the year (Julian).
     /precise     => High precision time.
     /utc         => Universal time (no zone).

RETURNS:
     [date! time! integer!]
```

```
>> a: [1 2 3]
== [1 2 3]

>> help a
A is a block! value: [1 2 3]
```

```
>> a: function [a b] [a + b]
== func [a b][a + b]

>> ? a
USAGE:
    A a b
DESCRIPTION:
    A is a function! value.
ARGUMENTS:
    a
    b
```

Você pode obter informação sobre objetos complexos:

```
Red [needs: 'view]

a: view/no-wait [
button
]
? a
```

```
Red Console
File  Options
     type              word!          window
     offset            pair!          637x387
     size              pair!          130x45
     text              string!        "Red: untitled"
     image             none!          none
     color             none!          none
```

Se você não sabe exatamente o que está procurando, "`?`" faz uma busca para você:

```
>> ? -to
     hex-to-rgb      function!     Converts a color in hex format to a
tuple value; returns NONE if it f...
     link-sub-to-parent function!     [face [object!] type [word!] old
new /local parent]
     link-tabs-to-parent function!     [face [object!] /init /local
faces visible?]
```

Você pode achar todas as palavras pré-definidas de um determinado datatype! :

```
>> ? tuple!
    Red               255.0.0
    white             255.255.255
    transparent       0.0.0.255
    black             0.0.0
    gray              128.128.128
    ; ... the list is too long!
```

function! what

Imprime uma lista de todas as palavras pré-definidas. Tente!

function! source

Mostra o código fonte das funções *mezzanine* e das funções criadas pelo usuário.

tente `source replace` .

funções *mezzanine*

O interpretador Red tem:

- as funções nativas que fazem parte do interpretador e são executadas em um nível baixo;
- as funções *mezzanine* que, apesar de fazerem parte do interpretador Red (vem junto com o executável_ foram criadas usando o Red, quer dizer, tem um código fonte que você pode ler usando `source`.

function! about

Lista o número da versão e a data do *build*.

Notas sobre sintaxe

- Letras maiúsculas ou minúsculas são indiferentes para o Red, mas existem algumas exceções, a mais relevante é que um programa precisa começar com **Re**d (não REd nem red).
- Caracteres de nova linha `new-line` são ignorados pelo Red, uma exceção é um `new-line` dentro de uma string.

Nota da tradução: Não consegui encontrar em português uma palavra que traduzisse o exato sentido de "evaluate" no contexto do processamento de dados como é feito pelo Red: não só com números mas com palavras e rotinas. Assim vou usar a palavra "computação", que não me parece perfeita mas...

- Red é uma linguagem funcional, significando que ela avalia, ou computa (*evaluates*) resultados. A ordem desta computação (ordem de execução das funções) não é usual e é bom você dar uma olhada no capítulo Ordem de computação.

(os próximos tópicos podem não ser exatos, mas até agora explicam o funcionamento do Red satisfatoriamente)

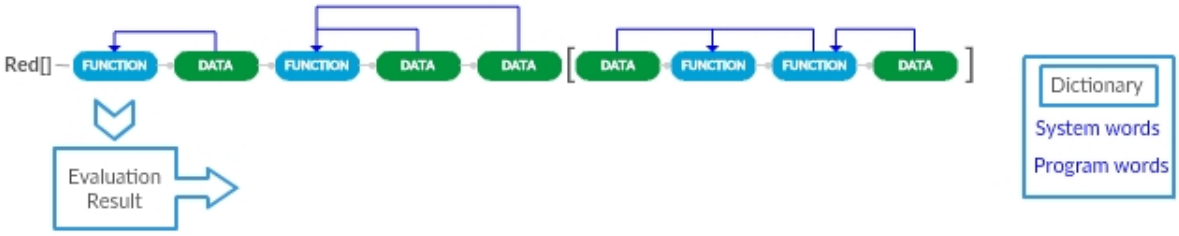
- um script de Red é uma longa cadeia de "palavras" ("words"). Estas palavras podem estar associadas a dados ou ações.
- "palavras" são separadas por um ou mais espaços em branco.
- o Red mantém um dicionário com as palavras pré-definidas e as palavras criadas pelo usuário.
- "palavras" podem ser agrupadas em "blocos" delimitados por colchetes ("[" e "]"). Blocos não são necessariamente rotinas, são só grupos de "palavras" que podem ou não ser "avaliados" por uma ação.
- todos os dados do programa estão dentro do próprio programa. Se são acrescentados dados externos, estes são adicionados à lista de "palavras" do programa.
- toda palavra tem que ter um valor quando avaliada. Este valor pode vir:
  - da computação, se a palavra for associada a uma ação;
  - da própria palavra, quando esta for associada a um dado;
  - de outra palavra ou bloco. Isto é feito com o símbolo de atribuição, que é dois-pontos (":"), seguido pela palavra ou bloco que se quer associar (por exemplo: `meuQuarto: 33`).
- Me parece que em Red, você pode dizer que **a variável é atribuida aos dados, e não o contrário**. **Na verdade, não existem "variáveis" em Red, só palavras que**

são associadas a dados.

- Copiar palavras (variáveis) em Red requer muito cuidado. Quando você quiser fazer uma cópia realmente independente de uma palavra (variável), você deve usar a palavra pré-definida (comando) `copy` . Veja o capítulo Copiando.
- Assim como "copiar", "limpar" uma série (note que todas as strings são séries) também requer cuidado. Simplesmente atribuir "" (string vazio) o zero à série pode não produzir os resultados esperados. A lógica interna do Red faz com que ele pareça "lembrar" de coisas de uma forma inesperada. Então, para limpar uma série, você deve usar a palavra pré-definida `clear`.
- Toda palavra tem um *datatype*. Red tem um número notavelmente grande de *datatypes*. Eles estão listados no capítulo Datatypes . O nome de um *datatype* é sempre seguido por um ponto de exclamação.
- Quando uma palavra é criada pela primeira vez, ela tem o *datatype* **word!** que é usado assim:

Notação	Significado
palavra	Obtém o valor natural da palavra. (se o valor for uma função, avalia a função, senão, retorna o valor).
palavra:	Associa o valor da palavra (como atribuição) a um valor.
:palavra	Obtém o valor da palavra sem computar o resultado (Útil par obter o valor de uma função)
'palavra	Trata a palavra como um valor em si própria (um símbolo). Não avalia o seu valor.
/palavra	Trata a palavra como um refinamento. Usado para argumentos opcionais.

Uma visão um tanto simplificada do fluxo do Red:



Nota: A função que obtém dados anteriores a ela (a terceira da direita para a esquerda) represnta um operador infixo, como "+", "-" , "*" , "/" etc.

Refinamentos

Muitas ações em Red admitem "refinamentos". Um refinamento é declarado adicionando "/<refinamento>" ao comando (palavra pré-definida) e modifica seu funcionamento.

Comentando o código:

Todo o texto depois de um ponto-e-vírgula (;) em uma linha é ignorado pelo interpretador. Também existe a função interna `comment` . Um grupo de words após `comment` também será ignorado pelo interpretador. Esse grupo de palavras deve estar dentro de `" "`, `{ }` ou `[ ]` .

Eu também notei que qualquer texto escrito no código-fonte antes do "prólogo" do Red (`Red [ ... ]`), no início, também é ignorado pelo interpretador, mas não tenho certeza se essa é uma maneira segura de adicionar informação ao código.

Exemplos de comentários:

```
Me parece que qualquer coisa escrita aqui, antes do código,
é ignorada.

Red [                                ; Aqui começa o prólogo
  Author: "Ungaretti"                ; você pode adicionar comentários
depois de ";"
  Date: "september 2018"             ; mas apenas na mesma linha.
  Purpose: "to show how to comment the code"
]

; Um prólogo deve ser informativo

comment [ Este é um comentário multi-linhas
usando o colchetes.]

print "End of first comment."

comment " This is a comment."        ; se você usa aspas o comentário
                                     ; é limitado a uma linha

print "End of second comment."

comment { Essa me parece a melhor maneira
de fazer comentários multi-linhas: usando
comment e chaves}

print "End of third comment."

{estranhamente, o interpretador parece ignorar texto
dentro de chaves, mesmo sem o uso do comando "comment".
Me parece bem elegante, mas cuidado, não vi nada na
documentação sobre isso}

print "End of the fourth, strange, comment."
```

```
End of first comment.
End of second comment.
End of third comment.
End of the fourth, strange, comment.
```

Usando *words*

Já que um programa Red é uma série de palavras (words), é uma boa ideia dar uma olhada nelas.

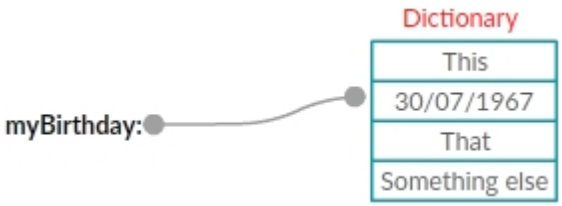
word

Uma palavra por si só (não um dado) não significa muito para o Red. Cada palavra deve ter um valor associado a ela enquanto computada. Esse valor pode vir da computação de uma expressão ou do "dicionário". Neste último caso, pode ser dados ou ação.

```
>> myBirthday
*** Script Error: myBirthday has no value
```

word:

Os dois pontos depois de uma palavra a associa a algo no dicionário. É a "atribuição" clássica de outras linguagens de programação. A propósito, esta é um `set-word!` datatype.



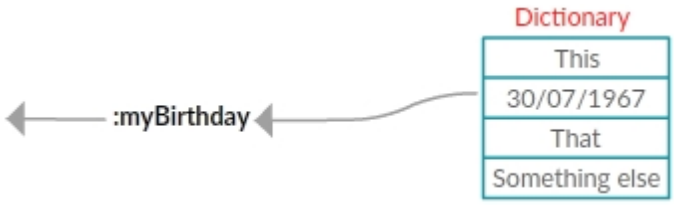

```
>> myBirthday: 30/07/1963
== 30-Jul-1963
>> print myBirthday
30-Jul-1963
```

As palavras podem estar associadas ao código (ação) também:

```
>> a: [print "hello"]
== [print "hello"]
>> do a
hello
```

:word

Os dois pontos antes de uma palavra faz com que esta retorne o que estiver associado a ela no dicionário sem qualquer avaliação. Valores e ações são retornados "como estão". Esse é um `get-word!` datatype.

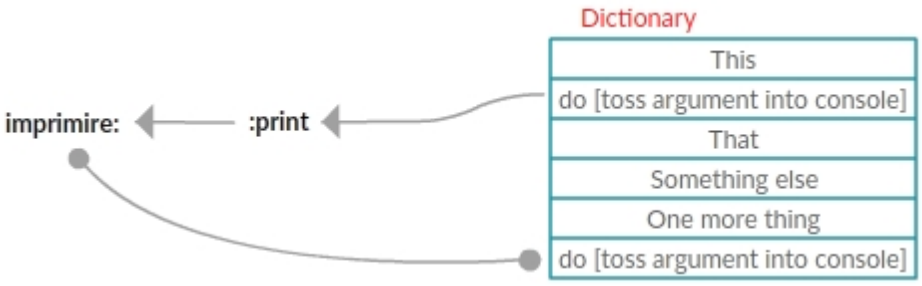



```
>> myBirthday: 30/07/1963
== 30-Jul-1963
>> partyDay: :myBirthday
== 30-Jul-1963
>> print partyDay
30-Jul-1963
```

Se uma palavra estiver associada a uma ação, dois pontos antes dela retornam o código inteiro dessa ação. Isso cria uma situação interessante se você usá-la com as funções internas do Red:

```
>> imprimire: :print
== make native! [[
    "Outputs a value followed by a newline"
    ...
>> imprimire "hello"
hello
```

O que aconteceu acima é que "imprimire" agora tem a mesma funcionalidade de `print` . Algo mais ou menos assim:



Important notes:

- a sintaxe `:word` também é usada para acessar dados em uma série, conforme descrito em Blocos & Series;
- se você redefinir funções internas do Red, poderá causar uma falha, não por causa da alteração em si, mas porque todas as outras funções internas que dependem do

significado original dessa palavra podem não funcionar corretamente.

'word

Retorna a própria palavra, isto é: apenas um grupo de letras (mas não uma string! Apenas um símbolo). É um `lit-word!` datatype.

```
>> print something
*** Script Error: something has no value
*** Where: print
*** Stack:

>> print 'something
something
```

```
>> type? :print
== native!
>> type? 'print
== word!
```

/word

A barra antes de uma palavra a transforma em um refinamento. Obviamente, esse é um `refinement!` datatype.

native! set

Atribui um valor a uma palavra. Parece-me como sendo o mesmo que os dois pontos depois da palavra ...

```
>> set 'test 33
== 33
```

... exceto que você pode definir muitas palavras de uma só vez:

```
>> set [a b c] 10
== 10
>> b
== 10
```

native! unset

A definição prévia de uma palavre pode ser desfeita a qualquer momento usando `unset`:

```
>> set 'test "hello"
== "hello"
>> print test
hello
>> unset 'test
>> print test
*** Script Error: test has no value
```

Avaliação (computação)

Há uma boa descrição da avaliação do Rebol aqui . É praticamente a mesma coisa para o Red. Não vou repetir essa explicação, em vez disso, vou descrever como vejo a avaliação de Red do meu ponto de vista pessoal. Novamente, isso pode ser impreciso, mas até agora explica muito bem o comportamento do Red.

Red, o avaliador furioso!

Uma vez acionado, o Red começará a ler um texto da esquerda para a direita (→) executando todas as operações que puder encontrar. Se ele encontrar uma operação que requer argumentos, ele selecionará os argumentos desse texto principal conforme necessário para chegar a um valor final. Dê uma olhada no conceito de grupos avaliáveis e escolha de argumentos . Red considera texto (strings) como um bloco de caracteres, então este texto principal do código é apenas um grande bloco para o Red, mesmo sem colchetes ou aspas.

O que dispara a fúria do Red?

Red é acionado pelo "commando" `do`. Você nem sempre tem que realmente digitá-lo, quando você executa um script ou pressiona enter no console, o que está acontecendo é que você está aplicando `do` implícito no texto à frente. No caso de um script, a avaliação só começa depois que o interpretador encontra os caracteres "`Red [`"

Uma consequência interessante de tudo isso é que, embora geralmente não seja considerada boa prática, você pode realmente executar o texto:

```
>> do "3 + 5"
== 8

>> 3 + 5 ;same thing. The "do" is implicit and input is text (but not a string! datatype).
== 8
```

Se é uma computação, qual é o resultado?

O resultado de uma computação em Red é o valor resultante do último grupo avaliável. É claro que você pode fazer todo tipo de coisas interessantes ao longo do caminho, como escrever arquivos, ler páginas da web e criar belos desenhos na tela, mas o valor retornado pelo Red (se houver um) é esse último resultado.

```
>> do "3 + 5 7 * 8 print 69"
69
```

O que detém a fúria de Red?

O final do texto (código) e os comentários, é claro.

Mas a avaliação do Red também pula blocos (blocks) dentro do texto principal, deixando-os como estão. Só os avalia se são o argumento de uma operação, observando que esta operação pode ser outro `do`:

```
>> do {print "hello"  7 + 9  [8 + 2]} ; the last result is the
unevaluated block
hello
== [8 + 2]

>> do {print "hello"  7 + 9  print [8 + 2]}
hello
10

>> do {print "hello"  7 + 9  do[8 + 2]}
hello
== 10
```

Você descobrirá que, para desenvolver scripts Red, às vezes precisará dos valores resultantes de todos os grupos avaliados em um bloco, não apenas do último. Você pode conseguir isso com `reduce`. Ele retorna um bloco com todos os resultados. No entanto, não é como se você aplicasse um `do` para cada grupo avaliável dentro do bloco, como você pode ver aqui:

```
>> reduce [3 + 5 7 * 8 print 69]
69
== [8 56 unset]

>> reduce [3 + 5 7 * 8 "print 69"] ; do "print 69" should print 69!
== [8 56 "print 69"]
```

Ordem de avaliação matemática:

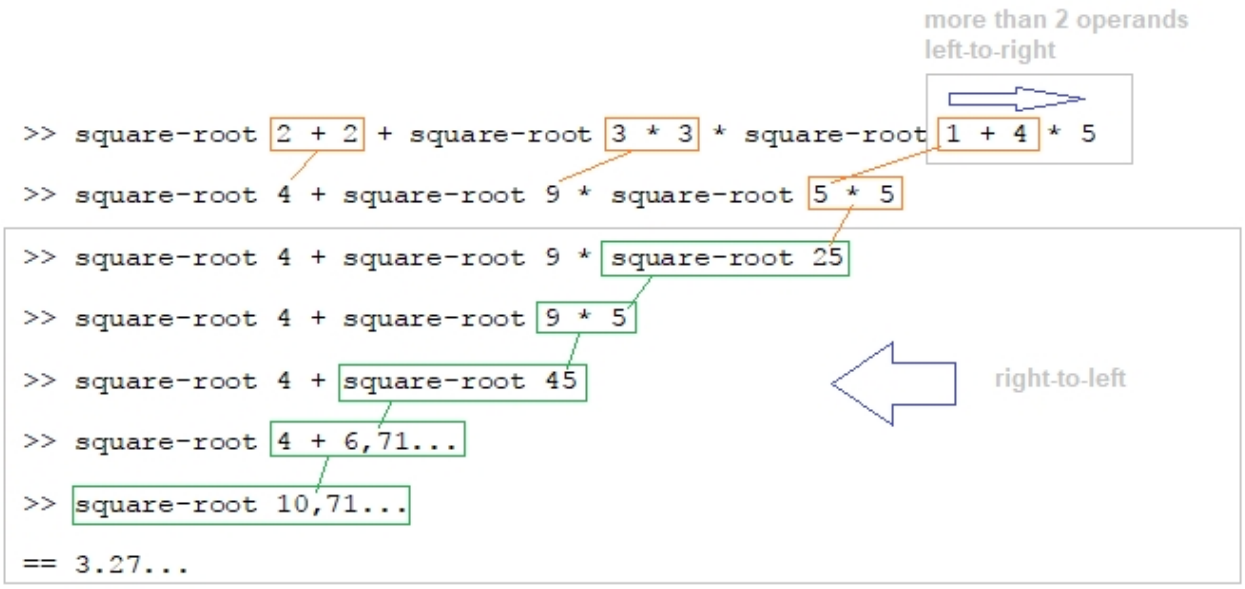
Eu ainda estou procurando uma regra simples para explicar o processo de computação do Red. No momento, eu tenho duas candidatas favoritas. A primeira é bem direta e fácil de usar. A segunda não é muito prática, mas dá uma boa visão de como (eu acho) que o interpretador "pensa", e eu acho que é uma boa ideia dar uma olhada nela para captar alguns conceitos que podem ser úteis

1) Minha regra favorita no momento:

1- Todas as operações com operadores infixos que têm apenas valores (não funções) como operandos, são executadas primeiro. Se estes operadores infixos tem mais de dois operandos, eles são avaliados (resolvidos) da esquerda para a direita, sem precedência, ou seja, por exemplo,a multiplicação não é automaticamente feita antes da soma.

2- Então toda a expressão é calculada da direita para a esquerda (←).

```
>> square-root 2 + 2 + square-root 3 * 3 * square-root 1 + 4 * 5
== 3.272339214155429
```



2) Minha segunda favorita, a explicação dos 3 conceitos:

Parece funcionar, e eu acho que, de alguma forma, é parecida com o que o iterpretador realmente faz.

Não é uma simples regra pode não ser formalmente correta, uma vez que não tenho certeza que todo operador infixo tem uma função correspondente.

Conceito 1: Esquerda para a direita sempre →

Em Red, as coisas são avaliadas (resolvidas) da esquerda para a direita. Não existe ordem de precedência, ou seja, por exemplo, a multiplicação não é necessáriamente feita antes da soma. Se você quiser forçar uma precedência, tem que usar parênteses.

```
>> 2 + 3 * 5
== 25 ; não 17!
```

Não apenas as expressões, mas todo o código do programa é avaliado da esquerda para a direita.

> **Operadores infixos**

> "+", "-", "*", "/" são chamados operadores infixos. Eles correspondem às funções `add` , `multiply`, `divide` e `subtract`, que precisam de dois argumentos. Então:

> `3 + 2` é o mesmo que `add 3 2`

> `5 * 8` é o mesmo que `multiply 5 8` ...

> ...e assim por diante.

> `2 + 3 * 5` é só uma forma mais legível de `multiply add 2 3 5` . O interpretador Red faz a conversão para você.

Conceito 2: Grupos avaliáveis (computáveis). (N.T. Tradução horrível de "evaluable groups")

Quando você tem um pedaço de código, existem grupos de palavras que são "avaliáveis" (resolvíveis,l computáveis), isto é, podem ser reduzidos a um *datatype* básico. Por exemplo `[square-root 16 8 + 2 8 / 2 77]` é, na verdade, composto de 4 grupos avaliáveis: `square-root 16` ; `8 + 2` ; `8 / 2` and `77`. Você pode user `reduce` para "ver" os valores dos quatro grupos:

```
>> a: [ square-root 16 8 + 2 8 / 2 77]
```

```
a: [ square-root 16 8 + 2 8 / 2 77]
```

```
>> reduce a
== [4.0 10 4 77]
```

Conceito 3: Funções pegam seus argumento dos grupos avaliáveis

Uma função pega seus argumentos dos grupos avaliáveis à sua frente, da esquerda para a direita (pense nos operadores infixos como suas funções correspondentes). Uma função que precisa de 1 argumento, pega o próximo grupo avaliável; uma função que precisa de 2 argumentos, pega os dois próximos grupos avaliáveis, e assim por diante. Note que uma função pode usar um grupo avaliável que tem outra função. Neste caso, a primeira função suspende o seu processamento até que a segunda função seja avaliada, e aí usa o resultado.

Mais uma vez, sem regras de precedência, simplesmente esquerda para direita.

A consequência disso é que uma expressão como...

```
square-root 16 + square-root 16
```

... **não** é 8, como muitos poderiam esperar, mas 4.47213595499958, porque o que o Red vê é:

(ou mesmo: `square-root add 16 square-root 16`)

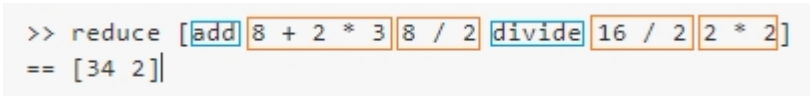
Isto é: Uma função que tem um argumento e um grupo avaliável (que por acaso tem uma função dentro).

Para obter aquele 8 intuitivo, é preciso usar parêntesis:

```
>> (square-root 16) + square-root 16
== 8.0
```

Outro exemplo, misturando operadores infixos e funções correspondentes:

```
>> reduce [add 8 + 2 * 3 8 / 2 divide 16 / 2 2 * 2]
== [34 2]
```



Outras explicações:

Estas são outras regras que eu ví em discussões na internet:

#1

"Esquerda para a direita e operadores tem precedência sobre funções e se um operador infixo vê uma função como seu operando, a resolve primeiro"

#2

"Em geral, as expressões são avaliadas da esquerda para a direita; entretanto, dentro de cada expressão, a computação ocorre da direita para a esquerda"

#3

"Cada expressão pega tantos argumentos quanto precisa. Por sua vez, cada argumento pode ter uma outra expressão e o Red vai fazer o *parse* da expressão até ter o conjunto completo de argumentos."

Dificuldades no aprendizado de Red:

Red é muito produtivo. É a linguagem de programação mais produtiva que conheço. Você pode fazer tanto usando tão pouco código! Também é muito fácil de usar depois de aprender, mas gostaria de comentar aqui alguns dos problemas que encontrei no processo. Você não pode realmente evitar essas armadilhas, mas sua jornada pode ser mais fácil se você estiver ciente delas.

 # 1 - Nova maneira de pensar. Demora mais tempo a aprender do que o esperado:

A produtividade do Red tem um preço. Embora os exemplos básicos sejam fáceis, parece-me que é muito difícil fazer programação real em Red sem entender seus principais conceitos. O Red não é feito de alguns blocos básicos que você monta como quiser, em Red tudo está interconectado. Avaliações, tipos de dados e dialetos permeiam toda a codificação.Trabalhar com o conceito de "código é dados e dados são código" requer prática para se acostumar. É como aprender uma língua estrangeira, você absorve pela repetição.

 # 2 - Tipos de dados incorretos em argumentos:

Uma palavra em Red pode ter qualquer um dos muitos datatypes disponíveis, mas as funções esperam um conjunto muito definido de datatypes em seus argumentos. Logo logo você encontrará um bug em que uma "variável" aparentemente inocente está bloqueando seu script ou gerando resultados inesperados sem motivo aparente. Uma boa ideia é iniciar sua depuração, verificando o tipo de dados de seus argumentos. Uma abordagem básica seria inserir alguns " `print type? <Variável>` " em seu código quando as coisas dão errado. Você pode descobrir quais tipos de dados sua função espera digitando " `? <Function>` " no console.

 # 3 - os dialetos usam apenas os comandos do dialeto:

Em breve você usará os dialetos internos do Red, como **VID** (para GUI), **parse** ou **draw**, e tentará inserir estruturas do Red dentro do bloco dialeto. Não vai funcionar. Os dialetos podem (ou não) ter seus próprios comandos para permitir que você use o Red normal dentro de seu bloco, mas você não pode simplesmente inserir um loop ou uma estrutura de controle sem a codificação adequada. Por exemplo, no VID, você pode usar `do [<Red code>],` mas outros dialetos exigem que você use funções externas e, em seguida, compute os resultados usando `compose`. Esse é um assunto para mais adiante mas, por enquanto, apenas tome cuidado.

Assim:

```
Red [needs: view]
parse [xxx] [só código parse aqui]
view [
     só comandos view aqui
     draw[só comandos draw aqui]

]
```

Entrada e saída no console

Nota: utilizar os comandos de entrada e saída do console pode causar problemas se você compilar os seus programas. Faz sentido, se você compilar, o console simplesmente não está lá!

native! print

`print` envia dados para o console. Após os dados, envia um `newline character` (nova linha) para o console. Ele avalia o argumento antes de colocá-lo no console.

```
Red []

print "hello"
print 33
print 3 + 5
```

```
hello
33
8
```

native! prin

`prin` também manda dados para o console, mas não envia o `newline character` . Ele avalia o argumento antes de colocá-lo no console.

```
Red []

prin "Hello"
prin "World"
prin 42
```

```
HelloWorld42
```

function! probe

`probe` manda o console o seu argumento sem avaliá-lo, **mas também retorna o argumento**. Lembre que `print` avalia o argumento. `probe` manda para o argumento para o console "como ele é" por assim dizer.
Pode ser usado para debugging como uma maneira de mostrar o código sem alterá-lo.

```
>> print [3 + 2]
```

```
5

>> probe [3 + 2] [3 + 2]
== [3 + 2]

>> print probe [3 + 2]
[3 + 2]
5
```

Descrito também aqui, após `mold`.

function! input

Lê uma **string** a partir do console. Note que qualquer número digitado no console será convertido para uma string. `newline character` são removidos.

```
Red []

prin "Enter a name: "
name: input
print [name "is" length? name "characters long"]
```

```
John
John is 4 characters long
```

routine! ask

A mesma coisa que `input`, mas exibe uma string fornecida por você.

```
Red []

name: ask "What is your name: "
prin "Your name is "
print name
```

```
What is your name: John
Your name is John
```

Executando código

Claro que você pode salvar o seu script como um arquivo e executar do command prompt, como um argumento do Red, assim:

```
C:\Users\you\whatever> red-063.exe myprogram.red
```

Isto abre o interpretador Red, abre o console (REPL) e executa o seu script.

Mas quando o Red já está sendo executado, você pode usar a palavra pré-definida `do` .

native! do

Avalia o código no seu argumento. Em outras palavras: executa o código. Este argumento pode ser um block, um arquivo, uma função ou qualquer outro valor.

```
>> do [loop 3 [print "hello"]]
hello
hello
hello
```

Dê uma olhada no capítulo Arquivos antes de continuar.

Por exemplo, se você salvou um script Red como myprogram.txt você pode executá-lo do console digitando isto:

```
>> do %myprogram.txt
```

Note que neste exemplo o interpretador Red e o arquivo texto estão na mesma pasta, se não for assim, você tem de fornecer o caminho (path) correto.

Ainda, se você digitar:

```
>> a: load %myprogram.txt
```

E depois:

```
>> do a
```

...o seu programa é executado normalmente.

`do`, `load` e `save` são melhor entendidos se você pensar no console do Red como a tela de algum computador dos anos 80 rodando alguma versão de BASIC. Você pode carregar um programa (`load`), salvá-lo (`save`) ou executá-lo (`do`).

Você pode ainda carregar e executar funções salvas como texto:

```
>> do load %myfunction.txt
```

Note que você pode fazer isto tudo **dentro de um script Red**,! Então, é um comando poderoso.

Parando a execução

function! quit

Pára a execução e sai do programa.

Se você digitar isso no console GUI (REPL), ele fecha. Se você digitar no *Command Line Interface*, você só sai do interpretador Red.

/return => Para a execução e sai do programa retornando um *status* para o sistema operacional. Não consegui achar um exemplo que consiga usar isso.

```
quit/return 3   ;passa 3 para o sistema operacional.
```

function! halt

Acho que simplesmente pára a execução do script. A documentação diz que retorna o valor 1.

routine! quit-return

Pára a execução com um dado *status*. Me parece ser a mesma coisa que `quit/return`, mas é um tipo `routine!` , não um `function!` . Vá entender!

VID DLS on-close

É um evento da VID. Executa um pedaço de código quando você fecha uma janela GUI. Mencionado também em GUI - Tópicos avançados.

Execute o programa abaixo e, quando você fechar a janela, ele vai imprimir "bye!" no console.

```
Red [needs: view]

view [
      on-close [print "bye!"]
      button [print "click"]
]
```

Control-C

Digitar control-C pára a execução do e sai do interpretador no *Command Line Interface*, mas não no console GUI.

Datatypes - tipos de dados

Pode ser uma boa idéia você dar uma olhada antes no capítulo sobre séries, uma vez que alguns exemplos usam palavras pré-definidas que estão listadas lá.

Datatypes básicos:

♦ none!

O equivalente a "null" em outras linguagens de programação. Um dado não-existente.

```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]
>> pick a 7
== none
```

♦ logic!

Além dos clássicos `true` e `false`, Red reconhece também `on` , `off`, `yes` e `no` como tipos de dado `logic!` .

```
>> a: 2 b: 3
== 3
>> a > b
== false
```

```
>> a: on
== true
>> a
== true
```

```
>> a: off
== false
>> a
```

```
== false
```

```
>> a: yes
== true
>> a
== true
```

```
>> a: no
== false
>> a
== false
```

♦ string!

Uma série de caracteres dentro de aspas ou colchetes {}. Se uma string se estende por mais de uma linha, os colchetes são obrigatórios.

Strings são séries, e podem ser manipulados usando os comandos descritos no capítulo sobre séries.

```
>> a: "my string"
== "my string"
```

```
>> a: {my string}
== "my string"
```

```
>> a: {my
{ string}       ;o primeiro "{" não é um erro, é como o console
mostra. Tente!
== "my^/string"
>> print a
my
string
```

```
>> a: "my new                          ;tentando usar aspas para
fazer uma string com mais de uma linha.
*** Syntax Error: invalid value at {"my new}
```

♦ char!

Precedidos por # e dentro de aspas, valores `char!` representam um ponto de Unicode. São integers (inteiros) que se estendem de hexadecimal 00 a hexadecimal 10FFFF. (0 a 1,114,111 em decimal.)

#"A" é um `char!`

"A" é um `string!`

Podem se submeter a operações matemáticas:.

```
>> a: "my string"
== "my string"
>> pick a 2
== #"y"
>> poke a 3 #"X"
== #"X"
>> a
== "myXstring"
```

```
>> a: #"b"
== #"b"
>> a: a + 1
== #"c"
```

♦ integer!

signed numbers de 32 bits. vão de −2,147,483,648 a 2,147,483,647. Se um número cai fora destes limites, o Red designa um `float!` datatype.

Nota: Dividir dois inteiros (integer!) dá um resultado truncado:

```
>> 7 / 2
== 3
```

♦ float!

Números floating point de 64 bits. Representados por números com um ponto ou usando a notação exponencial.

```
>> 7.0 / 2
== 3.5
```

```
>> 3e2
== 300.0
```

```
>> 6.0 / 7
== 0.8571428571428571
```

♦ file!

Tipo que representa arquivos é é precedido por %. Se você não está usando o path corrente, você tem que adicionar o path usando aspas. Barras e barras invertidas ("/" "\") são convertidos automaticamente pelo Red conforme o sistema operacional.

```
>> write %myfirstfile.txt "This is my first file"
```

```
>> write %"C:\Users\André\Documents\RED\mysecondfile.txt" "This is
my second file"
```

♦ path!

Usado para acessar dados dentro de estruturas usando "/". Pode ser usado em diferentes situações, por exemplo:

```
>> a: [23 45 89]
== [23 45 89]
>> print a/2
45
```

Barras "/" também são usadas para acessar objetos e refinamentos. Eu desconheço o funcionamento interno do Red, mas me parece que se trata de casos do datatype `path!` .

♦ time!

Tempo expresso em horas:minutos:segundos.subsegundos. Note que segundos e subsegundos são separados por um ponto e não por dois pontos. Você pode acessar cada um deles usando refinamentos. Veja o capítulo sobre Tempo e temporização.

```
>> mymoment: 8:59:33.4
== 8:59:33.4
>> mymoment/minute: mymoment/minute + 1
== 60
>> mymoment == 9:00:33.4
```

```
>> a: now/time/precise      ; o datatype de "a" é time!
== 22:05:46.805
>> type? a
== time!
>> a/hour
== 22
>> a/minute
== 5
>> a/second
== 46.805             ;second é um float!
```

♦ date!

O Red aceita datas em uma grande variedade de formatos:

```
>> print 31-10-2017
31-Oct-2017
>> print 31/10/2017
31-Oct-2017
>> print 2017-10-31
31-Oct-2017
>> print 31/Oct/2017
31-Oct-2017
>> print 31-october-2017
31-Oct-2017
>> print 31/oct/2017
31-Oct-2017
>> print 31/oct/17          ;só funciona se o ano é o último campo,
mas cuidado: 1917 or 2017?.
31-Oct-2017
```

O Red também checa se as datas são válidas, e até leva em consideração anos bisextos. Você pode acessar dia, mês e ano usando refinamentos:

```
>> a: 31-oct-2017
== 31-Oct-2017
>> print a/day
31
>> print a/month
10
>> print a/year
2017
```

♦ point! e ♦ pair!

`Point!` e `pair!` me parecem ser a mesma coisa. Provavelmente `pair!` existe para manter a compatibilidade com Rebol.
Representam pontos em um sistema cartesiano de coordenadas (eixos "x" e "y"). São compostos de dois inteiros separados por "x", por exemplo 23x45.

```
>> a: 12x23
== 12x23
>> a: 2 * a
== 24x46
>> print a/x
24
>> print a/y
46
```

♦ percent!

Representado adicionando "%" após o número.

```
>> a: 100 * 11.2%
== 11.2
>> a: 1000 * 11.3%
== 113.0
```

♦ tuple!

Um `tuple!` é uma lista de 3 até 12 bytes (inteiros de 0 a 255) separados por pontos.
Note que 2 números separados por ponto fazem um `float!` não um `tuple!`
Tuples são úteis para representar coisas como número de versões, número de IP e cores (exemplo: 0.225.0)
Um tuple! não é uma série, assim, a maior parte das operações de séries dão um erro se aplicadas a `tuples!`. Algumas operações que podem ser aplicadas a `tuples!` são:

`random`, `add`, `divide`, `multiply`, `remainder`, `subtract`, `and`, `or`, `xor`, `length?`, `pick` (não `poke`), `reverse`.

```
>> a: 1.2.3.4
== 1.2.3.4
>> a: 2 * a
== 2.4.6.8
>> print pick a 3
6
>> a/3: random 255
== 41
>> a
== 2.4.41.8
```

Classes de Datatypes ♦ number! e ♦ scalar!

Alguns datatypes são classes de datatypes.

`integer!`, `float!`, `percent!` pertencem ao datatype `number!`.

E qualquer um dos datatypes seguintes é também um `scalar!` datatype: `char!`, `integer!`, `float!`, `pair!`, `percent!`, `tuple!`, `time!`, `date!`

Hash! vector! e map!

Eu acho que estes datatypes são especiais e merecem um capítulo só para eles. Eles melhoram consideravelmente a qualidade e a velocidade do seu trabalho.

Hash! e vector! são séries de alta performance, ou seja, são mais rápidos no tratamento de séries grandes.

Eu sugiro que você dê uma olhada no capítulo Blocks & Series antes de estudar este.

♦ hash!

`hash!` é uma série que é "hashed" (indexada?) para fazer as buscas mais rápidas. Uma vez que o processo de "hashing" consome recursos, não vale a pena usar `hash!` para séries que vão sofrer buscas apenas poucas vezes. Entretanto, se as buscas forem constantes, há vantagem em fazer da sua série um `hash!` . O website do Rebol website diz que as buscas chegam a ser 650 vezes mais rápidas do que usando séries normais.

```
>> a: make hash! [a 33 b 44 c 52]
== make hash! [a 33 b 44 c 52]

>> select a [c]
== 52

>> select a 'c
== 52

>> a/b
== 44
```

Nada realmente novo, é só uma série.

♦ vector!

Vectors são séries de alta performance para `integer!` ,`float!`, `char!` ou `percent!`

Para criar um `vector!` você deve usar `make vector!`

Enquanto `hash!` permite buscas mais rápidas na série, `vector!` permite que operações matemáticas com a série sejam mais rápidas, pois são executadas na série toda de uma

vez só.

```
>> a: make vector! [33 44 52]
== make vector! [33 44 52]

>> print a
33 44 52

>> print a * 8
264 352 416
```

Note que você não poderia fazer isso numa série normal:

```
>> a: [2 3 4 5]
== [2 3 4 5]

>> print a * 2
*** Script Error: * does not allow block! for its value1 argument
*** Where: *
*** Stack:
```

♦ map!

Maps são dicionários de alta performance que associam chaves com valores (chave1: val1 chave2: val2 chave3: val3...).

Maps **não** são séries. Você não pode usar a maioria da palavras pré-definidas (comandos) de séries .

Para pôr e recuperar valores do dicionário, se deve usar `select` (series) e o comando especial: `put`.

```
>> a: make map! ["mini" 33 "winny" 44 "mo" 55]
== #(
    "mini" 33
    "winny" 44
    "mo" 55
...

>> print a
"mini" 33
"winny" 44
"mo" 55

>> print select a "winny"
44

>> put a "winny" 99
```

```
== 99

>> print a
"mini" 33
"winny" 99
"mo" 55
```

Outros datatypes:

♦ issue!

Série de caracteres usados para sequenciar símbolos ou identificadore para coisas como números de telefone, números de modelos, números de série, números de cartões de crédito etc. Um `issue!` tem que começar pelo caracter "#". A maior parte dos caracteres pode ser usada dentro de um `issue!`, sendo a barra "/" uma exceção..

```
>> a: #333-444-555-999
== #333-444-555-999

>> a: #34-Ab.77-14
== #34-Ab.77-14
```

♦ url!

Representada por <*protocolo*>**://**<*path*>

```
>> a: read http://www.red-lang.org/p/about.html
== {<!DOCTYPE html>^/<html class='v2' dir='ltr' x
```

♦ email!

Usado para identificar endereços de e-mail. A sintaxe não é checada, apenas deve possuir o caracter "@".

```
>> a: myname@mysite.org
== myname@mysite.org

>> type? a
== email!
```

♦ image!

Para criar uma `image!` você deve usar `make image!`
Os formatos de imagem suportados são: GIF, JPEG, PNG e BMP.
Quando você carrega (load) um arquivo de imagem, os dados são tipificados como `image!`. É pouco provável que você vá criar uma imagem com texto, mas se você quiser, o formato seria:

```
>> a: make image! [30x40 #{ ; here goes the data...
;You can change or get information from your image using the actions
that apply to series:
>> a: load %heart.bmp
== make image! [30x20 #{
    00A2E800A2E800A2E800A

>> print a/size
30x20

>> print pick a 1           ; getting the RGBA data of pixel 1
0.162.232.0

>> poke a 1 255.255.255.0   ; changing the RGBA data of pixel 1
== 255.255.255.0
```

♦ block!

Qualquer série dentro de colchetes.

♦ paren!

Uma série dentro de parêntesis.

♦ set-word! ♦ lit-word! ♦ get-word!

w: - atribui à palavra um valor. O seu tipo é `set-word!`

:w - resgata o valor associado à palavra sem computação. Seu tipo é `get-word!`

'w - trata a palavra como um símbolo, sem computação. Seu tipo é `lit-word` (literal word)

♦ refinement!

Precedido por um "/" - indica uma variação no uso ou uma extensão do significado de uma `function!`, `object!`, `file!` ou `path!`.

♦ action!

É o datatype de todas as "ações" do Red, p. exemplo: `add` , `take` , `append`, `negate` etc.

```
>> action? :take ; Colon is mandatory.
== true
```

TPara obter uma lista de todas as palavras pré-definidas do tipo `action!` digite:

```
>> ? action!
```

♦ op!

É o datatype dos operadores infixos, como `+` ou `**`.

♦ routine!

Usado para "linkar" com código externo.

♦ binary!

É uma série de bytes. Pode codificar dados como imagens, sons, strngs (no formato UTF ou outros), videos, dados comprimidos, dados encriptados ou outros.

O formato de origem pode estar na base 2, 16 ou 64. Não sei qual é o *default* em Red.

O formato da fonte é : #{...}

#{3A1F5A} ; base 16

2#{01000101101010} ; base 2

64#{0aGvXmgUkVCu} ; base 64

♦ word!

A mãe de todos os datatypes. Quando uma palavra é criada, este é o datatype dela.

♦ datatype!

É o datatype de todos os `datatype!` .

♦ event!

Este datatype é explicado em is explained in the Event! posição do mouse e uso de teclas.

♦ function!

♦ object!

♦ handle!

♦ unset!

♦ tag!

♦ lit-path!

♦ set-path!

♦ get-path!

♦ bitset!

♦ typeset!

- **error!**

- **native!**

Conversão de datatypes:

action! to

Converte de um `datatype!` para outro. Por exemplo, um `integer!` para uma `string!` , um `float!` para um `integer!` e mesmo uma `string!` para um `number!`.

```
>> to integer! 3.4
== 3
```

```
>> to float! 23
== 23.0
```

```
>> to string! 23.2
== "23.2"
```

```
>> to integer! "34"
== 34
```

function! to-time

Converte valores para o datatype `time!`.

```
>> to-time [22 55 48]
== 22:55:48
```

```
>> to-time [22 65 70]
== 23:06:10
```

```
>> to-time "11:15"
== 11:15:00
```

native! as-pair

Converte dois `integer!` ou `float!` em um `pair!`

```
>> as-pair 11 53
== 11x53
```

```
>> as-pair 3.2 5.67
== 3x5
```

```
>> as-pair 88 12.7
== 88x12
```

function! to-binary

Converete para um tipo `binary!` . Me parece que não é um conversor de bases, apenas um conversor de datatype.

```
>> to-binary 8
== #{00000008}
```

```
>> to-binary 33
== #{00000021}
```

Acessando e formatando dados

native! get

Toda palavra em Red, as nativas e as que você cria, vão para um dicionário. Se a palavra é associada com uma expressão, o dicionário mantém toda a expressão que pode ou não ser avaliada, dependendo do tipo do comando que busca a palavra.

Se você quer saber qual a descrição da palavra que está no dicionário, você usa `get` . Note que quando você se refere a uma palavra em Red (a própria palavra, não o valor) você a precede com um apóstrofe ('). `get` te devolve até o "significado das palavras pré-definidad do Red, mas retorna um erro se usada com um valor, por exemplo `integer!` `pair!` `tuple!` :

```
>> get 'print
== make native! [[
    "Output...

>> get 'get
== make native! [[
    "Return...

>> a: 7
== 7

>> get 'a
== 7

>> a: [7 + 2]
== [7 + 2]

>> get 'a
== [7 + 2]

>> get 8
*** Script Error: get does not allow integer! for its word argument
```

action! mold

`mold` tranforma um `datatype!` (por exemplo um `block!`, um `integer!` uma `series!` etc.) em uma string e a **retorna**:

```
>> type? 8
== integer!

>> type? mold 8
== string!

>> print [4 + 2]
6

>> print mold [4 + 2]
[4 + 2]
```

Refinamentos
/only - Exclui os colchetes externos se o valor for um block!
/all - Retorna o valor em um formato carregável (loadable).
/flat - Exclui toda a indentação.
/part - Limita o comprimento do resultado (argumento é um integer!)

action! form

`form` também tranforma um `datatype!` em uma string mas, dependendo to tipo, o resultado pode não conter informações extras, tais como [] {} e "", que seriam incluidas por `mold`. Útil para Manipulação de strings e texto.

```
Red []
print "---------MOLD----------"
print mold {My house
      is a very
      funny house}
print "---------FORM----------"
print form {My house
      is a very
      funny house}
print "---------MOLD----------"
print mold [3 5 7]
print "---------FORM----------"
print form [3 5 7]
```

```
---------MOLD----------
"My house^/^-is a very^/^-funny house"
---------FORM----------
My house
    is a very
    funny house
---------MOLD----------
[3 5 7]
---------FORM----------
3 5 7
```

O refinamento `/part` limita o número de caracteres retornado.

Principais usos para mold e form:

`mold` é basicamente usado para transformar uma série em código que pode ser salvo (*save*) e interpretado depois.

`form` é basicamente usado para gerar texto normal a partir de uma série.

```
>> a: [b: drop-down data[ "one" "two" "three"][print a/text]]
== [b: drop-down data ["one" "two" "three"] [print a/text]]

>> mold a
== {[b: drop-down data ["one" "two" "three"] [print a/text]]}

>> form a
== "b drop-down data one two three print a/text"
```

function! probe

`probe` imprime o argumento sem fazer avaliações, mas também o **retorna.** Lembre que `print` faz a avalição do seu argumento. `probe` imprime o argumento "como ele é", por assim dizer.
Pode ser usado para depurar o programa (debugging) como uma forma de mostrar o código sem alterá-lo.

```
>> print [3 + 2]
5

>> probe [3 + 2] [3 + 2]
== [3 + 2]

>> print probe [3 + 2]
[3 + 2]
5
```

native! reduce

Faz a avalição de uma expressão dentro de um bloco e retorna um novo bloco com os valores avaliádos. Dê uma olhada no capítulo sobre computação.

```
>> a: [3 + 5 2 - 8 9 > 3]
```

```
== [3 + 5 2 - 8 9 > 3]

>> reduce a
== [8 -6 true]

>> b:[3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]
== [3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]

>> reduce b
== [8 11 true [6 + 6 3 > 9]]            ;it does not evaluate
expressions of blocks inside blocks

>> b
== [3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]    ;the original block remains
unchanged.
```

/into => Põe o resultado em um bloco existente ao invés de criar um novo bloco.

function! collect e keep

`Collect` coleta em um novo bloco todos os valores passados pela função `keep` .
Em outras palavras: cria um novo bloco mantendo os valores determinados por `keep`, normalmente valores que atendem a uma determinada condição.

```
Red []

a: [11 "house" 34.2 "dog" 22]
b: collect [
      foreach element a [if string? element [keep element]] ;keeps string
elements
      ]
print b
```

```
house dog
```

/into => Coloca em um buffer ao invés de um criar um bloco (retorna a posição após a inserção).

syntax: collect/into [.....] <bloco de saída existente>

```
Red []

c: ["one" "two"]                    ; creating the output block with
some elements
```

```
a: [11 "house" 34.2 "dog" 22]   ; a generic series
collect/into [
      foreach element a [if scalar? element [keep element]] ;keeps
numbers of a
      ] c                                          ;appends them into c
print c
```

```
one two 11 34.2 22
```

native! compose

Retorna a cópia de um bloco, avaliando apenas os `paren!` (coisas dentro de parêntesis). `Compose` é muito importante para o dialeto DRAW;

```
Red []

a: [add 3 5 (add 3 5) 9 + 8 (9 + 8)]
print compose a               ;print evaluates everything!!
probe compose a               ;probe prints "as is"
```

```
8 8 17 17
[add 3 5 8 9 + 8 17]
```

/deep => faz o compose dentro de blocos aninhados (blocos dentro de blocos).

```
Red []

a: [add 3 5 (add 3 5) [9 + 8 (9 + 8)]]
probe compose a
probe compose/deep a
```

```
[add 3 5 8 [9 + 8 (9 + 8)]]
[add 3 5 8 [9 + 8 17]]
```

/only => faz o compose de blocos aninhados como blocos contendo seus valores.

/into => põe o resultado dentro de um bloco existente, ao invés de criar um novo bloco.

sintaxe: compose/into [.....] <bloco de saída existente>

```
Red []

a: [add 3 5 (add 3 5) 9 + 8 (9 + 8)]
b: []
compose/into a b
```

```
probe b
```

```
[add 3 5 8 9 + 8 17]
```

Matemática e lógica

Notas interessantes:

- Você pode usar um ponto ou uma vírgula como separador de decimal em um `float!`:

```
>> 5,5 + 9.2    ; vírgula no primeiro e ponto no segundo
== 14.7         ; A saída é sempre com ponto
```

- Se você usar apóstrofos para melhorar a leitura, o Red os ignora:

```
>> 5'420'120,00 * 2
== 10840240.0
```

- Você pode usar strings como input de fórmulas usando `do`:

```
>> do "2 + 5"
== 7
```

A maior parte da matemática e da lógica do Red é usual, exceto talvez a ordem de computação. Abaixo segue uma lista de operadores (palavras) usados para cálculos, acrescidos de algumas notas que achei úteis. A maior parte não precisa de uma descrição detalhada.

Matemática

O básico:

O grupo a seguir possui um operador **funcional** (por exemplo, `add`) e um **infixo** (por exemplo, "+"). Eles aceitam `number!` `char!` `pair!` `tuple!` ou `vector!` como argumentos (exceto `power`?).

Note que se você usar o operador funcional, ele vai antes dos operandos (por exemplo: 3 + 4 <=> `add` 3 4).

action! add ou op! +

```
>> add 3x4 2x3
== 5x7

>> now/time + 0:5:0   ; added five minutes to current time
== 7:16:27
```

action! subtract ou op! -

```
>> subtract 33 13
== 20

>> 3.4.6 - 1.2.1
== 2.2.5

>> now/month - 3  ;is october now
== 7
```

action! multiply ou op! *

```
>> multiply 3x2 2x5
== 6x10

>> 2.3.4 * 3.7.2
== 6.21.8
```

action! divide ou op! /

```
>> divide 3x5 2
== 1x2      ;truncate result because pair! is made of integer!

>> divide 8 3 ;truncate result because both are integer!
== 2

>> 8 / 3.0  ;3.0 is a float! so result is float!
== 2.666666666666667
```

action! power ou op! **

```
>> 3 ** 3
== 27
```

action! absolute

Avalia uma expressão e retorna o valor absoluto, isto é, um número positivo.

```
>> absolute 2 - 7
== 5
```

action! negate

Inverte o sinal de um valor, ou seja: positivo <=> negativo

```
>> negate 3x2
== -3x-2
```

float! pi

3,141592...

action! random

Retorna um valor aleatório do mesmo tipo que seu argumento.

Se o argumento for um inteiro, retorna um inteiro entre 1 (inclusive) e o argumento (inclusive).

Se o argumento for float, retorna um float entre 0 (inclusive) e o argumento (inclusive).

Se o argumento for uma série, ele embaralha os elementos.

```
>> random 10
== 2

>> random 33x33
== 13x23

>> random 1
== 1

>> random 1.0
== 0.07588539741741744

>> random "abcde"
== "cedab"

>> random 10:20:05
== 8:02:32.5867693
```

Refinamentos:

/ seed - Reinicia ou randomiza. Eu acho que o uso disso é se a sua função aleatória é chamada muitas vezes dentro de um programa.Nesse caso, ele pode não ser tão aleatório, a menos que você o reinicie com seed.

/ secure - TBD: Retorna um número aleatório criptograficamente seguro.

/ only - Escolha um valor aleatório de uma série.

```
>> random/only ["fly" "bee" "ant" "owl" "dog"]
== "fly"

>> random/only "aeiou"
== #"o"
```

action! round

Retorna o valor inteiro mais próximo. Metades (por exemplo, 0,5) são arredondadas para zero por padrão.

```
>> round 2.3
== 2.0

>> round 2.5
== 3.0

>> round -2.3
== -2.0

>> round -2.5
== -3.0
```

Refinamentos:

/to - Você fornece a "precisão" do seu arredondamento:

```
>> round/to 6.8343278 0.1
== 6.8

>> round/to 6.8343278 0.01
== 6.83

>> round/to 6.8343278 0.001
== 6.834
```

/even - Metades (por exemplo, 0,5) são arredondadas e não "para cima" como padrão, mas em direção ao número inteiro par.

```
>> round/even 2.5
== 2.0          ;not 3
```

/down - simplesmente trunca o número, mas mantém o número um `float!`.

```
>> round/down 3.9876
== 3.0

>> round/down -3.876
== -3.0
```

/half-down - Metades (0,5) são arredondados em direção ao zero, e não para longe do zero.

```
>> round/half-down 2.5
== 2.0

>> round/half-down -2.5
== -2.0
```

/floor - Arredonda na direção negativa

```
>> round/floor 3.8
== 3.0

>> round/floor -3.8
== -4.0
```

/ceiling - Arredonda na direção positiva

```
>> round/ceiling 2.2
== 3.0

>> round/ceiling -2.8
== -2.0
```

/half-ceiling - Metades arredondadas na direção positiva

```
>> round/half-ceiling 2.5
== 3.0

>> round/half-ceiling -2.5
== -2.0
```

native! square-root

Raiz quadrada. Usa qualquer `número!` como argumento.

Restos etc.:

action! remainder or op! //

Usa `number!` `char!` `pair!` `tuple!` e `vector!` como argumentos. Retorna o resto da divisão do primeiro pelo segundo valor.

```
>> remainder 15 6
== 3

>> remainder -15 6
== -3

>> remainder 4.67 2
== 0.67

>> 17 // 5
== 2

>> 4.8 // 2.2
== 0.3999999999999995
```

op! %

Retorna o resto quando um valor é dividido por outro.

function! modulo

Da documentação: "Wrapper para MOD que lida com erros como REMAINDER. Valores insignificantes (comparados com A e B) são arredondados para zero". Não consigo entender isso.

```
>> modulo 9 4
== 1

>> modulo -15 6
== 3

>> modulo -15 -6
== 3
```

```
>> modulo -15 7        ;?????
== 6

>> modulo -15 -7       ;?????
== 6
```

Logarítimos etc.:

function! exp

Eleva *e* (o número natural) à potência do argumento.

native! log-10

Retorna o logaritmo base 10 do argumento.

native! log-2

Retorna o logaritmo base 2 do argumento.

native! log-e

Retorna o logaritmo base *e* do argumento.

Trigonometria:

Todas as funções trigonométricas com nomes longos (`arccosine`, `cosine` etc) usam graus como padrão, mas aceitam o refinamento `/radians` para usar esta unidade. As versões de nome abreviado (`acos`, `cos` etc.) tomam radianos como argumentos e exigem que seja um `número!`

function! acos ou native! arccosine

function! asin ou native! arcsine

function! atan ou native! arctangent

Retorna o arco tangente trigonométrico.

function! atan2 ou native! arctangent2

Retorna o ângulo do ponto y / x em radianos, quando medido no sentido anti-horário a partir do eixo x de um círculo (onde 0x0 representa o centro do círculo). O valor de retorno está entre -pi e + pi.

function! cos ou native! cosine

function! sin ou native! sine

function! tan ou native! tangent

Extras:

native! max

Retorna o maior de dois argumentos. Argumentos podem ser `escalares!` ou `série!`

Não tenho certeza de como ele seleciona a série maior, mas parece escolher a série com o primeiro maior valor da esquerda para a direita.

```
>> max 8 12
== 12

>> max "abd" "abc"
== "abd"

>> max [1 2 3] [3 2 1]
== [3 2 1]
```

```
>> max [1 2 99] [3 2 1]
== [3 2 1]
```

Na comparação de dois `pair!` , retorna o maior para cada elemento:

```
>> max 12x6 7x34
== 12x34
```

native! min

Retorna o menor de dois argumentos. As notas de `max` aplicam-se aqui também.

action! odd?

Retorna `true` se o argumento (`integer!`) é ímpar, senão retorna `false` .

action! even?

Retorna `true` se o argumento (`integer!`) é par, senão retorna `false` .

native! positive?

`true` se for maior que zero. Nota: `falso` se zero.

native! negative?

`true` se menor que zero. Nota: `falso` se zero.

native! zero?

`true` somente se o argumento for zero.

function! math

Avalia um bloco! usando as regras matemáticas normais de precedência, ou seja, as divisões e multiplicações são avaliadas antes de adições e subtrações e assim por diante.

function! within?

Tem 3 argumentos do par! tipo. A primeira é a coordenada de um ponto (origem no canto superior esquerdo). Os outros dois descrevem um retângulo, o primeiro é sua origem superior esquerda e o segundo é a largura e a altura. Se o ponto estiver dentro ou na borda, retorna `true` , caso contrário, retorna `false` .

native! NaN?

Retorna true se o argumento for "não um número", caso contrário, false.

native! NaN

Retorna TRUE se o número for Não-um-número.

function! a-an

Retorna a variante apropriada de "a" ou "an" (língua inglesa- simples, vs 100% gramaticalmente correto).

Logic

action! and~ ou op! and (infix)

native! equal? ou op! =

native! greater-or-equal? ou op! >=

native! greater? ou op! >

native! lesser-or-equal? ou op! <=

native! lesser? ou op! <

native! not

native! not-equal? ou op! <>

action! or~ ou op! or (infix)

native! same? ou op! =?

Retorna true os argumentos se referem aos mesmos dados (objeto, string etc.), ou seja, ambos se referem ao mesmo espaço na memória.

```
>> a: [1 2 3]
== [1 2 3]

>> b: a          ; b points to the same data as a
== [1 2 3]

>> a =? b
== true          ; they are the same
```

```
>> c: [1 2 3]
== [1 2 3]

>> c =? a        ; c is equal to a, but is not the same data in
memory.
== false
```

native! strict-equal? or op! ==

Retorna `true` se os argumentos forem exatamente iguais, com o mesmo tipo de dados, letras maiúsculas / minúsculas (strings) etc.

```
>> a: "house"
>> b: "House"
>> a = b
== true

>> a == b
== false
```

Outras bases

native! to-hex

Converte um `integer!` em um hexadecimal (`issue!` datatype) com um # e 0's na frente).

```
>> to-hex 10
== #0000000A

>> to-hex 16
== #00000010

>> to-hex 15
== #0000000F
```

/size => Especifica o número de dígitos no resultado hexadecimal.

```
>> to-hex/size 15 4
== #000F

>> to-hex/size 10 2
== #0A
```

native! enbase e native! debase,

Estes são usados para codificar e decodificar strings codificadas em binário (binary - coded strings). Não é exatamente conversão de bases e, honestamente, eu não sei o uso disso, mas funciona assim:

```
>> enbase "my house"
== "bXkgaG91c2U="

>> probe to-string debase "bXkgaG91c2U="
"my house"
```

```
== "my house"
```

/base => base binária utilizada. Pode ser 64 (default), 16 ou 2.

```
>> enbase/base "Hi" 2
== "0100100001101001"

>> probe to-string debase/base "0100100001101001" 2
"Hi"
== "Hi"
```

native! dehex

Converte URLs que usam caracteres codificados em hexadecimal (%xx).

```
>> dehex "www.mysite.com/this%20is%20my%20page"
== "www.mysite.com/this is my page" ; Hex 20 (%20) é espaço "  "
```

```
>> dehex "%33%44%55"
== "3DU"
; %33 é hex para "3", %44 é hex para "D" e %55 é hex para "U".
```

Funções de manipulação de bits (bitwise):

op! **>>** Red-specs Red-by-example

right shift - documentação diz: "os bits mais baixos são jogados fora e os mais altos duplicados".

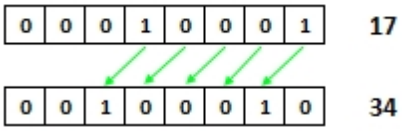
```
>> 144 >> 2
== 36
```

Não consegui entender como duplicar o bit mais alto se for 1. Tentei usar palavras de 32 bits, mas o Red as converte para floats.

op! **<<** Red-specs Red-by-example

left shift - bits mais altos são jogados fora, bit zero e adicionado à direita.

```
>> 17 << 1
== 34
```

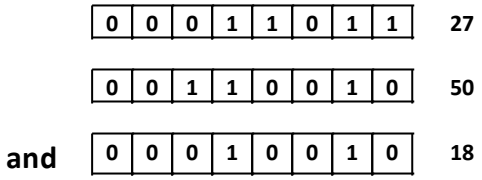


op! **>>>** Red-specs Red-by-example

logical shift - bits mais baixos são jogados fora, zeros são adicionados à esquerda. Não entendo porque é diferente de `>>`.

op! **and** & **and~** Red-specs Red-by-example

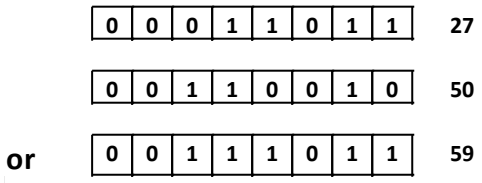
```
>> 27 and 50
== 18
```



A versão funcional (não infixa) de `and` é `and~`

op! **or** & **or~** Red-specs Red-by-example

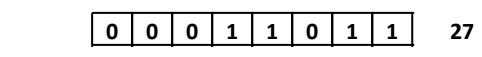
```
>> 27 or 50
== 59
```

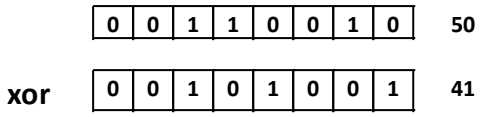


A versão funcional (não infixa) de `or` é `or~`

op! **xor** & **xor~** Red-specs Red-by-example

```
>> 27 xor 50
== 41
```





A versão funcional (não infixa) de `xor` é `xor~`

action! complement [Red-specs](#) [Red-by-example](#)

a fazer -

```
a fazer
```

Cryptography

native! checksum

Calcula a checksum usando algorítimos CRC, hash, ou HMAC.
Os argumento podem ser `string!` `binary!` ou `file!`

```
Red []

print "-----------  MD5  --------------"
print checksum "my house in the middle of our street" 'MD5
print "----------  SHA1  --------------"
print checksum "my house in the middle of our street" 'SHA1
print "---------  SHA256 --------------"
print checksum "my house in the middle of our street" 'SHA256
print "---------  SHA384  -------------"
print checksum "my house in the middle of our street" 'SHA384
print "---------  SHA512  -------------"
print checksum "my house in the middle of our street" 'SHA512
print "---------  CRC32  --------------"
print checksum "my house in the middle of our street" 'CRC32
print "----------  TCP  --------------"
print checksum "my house in the middle of our street" 'TCP
```

```
-----------  MD5  --------------
#{41F2FF19E5D7DF3B0E79FA9687C08397}
```

```
----------  SHA1  --------------
#{E97AE5E15E8EC1B87B0113E6A4758AAAE6E26901}
```

```
---------  SHA256 --------------
#{
98E2A2BFF328D893161CA6B6F50BA64D544026BD8C24C2022BE7007832714BA4
}
```

```
---------  SHA384  -------------
#{
2EAEA11D12F4CE8BE3CDE33DDED08765BFDCE1F277CF8E2126F7B1B6D4D17E31
96D05D2427576C348A0FECF63537B7D3
}
```

```
---------  SHA512  -------------
#{
0FAA749EAAEC728A6D821B85AC49CBE96DCE59E3FDC8E1005A3256A4CCE6797A
11603E9DB6B870C166057CF5EFBABB2365A87F37CDF2C8C1BF86DC8CE6D948C9
}
```

```
---------  CRC32  --------------
-1630692232
```

```
----------  TCP  --------------
13706
```

/with => Me parece que ainda não está implementado no Red, mas a descrição é: "Extra value for HMAC key or hash table size; not compatible with TCP/CRC32 methods."

Blocks & Séries

Blocks

Red é construído com "blocks". Em essência, qualquer coisa delimitada por colchetes é um bloco:[um block], [outro block [block dentro de um block]]

Séries

Séries são um tópico essencial em Red. Na verdade, os dados e até mesmo o próprio programa em Red são séries. Os elementos de uma série podem ser qualquer coisa dentro do léxico do Red: dados, palavras, funções, objetos e outras séries.

Strings etc.

Note que strings são tratadas pelo Red como séries de caracteres, então as técnicas de manipular séries também podem ser usadas para operações com strngs. Entretanto, como as manipulações com strings são tão importantes, tem um capítulo especial para Manipulação de strings e textos.

Na verdade, vários outros datatypes também são séries e podem ser manipulados com as palavras pré-definidas (comandos) descritos nos próximos capítulos.

Matrizes (Arrays)

Outras linguagens de programação tem um datatype chamado matriz (array). Não é difícil perceber que uma matriz de uma dimensão é simplesmente uma série, e matrizes multi-dimensionais são séries dentro de séries.

Aqui está um exemplo de uma matriz 3 x 2:

```
>> a: [[1 2][3 4][5 6]]
== [[1 2] [3 4] [5 6]]
```

Para acessar seus elementos, você pode usar "/":

```
>> a/1
== [1 2]

>> a/1/1
== 1
```

```
>> a/3/2
== 6
```

Navegação nas séries

- O primeiro elemento de uma série é chamado "`head`". Como veremos, ele pode não ser o "`first`" (primeiro) dependendo de como manipulamos a série;
- DEPOIS último elemento da série tem uma coisa chamada "`tail`" (cauda). Ele não tem um valor.
- Toda a série tem um "**entry index**". A melhor definição disto é "onde a parte utilizável da série começa. **Muitas operações com séries tem esse "entry index" como ponto de partida**. Você pode mover o **entry index** para frente e para trás para mudar o resultado de suas operações.
- Todo elemento da séria tem um número "`índex`" começando com 1 (não zero!) na primeira posição.
- Começando na posição do **entry index**, os elementos da série tem apelidos: "`first`" para o primeiro, "`second`" para o segundo "`third`"para o terceiro, "`fourth`" para o quarto e "`fifth`" para o quinto.

Nota: eu inventei o nome "**entry index**". Não está na documentação. Eu já ví o "**entry index**" sendo chamando apenas de "index", mas eu não gosto disso, pois pode causer confusão com o número `index` dos elementos da série.

action! head? action! tail? action! index?

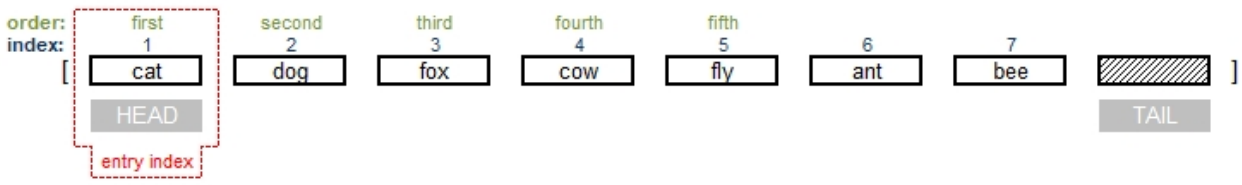
Essas palavras pre-definidas retornam informações sobre a posição do **entry index**. Se o **entry index** está na head, `head?` returna `true`, senão `false`. A mesma lógica se aplica para `tail?`. `index?` retorna o número `index` da posição do **entry index**.

Os exemplos a seguir devem deixar o seu uso claro:.

Primeiro criamos a série **s** com os strings **"cat" "dog" "fox" "cow" "fly" "ant" "bee"** :

```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

Teremos algo assim:



```
>> head? s
== true
```

```
>> index? s
== 1
```

```
>> print first s
cat
```

action! head action! tail

`head` move o **entry index** para o primeiro elemento da série, o "head".

`tail` move o **entry index** para a posição APÓS o último elemento da série, o "tail".

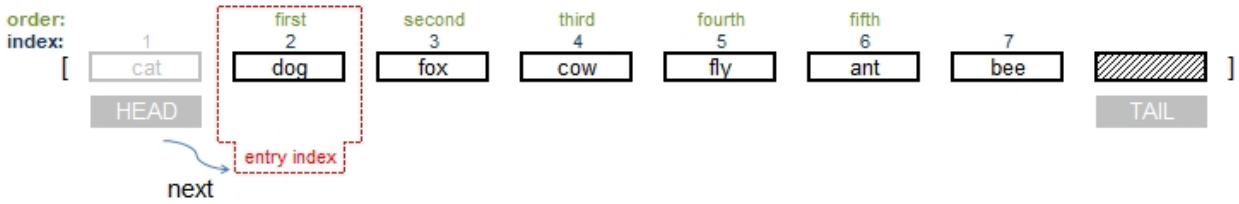
`head` e `tail` por si só não alteram a série, `head` apenas **retorna** toda a série, e `tail` não **retorna** nada. Para alterar uma série é preciso fazer uma atribuição de valor, por exemplo: `list: head list`

action! next

`next` move o **entry index** um elemento em direção ao tail. Note que `next` apenas **retorna** a série, não a modifica. Assim, simplesmente repetir `next` na mesma série não vai fazer o **entry index** ir além da segunda posição, pois você estará aplicando o comando na série original, onde o entry index ainda esta no primeiro elemento. Assim, para a maior parte dos usos, é preciso fazer uma atribuição, por exemplo: `s: next s.`

```
>> s: next s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

Agora temos:




```
>> print s
dog fox cow fly ant bee
```

```
>> head? s
== false

>> print first s
dog

>> index? s
== 2
```

Note que, apesar do primeiro elemento agora ser "dog", seu índice ainda continua sendo 2!

action! back

`back` é o oposto de `next`: move o **entry index** um elemento em direção ao "head". Se você usar back na nossa série **s,** "cat" é trazido de volta! Ele nunca foi "apagado"!

Isto significa que o Red nunca descartou nada da antiga série **s**. Essa é uma das peculiaridades do Red, os dados estão sempre lá, no próprio código.

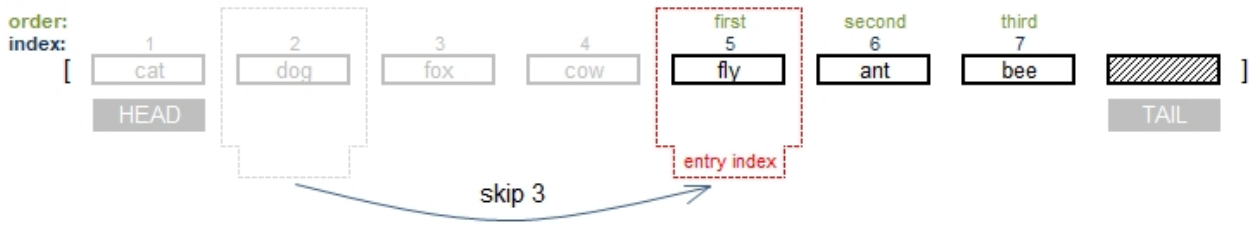
Depois de mover o índice da nossa série **s**, mesmo que você atribua esta série a outra palavra (variável), como **b**(b: s) você ainda pode executar movimentações e recuperar valores "escondidos na série **b**, pois **b** e **s** apontam para os mesmos dados.

Se você quer evitar isso , você deve criar a sua nova variável usando `copy`

Como já foi mencionado antes, em Red, diferentemente das outras linguagens, a variável (palavra) é associada aos dados, e não o contrário.

action! skip

Move o **entry index** um determinado número de elementos em direção ao *tail*.




```
>> s: skip s 3
== ["fly" "ant" "bee"]

>> print s
```

```
fly ant bee

>> print first s
fly

>> print index? s
5
```

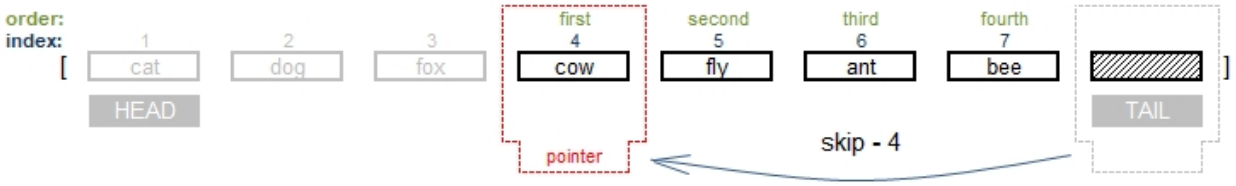
Se o número de skips for maior que o número de elementos da série, o **entry index** permanece na *tail*.

```
>> s: skip s 100
== []
```


```
>> tail? s
== true

>> index? s
== 8
```

Você também pode executar um número negativo de skips, o que faz o **entry index** ir em direção ao *head*:

```
>> s: skip s -4
== ["cow" "fly" "ant" "bee"]
```




```
>> print first s
cow

>> print index? s
4
```

Séries- comandos de consulta

Existem tantos comandos para manipular séries que eu os dividi em dois capítulos: um para os comandos de consulta, que apenas obtém informações sobre a série, sem alterá-la e outro para os comandos de alteração, que efetivamente alteram a série.

Os comandos de consulta apenas retornam valores, mas note que você pode criar uma nova série atribuindo a esta o valor retornado.

action! length?

Retorna o tamanho da série, do índice corrente até o fim.

```
>> a: [1 3 5 7 9 11 13 15 17]
== [1 3 5 7 9 11 13 15 17]

>> length? a
== 9

>> length? find a 13        ;veja o comando "find"
== 3                        ;do "13" ao tail existem 3 elementos
```

function! empty?

Retorna `true` se a série for vazia, do contrário retorna `false`.

```
>> a: [3 4 5]
== [3 4 5]

>> empty? a
== false

>> b:[]
== []

>> empty? b
== true
```

action! pick

Retorna o valor do elemento da posição dada pelo segundo argumento.


```
>> pick ["cat" "dog" "mouse" "fly"] 2
== "dog"
```

```
>> pick "delicious" 4
== #"i"
```

action! at

R**eturns** a série a partir de um elemento cuja posição é dada pelo segundo argumento.

```
>> at ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] 4
== ["cow" "fly" "ant" "bee"]
```

action! select e action! find

Ambos fazem uma busca na série por um determinado valor. A busca é da esquerda para a direita, exceto se forem utilizados os refinamentos `/reverse` ou `/last` .

Quando eles encontram o valor procurado:

- `select` retorna o próximo valor da série após a correspondência:

```
>> select ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
== "fly"
```

- `find` retorna a série a partir da correspondência até o fim:

```
>> find ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
== ["cow" "fly" "ant" "bee"]
```

/part

Limita a busca pela correspondência a um determinado número de elementos:

```
>> select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
 3
== none

>> select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["fox"]
 3
== "cow"
```

```
>> find/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]  3
 
== none

>> find/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
 4
== ["cow" "fly" "ant" "bee"]
```

/only

Trata o valor de busca como um bloco:

```
>> find/only ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"
"fly"]    ;finds nothing
== none

>> find/only ["cat" "dog" "fox" ["cow" "fly"] "ant" "bee" ] ["cow"
"fly"]    ;finds the block
== [["cow" "fly"] "ant" "bee"]
```

/case

Leva em conta letras maiúsculas e minúsculas

/skip

Trata a série como um conjunto de grupos, onde cada grupo tem um valor fixo. A correspondência é buscada apenas com o primeiro item de cada grupo. Abaixo eu ressaltem os "grupos" em amarelo e a correspondência em rosa:

```
>> find/skip ["cat" "dog"  "fox" "dog"  "dog" "dog" "cow" "dog"
"fly" "dog"  "ant" "dog"  "bee" "dog"] ["dog"]  2
```

```
== ["dog" "dog" "cow" "dog" "fly" "dog" "ant" "dog" "bee" "dog"]
```

/last

Encontra a correspondência a partir do fim da série (*tail*).

```
>> find/last [33 11 22 44 11 12] 11
== [11 12]
```

/reverse

O mesmo que `/last`, mas a partir do índice corrente que pode ser atribuído, por exemplo, pelo comando `at`.

find/tail

Normalmente `find` retorna o resultado incluíndo a correspondência. Com `/tail` o valor é retornado é a parte APÓS a correspondência, de forma similar a `select`

```
>> find ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] "fly"
== ["fly" "ant" "bee"]

>> find/tail ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] "fly"
== ["ant" "bee"]
```

find/match

Quando se usa `/match`, a comparação é feita com o começo da série. Além disso, o valor retornado se inicia após a correspondência.

```
>> find/match ["cat" "dog" "fox" "cow" "fly" "ant" "bee"] "fly"
== none          ;no match

>> find/match ["cat" "dog" "fox" "cow" "fly" "ant" "bee"] "cat"
== ["dog" "fox" "cow" "fly" "ant" "bee"]                  ;match
```

function! last

Retorna o último valor da série.

```
>> last ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== "bee"
```

function! extract

Extrai valores da série em intervalos regulares, **retornando** uma nova série.

```
>> extract [1 2 3 4 5 6 7 8 9] 3
== [1 4 7]

>> extract "abcdefghij" 2
== "acegi"
```

/index

Extrai valores a partir de um índice.

/into

Faz um "append" dos valores extraídos para uma dada série.

```
>> newseries: [] ;cria uma série vazia - necessário pois
extract/into não inicializa uma nova série.
== []

>> extract/into "abcdefghij" 2 newseries
== [#"a" #"c" #"e" #"g" #"i"]

>> extract/into ["cat" "dog" "fox" "cow" "fly" "ant" "bee" "owl"] 2
newseries
== [#"a" #"c" #"e" #"g" #"i" "cat" "fox" "fly" "bee"]
```

action! copy

Veja o capítulo Copiando.

Conjuntos:

native! union

Retorna o resultado da união de duas séries. Valores duplicados só são incluídos uma vez:

```
>> union [3 4 5 6] [5 6 7 8]
== [3 4 5 6 7 8]
```

/case

Leva em consideração maiúsculas e minúsculas.

/skip

Trata a série como grupos de tamanho fixo.

```
>> union/case [A a b c] [b c C]
== [A a b c C]
```

Com o refinamento `/skip`, apenas o primeiro elemento de cada grupo (tamanho dado pelo argumento) é comparado. Se houverem valores duplicados, os valores da primeira série são mantidos:

```
>> union/skip [a b c c d e e f f] [a j k c y m e z z] 3
== [a b c c d e e f f]

>> union/skip [k b c c d e e f f] [a j k c y m e z z] 3
== [k b c c d e e f f a j k]
```

native! difference

Retorna apenas os elementos que não estão presentes em ambas as séries.

```
>> difference [3 4 5 6] [5 6 7 8]
== [3 4 7 8]
```

/case

Leva em consideração maiúsculas e minúsculas.

/skip

Trata a série como grupos de tamanho fixo.

native! intersect

Retorna apenas os elementos que estão presentes em ambas as séries:

```
>> intersect [3 4 5 6] [5 6 7 8]
== [5 6]
```

/case

Leva em consideração maiúsculas e minúsculas.

/skip

Trata a série como grupos de tamanho fixo.

native! unique

Retorna a série removendo todos os elementos duplicados:

```
>> unique [1 2 2 3 4 4 1 7 7]
== [1 2 3 4 7]
```

Allows the refinements:

/skip

Trata a série como grupos de tamanho fixo.

native! exclude

Retorna uma série onde os elementos do segundo argumento são removidos do primeiro:

```
>> a: [1 2 3 4 5 6 7 8]
== [1 2 3 4 5 6 7 8]

>> exclude a [2 5 8]
```

```
== [1 3 4 6 7]

>> a
== [1 2 3 4 5 6 7 8]
```

Não encontrei na documentação, mas eu acho que a série retornada é composta por elementos não-repetidos:

```
>> exclude "my house is a very funny house" "aeiou"
== "my hsvrfn"

>> exclude [1 1 2 2 3 3 4 4 5 5 6 6] [2 4]
== [1 3 5 6]
```

/case

Leva em consideração maiúsculas e minúsculas.

/skip

Trata a série como grupos de tamanho fixo.

Séries- comandos de alteração

Estes comandos alteram a série original:

action! clear

Apaga os elementos da série.

Simplesmente atribuir " " (string vazia) ou zero para uma série não vai produzir os resultados esperados. A lógica do Red parece fazer com que ele "lembre" de coisas de forma inesperada. Para realmente limpar uma série, você precisa usar o `clear`.

```
>> a: [11 22 33 "cat"]
== [11 22 33 "cat"]

>> clear a
== []

>> a
== []
```

action! poke

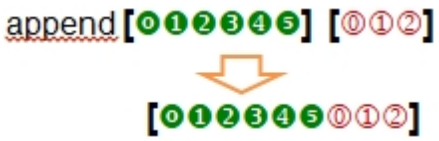
Altera o valor de um elemento da série na posição dada pelo segundo argumento. O novo valor da posição é dado pelo terceiro argumento.


```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> poke x 3 "BULL"
== "BULL"

>> x
== ["cat" "dog" "BULL" "fly"]
```

```
>> s: "abcdefghijklmn"
== "abcdefghijklmn"

>> poke s 4 #"W"
== #"W"

>> s
== "abcWefghijklmn"
```

action! append

Insere os valores do segundo argumento no final da série. Altera apenas a primeira série original.




```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> append x "HOUSE"
== ["cat" "dog" "mouse" "fly" "HOUSE"]

>> x
== ["cat" "dog" "mouse" "fly" "HOUSE"]
```

```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> y: ["Sky" "Bull"]
== ["Sky" "Bull"]

>> append x y
== ["cat" "dog" "mouse" "fly" "Sky" "Bull"]

>> x
== ["cat" "dog" "mouse" "fly" "Sky" "Bull"]
```

```
>> append "abcd" "EFGH"
== "abcdEFGH"
```

/part

Limita o numero de elementos do `append`.

```
>> append/part ["a" "b" "c"] ["A" "B" "C" "D" "E"]  2
== ["a" "b" "c" "A" "B"]
```

/only

Faz o `append` da série B na série A, mas B vai como um bloco:

```
>> append/only ["a" "b" "c"] ["A" "B"]
== ["a" "b" "c" ["A" "B"]]
```

/dup

Faz o `append` da série B na série A um determinado número de vezes. Acho que não devia se chamar dup de "duplicado", pois pode triplicar, quadruplicar...

```
>> append/dup ["a" "b" "c"] ["A" "B"] 3
== ["a" "b" "c" "A" "B" "A" "B" "A" "B"]
```

action! insert

É como o `append`, mas as adição é feita no entry index corrente (normalmente o começo). Enquanto o `append` **retorna** a série a partir do `head`, o `insert` **returna** a partir da inserção. Isto permite que sejam feitos muitas operações de `insert` em cadeia e ajuda a calcular o tamanho da parte inserida, mas note que `a: insert a xxx` não altera "a"!


```
>> a: "abcdefgh"
== "abcdefgh"

>> insert a "OOO"
== "abcdefgh"

>> a
== "OOOabcdefgh"
```


```
>> a: "abcdefgh"
== "abcdefgh"

>> insert at a 3 "OOO"
== "cdefgh"

>> a
== "abOOOcdefgh"
```

/part

Insere apenas um dado número de elementos do segundo argumento.

/only

Permite a inserção como bloco.

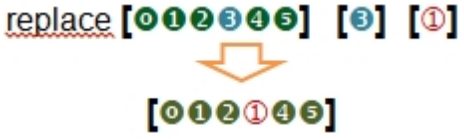
/dup

Permite a inserção um dado número de vezes:

```
>> a: "abcdefg"
== "abcdefg"

>> insert/dup a "XYZ" 3
== "abcdefg"

>> a
== "XYZXYZXYZabcdefg"
```

function! replace

Substitui os elementos de uma série.




```
>> replace ["cat" "dog" "mouse" "fly" "Sky" "Bull"] "mouse" "HORSE"
```

```
== ["cat" "dog" "HORSE" "fly" "Sky" "Bull"]
```

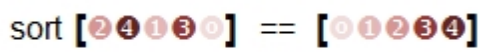
/all

Susbstitui todas as ocorrências:

```
>> a: "my nono house nono is nono nice"
== "my nono house nono is nono nice"

>> replace/all a "nono " ""
== "my house is nice"
```

action! sort

Ordena a série.




```
>> sort [8 4 3 9 0 1 5 2 7 6]
== [0 1 2 3 4 5 6 7 8 9]
```

```
>> sort "sorting strings is useless"
== " eeggiiilnnorrsssssssttu"
```

/case

Leva em consideração letras maiúsculas e minúsculas.

/skip

Trata a série como grupos de elementos de tamanho definido.

/compare

Para comparar offset, block ou function. (?)

/part

Ordena só parte da série.

/all

Compara todos os campos. (?)

/reverse

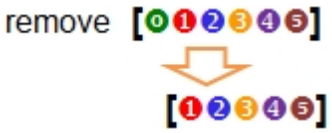
Reverte a ordem da ordenação.

/stable

Ordenação estável. (?)

action! remove

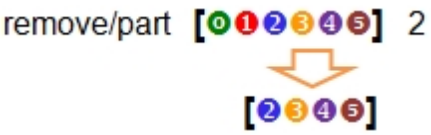
Remove o primeiro elemento da série.




```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> remove s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

/part

Remove um determinado número de elementos.




```
>> s: "abcdefghij"
== "abcdefghij"

>> remove/part s 4
== "efghij"
```

Note que você pode fazer a mesma coisa com `remove at [0 1 2 3 4 5] 2`.

native! remove-each

Assim como `foreach`, este comando executa um bloco para cada elemento da série. Se o bloco retornar `true`, o comando remove o elemento correspondente da série:

```
Red []

a: ["dog" 23 3.5 "house" 45]
remove-each i a [string? i] ;removes all strings
print a
```

```
23 3.5 45
```

```
Red []

a: "   my house in the middle of our street"
remove-each i a [i = #" "] ;removes all spaces
print a
```

```
myhouseinthemiddleofourstreet
```

action! take

Remove o PRIMEIRO elemento da série e **retorna** este primeiro elemento.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take s
== "cat"

>> s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

/last

Remove o último elemento da série e **retorna** este elemento.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/last s
== "bee"
```

```
>> s
== ["cat" "dog" "fox" "cow" "fly" "ant"]
```

`take/last` e `append` podem ser usados para fazer operações tipo "pilha" (stack).

/part

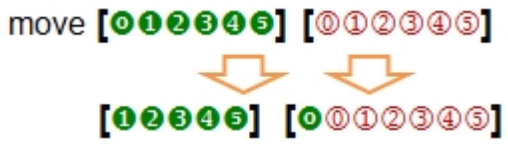
Remove um determinado número de elementos do começo de uma série e os dá como **retorno**.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 3
== ["cat" "dog" "fox"]

>> s
== ["cow" "fly" "ant" "bee"]
```

/deep
A documentação diz que "Copy nested values". Não entendi.

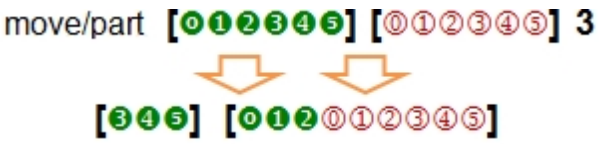
action! move

Move um ou mais elementos do segundo argumento para o primeiro. Altera as duas séries.



/part

Para mover mais de um elemento.




```
>> a: [a b c d]
== [a b c d]

>> b: [1 2 3 4]
== [1 2 3 4]
```

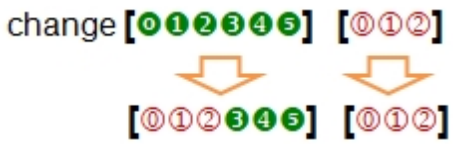
```
>> move a b
== [b c d]

>> a
== [b c d]

>> b
== [a 1 2 3 4]

>> move/part a b 2
== [d]

>> a
== [d]

>> b
== [b c a 1 2 3 4]
```

`move` pode ser usado em combinação com outros comandos para mover elementos dentro de uma mesma série:

```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]

>> move a tail a
== [2 3 4 5 1]

>> move/part a tail a 3
== [5 1 2 3 4]
```

action! change

Altera os primeiros elementos de uma série e retorna a série após a mudança. Modifica a primeira série original, não a segunda.




```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]

>> change a [a b]
```

```
== [3 4 5]

>> a
== [a b 3 4 5]
```

/part
Limita a quantidade de mudanças a um dado número de elementos.

/only
Muda uma série como série.

/dup
Faz a mudança um determinado número de vezes.

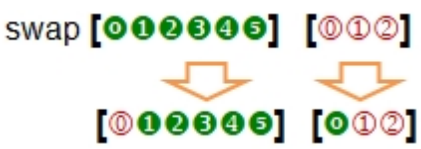
function! alter

Pode remover ou fazzer um `append` na série. Se `alter` NÃO encontrar o elemento na série, ele faz o append desse elemento e retorna `true`. Se ele encontra o elemento, remove-o , e retorna `false`.

```
>> a: ["cat" "dog" "fly" "bat" "owl"]
== ["cat" "dog" "fly" "bat" "owl"]

>> alter a "dog"
== false

>> a
== ["cat" "fly" "bat" "owl"]

>> alter a "HOUSE"
== true

>> a
== ["cat" "fly" "bat" "owl" "HOUSE"]
```

action! swap

Troca os primeiros elementos de duas séries. **Retorna** a primeira série, mas modifica as duas:



```
>> a: [1 2 3 4]   b: [a b c d]

>> swap a b
== [a 2 3 4]

>> a
== [a 2 3 4]

>> b
== [1 b c d]
```

Com `find` , por exemplo, pode ser usado para trocar quaisquer elementos de duas séries ou mesmo elementos dentro de uma mesma série:

```
>> a: [1 2 3 4 5] b: ["dog" "bat" "owl" "rat"]
== ["dog" "bat" "owl" "rat"]

>> swap find a 3 find b "owl"
== ["owl" 4 5]

>> a
== [1 2 "owl" 4 5]

>> b
== ["dog" "bat" 3 "rat"]
```

action! reverse

Reverte a ordem dos elementos de uma série:

```
>> reverse [1 2 3]
== [3 2 1]

>> reverse "abcde"
== "edcba"
```

/part limita o número de elementos a serem revertidos:

```
>> reverse/part "abcdefghi" 4
== "dcbaefghi"
```

Copying

AVISO AOS INICIANTES: Se você está atribuindo o valor de uma palavra (variável) a outra palavra (variável) em Red, use COPY!

```
>> var1: var2          ;só se você tiver certeza do que está fazendo

>> var1: copy var2     ;pode te poupar horas tentando debugar.
```

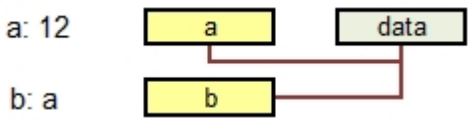
action! copy

Atribui uma cópia do dado a uma nova palavra.

Pode ser usado para copiar séries e objetos.

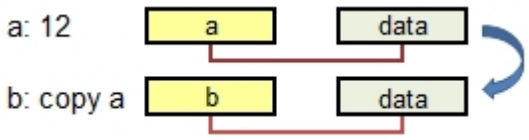
Não é usado em itens simples como: integer! float! char! etc. Para estes, podemos usar apenas os dois pontos (:).

Primeiro vejamos uma atribuição simples:




```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> b: s
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 4
== ["cat" "dog" "fox" "cow"]

>> b
== ["fly" "ant" "bee"]            ;b changes!!
```

Agora com `copy`:




```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> b: copy s
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 4
== ["cat" "dog" "fox" "cow"]

>> b
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

Se você tiver uma série aninhada, por exemplo, um bloco dentro de uma série, `copy` não altera a referência para esta série aninhada. Se você quiser criar uma cópia independente, você precisa usar o refinamento `/deep` para criar uma cópia "profunda".

/part

Limita o tamanho do resultado, onde o tamanho é dado por um `number!` ou `series!`

```
>> a: "my house is a very funny house"
>> b: copy/part a 8
== "my house"
```

/types

Copia apenas tipos específicos de valores não-escalares.

/deep

Copia valores aninhados, conforme mencionado acima.

Repetições

native! loop

Executa um bloco um dado número de vezes:

```
Red[]

loop 3 [print "hello!"]
```

```
hello!
hello!
hello!
>>
```

native! repeat

`repeat` é o mesmo que `loop`, mas possui um índice que é incrementado a cada repetição:

```
Red[]

repeat i 3 [print i]
```

```
1
2
3
>>
```

native! forall

Executa um bloco enquanto avança em uma série:

```
Red[]

a: ["china" "japan" "korea" "usa"]
forall a [print a]
```

```
china japan korea usa
japan korea usa
korea usa
usa
>>
```

native! foreach

Executa um bloco para cada elemento da série:

```
Red[]

a: ["china" "japan" "korea" "usa"]
foreach i a [print i]
```

```
china
japan
korea
usa
>>
```

native! while

Executa um bloco enquanto uma condição é verdadeira:

```
Red[]

i: 1
while [i < 5] [
      print i
      i: i + 1
]
```

```
1
2
3
4
>>
```

native! until

Executa um bloco até que este bloco retorne `true` .

```
Red[]

i: 4
until [
      print i
      i: i - 1
      i < 0 ;  <= condition
]
```

```
4
3
2
1
0
>>
```

native! break

Força uma saída da repetição.

/return

Força a saída e retorna um dado valor, como um código ou uma mensagem.

native! forever

Cria um loop infinito.

Estruturas de controle

native! if

Executa um bloco se o teste for `true`.

if <test> [block **]**

```
>> if 10 > 4 [print "large"]
large
```

native! unless

A mesma coisa que `if not`. Executa um bloco só se o teste for `false`.

unless <test> [block (se teste `false`) **]**

```
>> unless 10 > 4 [print "large"]
== none

>> unless 4 > 10 [print "large"]
large
```

native! either

Um novo nome para o clássico **if-else**. Executa o primeiro bloco se o teste for `true` ou executa o segundo bloco se o teste for `false`.

either <test> [`true` block**] [**`false` block**]**

```
>> either 10 > 4 [print "bigger"] [print "smaller"]
bigger

>> either 4 > 10 [print "bigger"] [print "smaller"]
smaller
```

native! switch

Executa o bloco correspondente a um determinado valor:

```
Red[]

switch 20 [
      10 [print "ten"]
      20 [print "twenty"]
      30 [print "thirty"]
]
```

```
twenty
```

/default

Se o programa não encontrar uma correspondência, executa um bloco *default*.

```
Red[]

switch/default 15 [
      10 [print "ten"]
      20 [print "twenty"]
      30 [print "thirty"]
][ print "none of them"]      ;default block
```

```
none of them
```

native! case

Faz uma série de testes, executando o bloco correspondente ao primeiro teste que retorna `true`.

```
Red[]

case [
      10 > 20 [print "not ok!"]
      20 > 10 [print "this is it!"]
      30 > 10 [print "also ok!"]
]
```

```
this is it!
```

/all

Executa **todos** os testes que retornam `true` .

```
Red[]

case/all [
	10 > 20 [print "not ok!"]
	20 > 10 [print "this is it!"]
	30 > 10 [print "also ok!"]
]
```

```
this is it!
also ok!
```

native! catch & throw

Catch e throw podem ser usados para criar estruturas de controle complexas. Na sua forma mais simples `catch` recebe o **return** de um entre vários throws:

```
Red[]

a: 10
print catch [
	if a < 10 [throw "too small"]
	if a = 10 [throw "just right"]
	if a > 10 [throw "too big"]
]
```

```
just right
```

catch/name

faz o catch de um throw com nome.

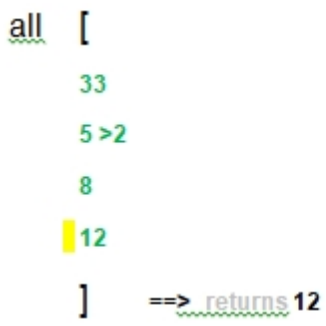
throw/name

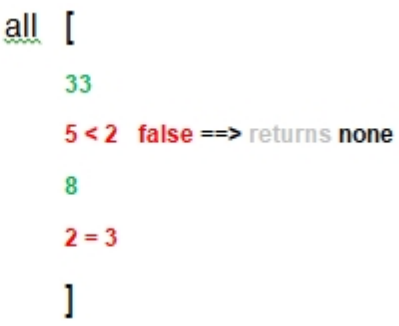
faz throws de um catch com nome.

Controle Booleano (lógica)

native! all

Avalia todas as expressões em um bloco. Se alguma retorna `false`, o retorno do `all` é `none`, senão retorna o resultado da última computação.

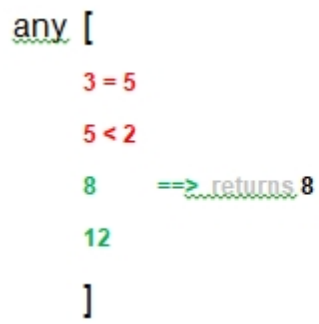





```
>> john: "boy"
== "boy"

>> alice: "girl"
== "girl"

>> all [john = "boy" alice = "girl" 10 + 3]   ;todas verdadeiras, a
última computação é retornada.
== 13

>> all [john = "boy" alice = "boy" 10 + 3]    ; alice = "boy" é
false!
== none
```

native! any

Avalia cada expressão em um bloco e retorna o primeiro valor que não é `false`. Se todos os valores forem `false` retorna `none`.




```
>> john: "boy"
== "boy"

>> alice: "girl"
== "girl"

>> any [john = "girl"  alice = "girl"  10 + 3]  ;alice = "girl" não é
falso: retorna !
== true
```

```
>> any [john = "girl"  10 + 3  5 > 2]                ; 10 + 3 não é
falso: retorna!
== 13
```

Manipulação de strings e texto

Nota: nos exemplos, algumas linhas de saída do console foram retiradas para clareza.

function! split

Cria um bloco (uma série) contendo as partes de uma string separadas por um delimitador. Este delimitador é dado como um argumento. `split` é particularmente útil para analisar e manipular arquivos de texto e no dialeto de *parse*.

```
>> s: "My house is a very funny house"
>> split s " "                                    ;every space is
a delimiter.
== ["My" "house" "is" "a" "very" "funny" "" "" "" ""
"house"] ;result is a series with 11 elements.

>> s: "My house ; is a very ; funny house"
>> split s ";"                                    ;split happens
at the semi-colons.
== ["My house " " is a very " " funny house"]      ;result is a
series with 3 elements.
```

removing characters: action! trim

A palavra `trim` sem refinamentos, remove os espaços em branco e os tabs do começo e do fim de uma string. Também remove `none` de um bloco ou um objeto. O valor do argumento é alterado. Possui refinamentos para retirar caracteres específicos..

Refinements:

/head - Remove só da *head* (início).
/tail - Remove só da *tail* (final).
/auto - Auto-indenta linhas relativas à primeirla linha.
/lines - Remove todos os line-breaks e espaços extras.
/all - Remove todos os espaços, mas não os line-breaks.
/with - A mesma coisa que /all, mas remove caracteres em um argumento "with" fornecido por você. Deve ser char! string! ou integer!

```
>> e: " spaces before and after "
>> trim e
== "spaces before and after"
```

trim espaços antes:

```
>> e: " spaces before and after "
>> trim/head e
== "spaces before and after "
```

trim espaços depois:

```
>> e: " spaces before and after "
>> trim/tail e
== " spaces before and after"
```

trim caracteres específicos:

```
>> d: "our house in the middle of our street"
>> trim/with d " "
== "ourhouseinthemiddleofourstreet"
```

```
>> a: "house"
>> trim/with a "u"
== "hose"
```

o oposto de trim: function! pad

`pad` expande uma string para um determinado tamanho usando espaços. O default é adicionar espaços à direita, mas com o refinamento `/left`, os espaços são adicionados à esquerda, ou seja, no início. Cuidado, pois a string original é modificada.

```
>> a: "House"
>> pad a 10
== "House "
```

```
>> pad/left a 20
== " House "
```

contatenação de texto: function! rejoin

```
>> a: "house" b: " " c: "entrance"
>> rejoin [a b c]
== "house entrance"
```

ou, usando `append` - modifica a série original.

```
>> append a c
== "house entrance"
```

```
>> a: "house" b: " " c: "entrance"

>> append a c
== "houseentrance"

>> append a append b c
== "houseentrance entrance"       ; "a" foi alterada para
"houseentrance" na última manipulação
```

transformando séries em texto: action! form

`form` transforma uma série em uma string, removendo os colchetes e adicionando espaços entre os elementos. `form` foi visto também no capítulo Acessando e formatando dados.

```
>> a: ["my" "house" 23 47 4 + 8 ["a" "bee" "cee"]]
>> form a
== "my house 23 47 4 + 8 a bee cee"
```

/part

O refinamento `/part` limita o número de caracteres da string criada.

```
>> a: ["my" "house" 23 47 4 + 8 ["a" "bee" "cee"]]
>> form/part a 8
== "my house"
```

comprimento da string: action! length?

```
>> f: "my house"
>> length? f
== 8
```

parte esquerda da string:

usando **copy/part** :

```
>> s: "nasty thing"
>> b: copy/part s 5
== "nasty"
```

parte direita da string:

usando `at` :

```
>> s: "nasty thing"
>> at tail s -5
== "thing"
```

usando `remove/part` - isto muda a série original, cuidado!

```
>> s: "nasty thing"
>> remove/part s 6
== "thing"
```

parte do meio da string:

usando `copy/part` e `at`:

```
>> a: "abcdefghijkl"
>> copy/part at a 4 3
== "def"
```

inserindo strings:

no começo, usando `insert`:

```
>> s: "house"
>> insert s "beautiful "

>> s
== "beautiful house"
```

no fim, usando `append`:

```
>> s: "beautiful"
>> append s " day"
== "beautiful day"
```

no meio, usando `insert at`:

```
>> s: "nasty thing"
>> insert at s 7 "little "

>> s
== "nasty little thing"
```

native! lowercase - letras minúsculas

Muda a string original, cuidado.

```
>> a: "mY HoUse"
>> lowercase a
== "my house"
```

/part

```
>> a: "mY HoUse"
```

```
>> lowercase/part a 2
== "my HoUse"
```

native! uppercase - letras maiúsculas

```
>> a: "mY HoUse"
>> uppercase a
== "MY HOUSE"
```

/part

```
>> a: "mY HoUse"
>> uppercase/part a 2
== "MY HoUse"
```

Imprimindo caracteres especiais

Essa informação foi obtida na documentação do Rebol, mas eu testei a maior parte deles em Red e parecem funcionar:

Caracteres de controle:

Caracter	Definição
`#"^(null)" or #"^@"`	null (zero)
`#"^(line)", or #"^/"`	nova linha
`#"^(tab)" or #"^-"`	tab horizontal
`#"^(page)"`	new page (and page eject)
`#"^(esc)"`	escape
`#"^(back)"`	backspace
`#"^(del)"`	delete
`#"^^"`	caret character
`#"^""`	aspas
`#"(0)" to #"(FFFF)"`	caracteres em hex

Caracteres especiais para usar dentro de strings:

Caracter	Função
`^"`	imprime um " (aspas)
`^}`	insere um } (chave - fechar)
`^^`	insere um ^ (caret?)
`^/`	começa nova linha
`^(line)`	começa nova linha
`^-`	insere um tab
`^(tab)`	insere um tab
`^(page)`	nova página (?)
`^(letter)`	insere control-letter (A-Z)
`^(back)`	apaga um caracter atrás
`^(null)`	insere um carcter "null"
`^(esc)`	insere um caracter "escape"
`^(XX)`	insere um ASCII pelo número hexadecimal dele

Tempo e temporização

native! wait

Pára a execução pelo número de segundos do argumento.

- Nota: pelo menos até novembro 2017, o console GUI não trabalhava bem com `wait`.

native! now

Retorna a data e a hora:

```
>> now
== 12-Dec-2017/19:24:41-02:00
```

Refinamentos:

/time - Retona apenas a hora. time!

```
>> now/time
== 21:42:41
```

/precise - data e hora de alta precisão. date!

```
>> now/precise
== 2-Apr-2018/21:49:04.647-03:00
```

```
>> a: now/time/precise
== 22:05:46.805       ;a é um time!

>> a/hour
== 22                 ;hora é integer!

>> a/minute
== 5                  ;minuto é integer!

>> a/second
== 46.805             ;segundo é float!
```

Este script cria um dalay de 300 milisegundos (0,3 segundos):

```
Red []
thismoment: now/time/precise
print thismoment
while [now/time/precise < (thismoment + 00:00:00.300)][]
print now/time/precise
```

```
21:51:58.173
21:51:58.473
```

/year - Retorna apenas o ano. integer!

```
>> now/year
== 2018
```

/month - Retorna apenas o mês. integer!

```
>> now/month
== 4
```

/day - Returns apenas o dia do mês. integer!

```
>> now/day
== 2
```

/zone - Retorna apenas o *offset* (diferença) do UCT (GMT) . time!

```
>> now/zone
== -3:00:00
```

/date - Retonra apenas a data. date!

```
>> now/date
== 2-Apr-2018
```

/weekday - Retorna o dia as semana como integer! (segunda-feira é 1).

```
>> now/weekday
== 1
```

/yearday - Returna o dia do ano (Juliano). integer!

```
>> now/yearday
== 92
```

/utc - Hora Universal UTC (sem zona). date!

```
>> now/utc
== 3-Apr-2018/0:53:50
```

VID DLS rate

A temporização também pode ser obtida com a *facet* `rate` do dialeto VID

Tratamento de erros

function! attempt

Avalia um bloco e retonra o resultado ou `none` se ocorrer um erro.

```
>> attempt [a: 10 b: 9]     ;vamos tentar sem erros...
== 9

>> a
== 10                       ;... sem problemas!

>> attempt [a: 10 nosyntax]       ;nosyntax não tem valor: ERROR!
== none
```

native! try

Tenta computar um bloco. Retonra o valor do bloco, mas se ocorrer um `error!` , o bloco é abandonado e um valor de erro é retornado.
Para identificar um bloco que gera erro sem ter o erro impresso, usamos a função `error?` .

Você pode estar se perguntando porque não usar `attempt` ao invés de `error?` & `try`. Eu acho que a resposta é que a combinação `error?` & `try` retorna `true` e `false`, ao invés de `none` ou um resultado de computação. Isto é útil quando usado dentro de outras estruturas.

```
>> error? [nosyntax]
== false              ;nosyntax não tem valor e gera um erro,
                      ;mas apenas se avaliado. Por sí só, não é um
datatype error!

>> try [nosyntax]
*** Script Error: nosyntax has no value
*** Where: try
*** Stack:            ; apenas "try" não funciona, dá erro!!

>> error? try [nosyntax]
== true               ;OK!
```

native! catch e native! throw

Também são usados para tratar erros, mas eu não entendi como. Parece ser algo muito avançado para principiantes, como está discutido aqui.

Arquivos

Path (caminho), diretórios e arquivos

nomes de Path

Caminhos para arquivos são escritos com um sinal de percentagem (%) seguido pela sequência de nomes de diretórios, separados por uma barra (/). No Windows, o Red faz a conversão de barras à esquerda ou à direita (/ ou \), você não precisa se preocupar com isso.

Apenas lembrando:

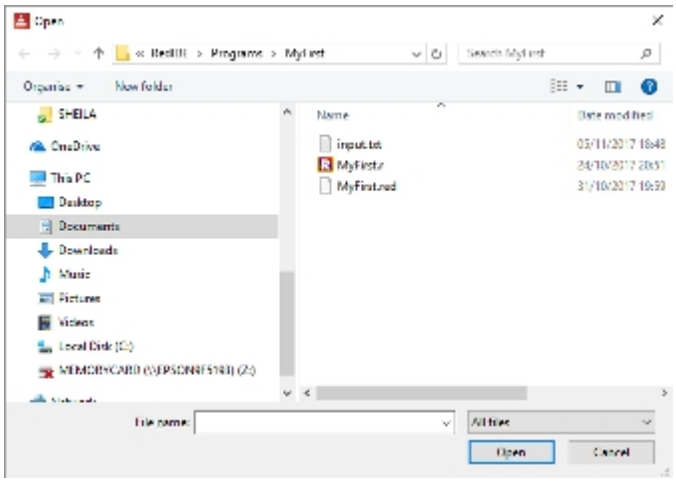
- `/` é a raiz (root) do drive corrente;
- `./` é o diretório corrente;
- `../` é o diretório superior do diretório corrente;
- caminhos de arquivo que não começam com barra (/) são caminhos relativos;
- para se referir ao drive "C" do Windows, você deve usar: `%/C/docs/file.txt`
- caminhos absolutos devem ser evitados para garantir scripts que sejam independentes da máquina;

Um seletor gráfico de arquivos:

function! request-file

`request-file` abre um seletor gráfico (GUI) de arquivos e retorna o caminho completo como um `file!`

```
>> request-file
```



```
== %/C/Users/André/Documents/RED/myFirstFile.txt
```

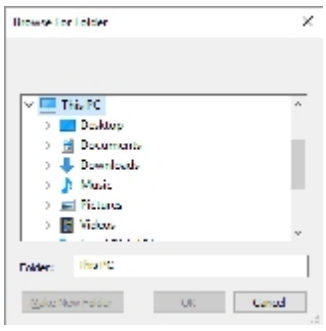
Refinamentos

/title - título da janela. Exemplo: `request-file/title "My file is:"`
/file - nome de arquivo ou diretório usado como padrão (default). Exemplo: `request-file/file %"MyFile.txt"`
/filter -Fornece um bloco de filtros que consiste de pares compostos do nome dos filtros e os filtros propriamente ditos. Exemplo: `request-file/filter ["executables" "*.exe" "text files" "*.txt"]`
/save - Modo de salvar arquivo. Exemplo com filtros: `request-file/save/filter ["executables" "*.exe" "text files" "*.txt"]`
/multi - Permite seleções múltiplas, retornadas como um bloco.

Um seletor gráfico de diretórios:

function! request-dir

`request-dir` abre um seletor gráfico de diretórios e retorna o caminho completo como um `file!`



```
== %/C/Users/André/Documents/RED/
```

Refinamentos:

/title => título da janela.
/dir => Determina o diretório inicial.
/filter => TBD: Bloco de filtros (filter-name filter).
/keep => Mantém o caminho de diretório anterior.
/multi => TBD: Permite múltiplas seleções, retornadas como um bloco.

Apagando um arquivo:

action! delete

Apaga um arquivo e retorna `true` em caso de sucesso ou, caso contrário, `false` .

```
>> delete %file.txt
== true
```

Obtendo o tamanho de um arquivo:

native! size?

Retorna o número de bytes de um arquivo ou, se o arquivo não existir, retorna `none` .

```
>> size? %myFirstFile.txt
== 37
```

Outras funções de diretório e arquivo:

cd or change-dir - Muda o diretório corrente.

dir, ls or list-dir - Lista o conteúdo de um dado diretório. Se nenhum argumento for dado, lista o diretório corrente.

dir? - Retorna `true` se o nome fornecido for um caminho (`path!`) válido. Senão retorna `false`.

dirize - Transforma o argumento em um diretório válido. O argumento pode ser um file! string! url!.
Só coloca um / no final se for necessário.

exists? - Retorna `true` se o argumento for um `path!` existente, senão retorna `false` .

file? - Retorna true se o argumento é um `file!`.

get-current-dir - Retorna o diretório corrente.

get-path? - Retorna `true` se o argumento é um `get-path!`

path? - Retorna `true` se argumento é um `path!`

split-path - Divide o caminho de um `file!` ou `url!` . Retorna um bloco contendo o caminho e o alvo.

suffix? - Retorna o sufixo de um arquivo, por exemplo: exe, txt.

what-dir - Retorna o diretório corrente como um valor `file!` .

to-red-file - Converte um sistema de arquivos local em um sistema independente de máquina.

to-local-file - Converte do sistema de arquivos independente de máquina para o sistema de arquivos do sistema operacional da máquina.

clean-path - Limpa os '.' e '..' de um caminho e retorna o caminho limpo.

red-complete-file

red-complete-path

set-current-dir

Escrevendo em arquivos

Escrevendo para um arquivo texto:

action! write

Escreve em um arquivo, criando-o se necessário.

```
>> write %myFirstFile.txt "Once upon a time..."
```

Acrescentando (Appending) à um arquivo texto:

/append

Se você simplesmente usar `write` de novo no arquivo criado acima, ele vai ser sobrescrito (original apagado). Se você quer acrescentar texto a ele, use `write/append`:

```
>> write/append %myFirstFile.txt "there was a house."
```

Seu arquivo agora tem "Once upon a time...there was a house" .

Escrevendo uma série para um arquivo fazendo de cada elemento da série uma linha do arquivo:

Now lets add another file with 3 lines of text:

```
>> write/lines %mySecondFile.txt ["First line;" "Second line;"
"Third line."]
```

Appending full lines:

```
>> write/append/lines %mySecondFile.txt ["Fourth line;" "Fifth
```

```
line;" "Sixth line."]
```

Seu arquivo agora está assim:

```
First line;
Second line;
Third line.
Fourth line;
Fifth line;
Sixth line.
```

Note que você poderia ter escrito `write/lines/append`. A ordem dos refinamentos não faz diferença.

Substituindo caracteres em um arquivo:

Para substituir caracteres em um arquivo texto, começando na posição n+1, use `write/seek %<file> <n>` :

```
>> write/seek %myFirstFile.txt "NEW TEXT" 5
```

Agora aquele arquivo criado no início do capítulo tem: "Once NEW TEXTime...there was a house."

Refinamentos de Write :

/binary => Preserva o conteúdo exato (binário).

/lines => Escreve cada valor de um bloco em uma nova linha..

/info =>

/append => Escreve no fim do arquivo.

/part => Escrita parcial, dado o número de unidades.

/seek => Escreve a partir de uma posição específica.

/allow => Especifica atributos de proteção.

/as => Escreve em uma codificação específica, o default é 'UTF-8.

function! save

Salva o valor, bloco ou outro tipo de dados para um arquivo, URL, binário ou string.

Diferença entre write e save:

```
>> write %myFourthFile.txt [11 22 "three" "four" "five"]
```

Seu arquivo agora tem: `[11 22 "three" "four" "five"]`

```
>> save %myFourthFile.txt [11 22 "three" "four" "five"]
```

Seu arquivo agora tem: `11 22 "three" "four" "five"`

Um uso importante de `save` é de simplificar o salvamento de scripts Red para que possam ser interpretados usando a ação `do` :

```
>> save %myProgram.r [Red[] print "hello"]
>> do %myProgram.r
hello
```

`do`, `load` e `save` são entendidos melhor se você pensar no console do Red como a tela de um daqueles computadores dos anos 80, rodando alguma variação da linguagem Basic. Você pode `load` o seu programa, salvá-lo com `save` , ou rodá-lo com `do` .

Lendo arquivos

Lendo arquivos como texto:

action! read

```
>> a: read %mySecondFile.txt
== {First line;^/Second line;^/Third line.^/Fourth line;^/Fifth li
```

Agora a palavra (variável) "a" tem todo o conteúdo do arquivo:

```
>> print a
First line;
Second line;
Third line.
Fourth line;
Fifth line;
Sixth line.
```

Lendo arquivos como séries onde cada linha é um elemento:

Note que a palavra (variável) a acima é, por enquanto, só texto com *newlines*. Se você quiser ler o arquivo como uma série, tendo cada linha como um elemento, você deve usar `read/lines`:

```
>> a: read/lines %mySecondFile.txt
== ["First line;" "Second line;" "Third line." "Fourth line;"...

>> print pick a 2
Second line;
```

Read refinements:

/part => Leitura parcial de um dado número de elementos.
/seek => Lê a partir de uma posição específica (source relative).
/binary => Preserva o conteúdo (binário).
/lines => Converte para bloco de strings.
/info =>
/as => Lê com a codificação especificada, o *default* é 'UTF-8.

function! load

Lendo arquivos como séries, onde cada palavra (separada por espaço) é um elemento:

Neste caso, você deve usar `load` ao invés de `read`:

```
>> a: load %mySecondFile.txt
== [First line Second line Third line.
          Fourth line Fifth...

>> print pick a 2
line
```

Lendo e escrevendo arquivos binários:

Para ler ou escrever um arquivo binário como uma imagem ou um som, você deve usar o refinamento `/binary` . O código abaixo carrega uma imagem bitmap para uma palavra (variável) e salva essa imagem com outro nome:

```
>> a: read/binary %heart.bmp
== #{
424D66070000000000003600000028000001E00000014000000010...
>> write/binary %newheart.bmp a
```

Load refinements:

/header => <em discussão>.
/all => Carrega todos os valores e retorna um bloco..
/trap => Carrebga todos os valores, retorna [[values] position error].
/next => Carrega apenas o próximo valor.
/part =>
/into => Põe o resultado em um bloco dado ao invés de criar um novo bloco.
/as => Especifica o tipo de dados. Use NONE para carregar como código.

Funções

Funções devem ser declaradas antes de ser usadas, portanto devem ser escritas no topo do seu programa. Entretando isso não é necessário se uma função é chamada por outra função.

native! func

Variáveis criadas dentro de uma função criada com `func` são **globais**. Elas são vistas por todo o programa.

Uma função é criada com `func` da seguinte forma:

<function name>: func [<argument1> <argument2> ... <argument n>] [<actions performed on arguments>]

```
Red []
mysum: func [a b] [a + b]
print mysum 3 4
```

```
7
```

Demonstrando que as variáveis são **globais**:

```
Red []
mysum: func [a b] [
      mynumber: a + b
      print mynumber
]
mynumber: 20
mysum 3 4
print mynumber
```

```
7
7
```

native! function

`function` faz as suas variaveis **locais**, ou seja, "esconde" as variáveis dentro da função do resto do programa.

Repetindo o programa assima, mas usando `function` ao invés de `func`:

```
Red []
mysum: function [a b] [
      mynumber: a + b
      print mynumber
]
mynumber: 20
mysum 3 4
print mynumber
```

Resultados diferentes:

```
7
20
```

Forçando as variáveis a serem globais, com o refinamento `/extern`:

```
Red []
myfunc: function [/extern a b] [
      a: 22
      b: 33
]
a: 7
b: 9
myfunc
print a
print b
```

```
22
33
```

Definindo o tipo de argumento:

Você pode forçar os argumentos a serem de um determinado tipo de `datatype`:

```
Red []
mysum: function [a [integer!] b[integer!]] [print a + b]
print mysum 3.2 4 ; oops! 3.2 is a float!
```

```
*** Script Error: mysum does not allow float! for its a argument
*** Where: mysum
*** Stack: mysum
```

Você pode permitir vários `datatypes`:

```
Red []
mysum: function [a [integer! float!] b[integer!]] [print a + b]
print mysum 3.2 4
```

```
7.2
```

Ou usar uma classe superior de `datatypes`:

```
Red []
mysum: function [a [number!] b[number!]] [print a + b]
print mysum 3.2 4
```

```
7.2
```

Retornando valores de uma função: native! return

O valor de **retorno** de uma função pode ser ou o último valor avaliado por ela ou um determnado explicitamente pela palavra `return`:

Exemplo com o último valor avaliado:

```
Red []
myfunc: function [] [
      8 + 9
      3 + 3
      print "got here"  ; isso é executado
      10 + 5           ; isso é retornado
]
print myfunc
```

```
got here
15
```

Exemplo com `return` :

```
Red []
myfunc: function [] [
      8 + 9
      return 3 + 3                ; this is returned
      print "never got here"  ; NOT executed
      10 + 5
]
print myfunc
```

```
6
```

Criando seus próprios refinamentos:

Você pode criar refinamentos para suas funções, como os refinamentos nativos do Red: <minhfunção>/<meurefinamento>. Os refinamentos são valores booleanos (true ou false) que são checados pela função:

```
Red []
myfunc: function [a /up   b /down   c] [
      if up [print a + b]
      if down [print a - c]
]
myfunc/up 10 3
myfunc/down 10 3
```

```
13
7
```

Note que argumentos não são obrigatórios para refinamentos.

Atribuindo funções para palavras (variáveis)

Para atribuir uma função a uma palavra (variável) você deve preceder a função com dois pontos: <palavra>: :<função>

```
>> mysum: func [a b] [a + b]
== func [a b][a + b]

>> a: :mysum
== func [a b][a + b]

>> a 3 9
== 12
```

native! does

Se a sua função não precisa de argumentos ou variáveis locais, use a palavra `does` para criá-la:

```
Red []
greeting: does [
      print "Hello"
      print "Stranger"
]

greeting
```

```
Hello
Stranger
```

native! has

Se a sua função não usa argumentos, mas tem variáveis locais, use a palavra `has` para criá-la. `has` transforma o argumento em uma variável local. Compare os três programas abaixo. O primeiro usa has sem argumento, portanto "number" é uma variável global. O segundo dá a "number" um argumento, fazendo-o local. E o terceiro mostra que uma função com argumento **precisa** que o argumento seja enviado pelo evento de chamada:

```red
Red []

myhas: has  [] [
  number:  10 + 33
  print number
]
myhas
print number
```

```
R...
File  Options
43
43
>>
```

```red
Red []

myhas: has  [number] [
  number:  10 + 33
  print number
]
myhas
print number ; error here
```

```
Red Console
File  Options
43
*** Script Error: number ha
s no value
*** Where: print
*** Stack:
>>
```

```red
Red []

myhas: function  [number] [
  number:  10 + 33
  print number ;error here
]
myhas
print number
```

```
Red Console
File  Options
*** Script Error: number ha
s no value
*** Where: print
*** Stack: myhas
>>
```

native! exit

Sai da função sem retornar nenhum número.

Objetos

Um objeto é um "pacote" que agrupa dados e/ou funções, usualmente (sempre?) atribuído a uma palavra (variável). Para acessar um atributo de um objeto em Red, nós usamos a barra (/) como separador. Isto é pouco usual, pois a maoria das linguagens usa um ponto mas, depois que você se acostuma, me parece mais intuitivo já similar a um *path* (caminho).

Criando um objeto:

action! make object! , function! context and function! object

Você pode usar `make object!` , `object` ou `context` para criar um objeto. Eles são o mesmo comando. `object` e `context` são só simplificações de `make object!`.

```
Red []
myobject: object [
      x: 10
      y: 20
      f: function [a b] [a + b]
      name: none
      tel: none
]
myobject/name: "Dimitri"
myobject/tel: #3333-3333
print myobject/x
print myobject/y
print myobject/f 3 5
print myobject/name
print myobject/tel
```

```
10
20
8
Dimitri
3333-3333
```

A computação é feita apenas quando você cria um `object!` (é o constructor). Note que o comando `print` no código abaixo não é executado quando o objeto é acessado:

```
>> myobject: object [print "hello" a: 1 b: 2]
hello
== make object! [
```

```
    a: 1
    b: 2
]

>> myobject/a
== 1
```

Auto-referência:

Quando um objeto precisa fazer uma referência a si próprio, usamos a palavra `self`:

```
Red []
myobject: object [
      x: 10
      y: 20
      f: function [a b] [a + b]
      autoanalisys: does [print self]
]

myobject/autoanalisys
```

```
x: 10
y: 20
f: func [a b][a + b]
autoanalisys: func [][print self]
```

Clonando um objeto e *inheritance* (herança):

Simplesmente atribui um objeto a outro apenas cria um vinculo aos mesmos dados. Se o primeiro muda, o segundo muda também:

```
>> a: object [x: 10]        ;lines of the console deleted for the
sake of clarity.
>> b: a                     ;lines of the console deleted for the
sake of clarity.
>> a/x: 20
== 20

>> b/x
== 20                       ;changed too!
```

Para fazer uma cópia verdadeira e independente, use `copy`:

```
>> a: object [x: 10]        ;lines of the console deleted for the
sake of clarity.
>> b: copy a                ;lines of the console deleted for the
sake of clarity.
```

```
>> a/x: 20
== 20

>> b/x
== 10                          ;NO change! b is a true copy.
```

Se quisermos criar um objeto que herde (inherits) o primeiro objeto, nós usamos `make` **<original object> <new specifications>**:

```
Red []
a: object [x: 3]
b: make a [y: 12]
print b
```

```
x: 3
y: 12
```

find e select - para objetos

`find` simplemente checa se o campo existe, retornando `true` ou `none` .

`select` faz a mesma coisa, mas se o campo existe, retorna seu valor.

```
Red []
obj: object [a: 44]
print find obj 'a
print select obj 'a
print find obj 'x
print select obj 'something
```

```
true
44
none
none
```

Note que ambos buscam a palavra (indicado pelo símbolo ' antes), não pela própria variável. Para acessar a variável, se usa um simples *path* como `obj/a`.

Programação Reativa

A Programação Reativa cria um mecanismo interno que atualiza coisas automaticamente quando um tipo especial de objeto é modificado. Não há necessidade de chamar funções ou subrotinas para fazê-lo. Você muda o objeto A e alguma coisa B é automaticamente mudada também.

Reactor: é o objeto que, quando modificado, dispara as mudanças. É criado por `make reactor!` .

Expressão Reativa: é alterada quando o reacto muda. É criada por `is` .

action! make reactor! e op! is

Exemplo bem básico de Programação Reativa::

```
Red[]

a: make reactor! [x: ""]      ;reactor object - dispara mudanças quando
alterado
b: is [a/x]                   ;expressão reativa - muda quando "a" muda

forever [
      a/x: ask "?"            ;aqui nós entramos um valor para o campo
'x' de 'a'
      print b                 ;aqui nós imprimimos 'b' e... surpresa!
ele mudou!
]
```

```
?house
house
?fly
fly
?bee
bee
```

Um reactor pode atualizar a si mesmo:

```
Red[]

a:  make reactor! [x: 1 y: 2 total: is [x + y]]

forever [
      a/x: to integer! ask "?"
      print a/total
]
```

```
?33
35
?45
47
```

Cuidado para não criar um loop infinito. Isto acontece quando uma mudança diaspara uma mudança em sí mesma.

deep-reactor!

Assim como `copy` tem o refinamento `/deep` para alcançar valores aninhados (blocos dentro de blocos), também o `reactor!` tem esse refinamento.

Este programa deveria repetir o que você digita no console, **mas não funciona**:

```
Red[]

a: make reactor! [z: [x: ""]]
b: object [w: is [a/z/x]]
b/w: "no change"

forever [
      a/z/x: ask "?"
      print b/w
]
```

```
?house
no change
?blue
no change
```

Entretanto, se você muda para `deep-reactor!`:

```
Red[]

a: make deep-reactor! [z: [x: ""]]
b: object [w: is [a/z/x]]
b/w: "no change"

forever [
      a/z/x: ask "?"
      print b/w
]
```

```
?house
house
?blue
blue
```

function! react

Esta é a palavra pré-definida para criar GUIs reativas. Por favor, veja em GUI/Tópicos avançados.

Copiado e colado da documentação:

function! clear-reactions

Remove todas as reações definidas, incondicionalmente.

function! react?

Checa se um campo de um objeto é uma fonte de reação. Se for, retorna a primeira reação encontrada onde o campo do objeto está presente como fonte, caso contrário, retorna `none` . O refinamento `/target` checa se o campo é um alvo, ao invés de uma fonte, e retorna a primeira reação cujo alvo é esta fonte ou `none` se não for encontrada.

/target => Checa se é um alvo ao invés de uma fonte.

function! dump-reactions

Mostra todas as reações registradas para *debug*.

Interface com o Sistema Operacional

native! call

Executa um comando em *shell*. Na maior parte dos casos, é a mesma coisa que escrever no commando prompt (CLI), mas tem algumas particularidades.

O código abaixo abre o Windows Explorer:

```
>> call "explorer.exe"
== 11272              ; este é o número do processo aberto.
```

Isso também funciona:

```
>> str: "explorer.exe"
== "explorer.exe"

>> call str
== 11916
```

Entretanto, o código abaixo cria o processo, mas não abre o Notepad na tela:

```
>> call "notepad.exe"
== 4180
```

Se você busca um comportamento mais próximo a digitar o comando no shell, você deve usar o refinamento `/shell`:

```
>> call/shell "notepad.exe"       ;abre o notepad na tela
== 6524
```

Outros refinamentos:

/wait

Roda o comando e espera até o comando terminar sua execução. Cuidado: se você usar /wait em um commando que você não pode terminar (como `call "notepad.exe"` acima), o Red vai esperar... e esperar... indefinidamente.

/input - nós fornecemos uma string!, um file! ou um binary!, que vai ser redirecionado

para stdin.

Não entendo este comando, me parece a mesma coisa que simplesmente call, uma vez que nós fornecemos a string ou o arquivo de qualquer forma.

/output

Nós fornecemos uma string! , file! ou um binary! que vai receber o stdout redirecionado do comando. Note que o output é *appended*.

O código a seguir cria uma arquivo de texo ou o output do shell para o comando "dir" (uma lista de todos os arquivos e pastas do *path* corrente):

```
>> call/output "dir" %mycall.txt
== 0
```

Este cria uma (longa) string com os resultados de "dir":

```
>> a: ""
== ""

>> call/output "dir" a
== 0

>> a
== { Volume in drive C has no label.^/ Volume Serial Number is BC5
;...
```

/show

Força a exibição da janela de shell (Windows only). Seu script vai rodar com o *command prompt* do Windows aberto.

```
>> call/shell/show "notepad.exe"
== 12372
```

/console

Roda o comando com o I/O redirecionado para o console. Por enquanto funciona apenas se você está rodando o Red do CLI, não funciona no console GUI normal do Red.

native! write-clipboard & read-clipboard

Escreve e lê do clipboard do sistema operacional:

```
>> write-clipboard "You could paste this somewhere you find useful"
```

```
== true

>> print read-clipboard
You could paste this somewhere you find useful
```

I/O

Não está disponível ainda no Red 0.63. Planejado para o Red 0.7

I/O - HTTP

Eu criei alguns arquivo no servidor do helpin.red para fazer testes com HTTP I/O:

http://helpin.red/samples/samplescript1.txt - um loop simples sem o *header* (cabeçalho) do Red (`repeat i 3 [prin "hello " print i]`);

http://helpin.red/samples/samplescript2.txt - um loop simples com *header* (`Red[] repeat i 3 [prin "hello " print i]`);

http://helpin.red/samples/samplehtml1.html - uma página HTML de exemplo.

```
>> print read http://helpin.red/samples/samplescript1.txt
repeat i 3 [prin "hello " print i]

>> print read http://helpin.red/samples/samplescript2.txt
Red[] repeat i 3 [prin "hello " print i]
```

De dentro um script ou usando o console, você pode executar código de um servidor remoto:

```
>>  do read http://helpin.red/samples/samplescript1.txt   ;without
header
hello 1
hello 2
hello 3
```

Se o código no servidor remoto tiver o *header* Red, você poderá executá-lo diretamente, sem a instrução `read`:

```
>> do http://helpin.red/samples/samplescript2.txt    ;with Red [] header
hello 1
hello 2
hello 3
```

Você pode carregar dados ou código, incluindo funções e objetos:

```
>> a: load http://helpin.red/samples/samplescript1.txt
== [repeat i 3 [prin "hello " print i]]
>> do a
hello 1
hello 2
```

```
hello 3
```

Arquivos HTML também podem ser acessados para processamento. Dê uma olhada no [examplo usando dialeto parse](#).

```
>> print read http://helpin.red/samples/samplehtml1.html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>testHtmlPage</title>
</head>
<body>
...
</html>
```

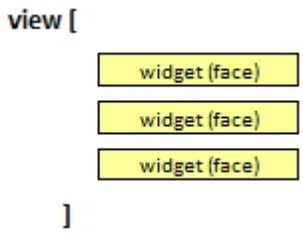
A ser terminado... um dia.

GUI - Visão geral

Os próximos capítulos vão descrever em detalhes cada um dos elementos do View Graphic Engine e do dialeto VID do Red (**faces, facets, definições do container, comandos de layout** e **refinamentos do view**) mas eu acho que ter uma visão geral de como o Red cria GUIs ajuda a entender melhor como esses elementos se relacionam.

Um começo simples:

O Red cria GUIs descrevendo-as em um bloco de view. Essa descrição é simples e direta e, em sua forma mais simples, seria assim:



Se você vai compilar o seu script, você precisa adicionar "needs: view" no header do Red. Se você for rodar o script do console GUI, isto não é estritamente necessário, uma vez que o módulo View já está presente.

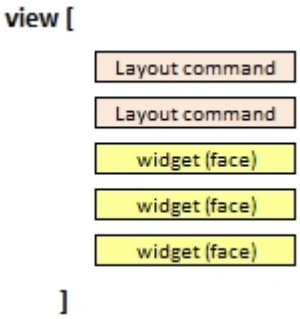
An example code of that:

```
Red [needs: view]  ; "needs: view" é necessário se o script for ser
compilado.

view[
      base
      button
      field
]
```

A GUI resultante:

A documentoação do Red chama coisas como botões e campos de "**faces**" (chamados "widgets" em algumas linguagens). Essas **faces** são colocadas em um **layout** dentro de um **container** (window)

Existem palavras pré-definidas (**comandos de layout**) que definem como as **faces** serão exibidad neste **layout**. Estes comandos devem ser colocados antes das faces que eles alteram:

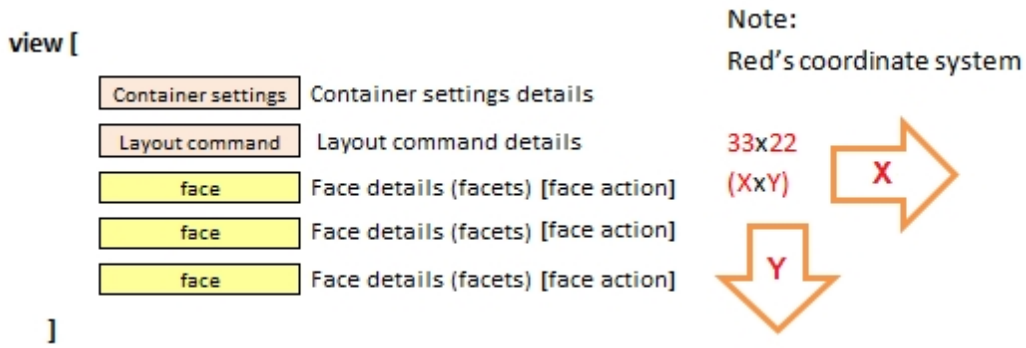


No exemplo a seguir, `below` (um **comando de layout**) diz ao Red para colocar as faces embaixo uma da outra, ao invés do *default* `across` do exemplo anterior:

```
Red [needs: view] ; "needs: view" is needed if the script is going to be
compiled

view[
      below ; layout command
      base  ; face (widget)
      button      ; face (widget)
      field ; face (widget)
]
```

A GUI resultante:

Existem ainda os as **definições do container**, que descrevem como a própria janela deve parecer. E ambos, **definições de container** e **comandos de layout** podem permitir maior detalhamento, como tamanho, cor etc. As **faces** não apenas permitem esse detalhamento (chamado **facets** no jargão do Red), como ainda permitem a inclusão de um bloco de comandos a ser executado pela **face** (chamado "**action facet**") em caso de um evento, como o clicar de um botão.



Código de exemplo:

```
Red [needs: view]

view[
      backdrop blue     ;definição de container
      below             ; comando de layout
      base 20x20        ; face e facet
      button 50x20 "press me" [quit]      ; face, facets  e action facet
      field red "field"       ; face e facets
]
```

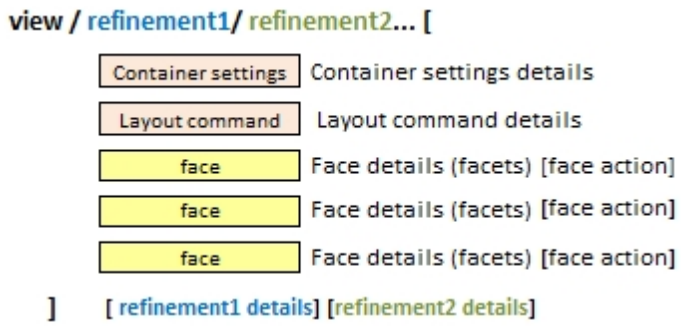
E a GUI resultante:

O Red entende o que fazer com uma **facet** simplesmente pelo seu `datatype!`. Assim, se ele vê um `pair!` , ele sabe que é o tamanho da face, se ele vê um `string!` ele sabe que é o texto a ser exibido. Uma consequência interessante disso é que...

```
button 50x20 "press me" [quit]
button "press me" [quit] 50x20
button [quit] 50x20 "press me"
```

... são todos a mesma coisa, isto é, they result in the same GUI.

A palavra pré-definida (comando) `view` permite refinamentos que vão mudar a própria janela (não o layout dentro dela). Os refinamentos são descritos em blocos codificados após o bloco principal da `view` e devem ser codificados na mesma ordem que foram declarados naquele comando.



No script a seguir, **flags** diz ao Red que a janela é do tipo **modal** e redimensionável, enquanto o refinamento **options** faz a janela aparecer no topo esquerdo da tela, 50 pixels para baixo e 50 pixels para a esquerda:

```
Red [needs: view]

view/flags/options[
      size 300x100      ;container setting
      below             ; layout command
      base 20x20        ; face and facet
      button 50x20 "press me" [quit]      ; face, facets  and actor
      field red "field"       ; face and facets
]['modal 'resize] [offset: 50x50] ; flags and options
```

A GUI resultante:

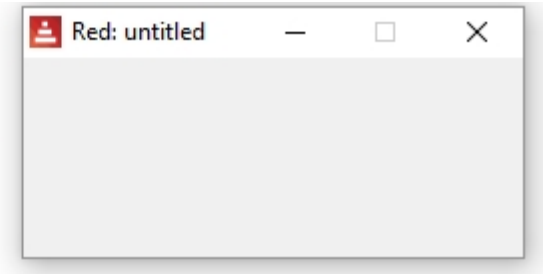
GUI - Definições do container

Estas definem as características da janela que vai conter os elementos da GUI.

VID DLS size

Determina o tamanho da janela em pixels

```
Red [needs: 'view]

view [
    size 250x100]
```



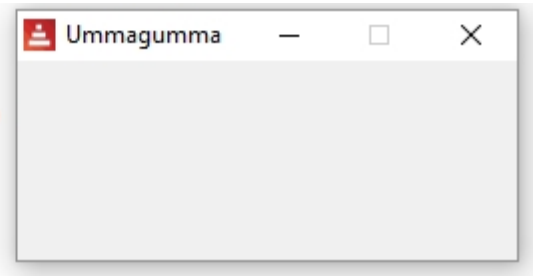
Se você não der um `size`, o Red o faz automaticamente.

Uma observação interessante é que, a não ser que a janela seja grande o suficiente para mostrar parte do título, você não consegue movê-la (*drag*).

VID DLS title

Determina o título no topo da janela.

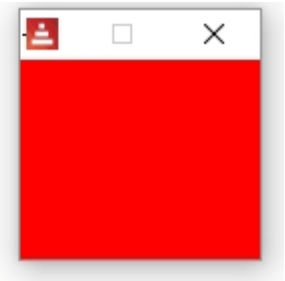
```
Red [needs: 'view]

view [
    title "Ummagumma"
    size 250x100]
```



VID DLS backdrop

Determina a cor de fundo da janela.

```
Red [needs: 'view]

view [
    backdrop red]
```



actors

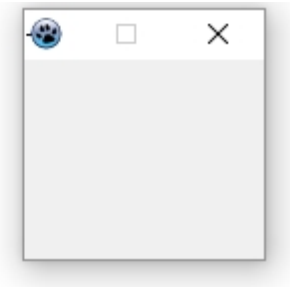
- Veja o capítulo específico.

Definindo um *icon*

Isto só funciona se você compilar o código. Não funciona no modo interpretado.

Não é uma definição de container, mas eu acho que se encaixa aqui. Se você quer colocar o seu próprio ícone numa janela do Red, adicione `icon: <path-to-icon>` após o `needs: 'view` no bloco inicial do Red.

```
Red [needs: 'view
     icon: %project1.ico]

view []
```



Refinamentos

Containers (janelas) permitem os refinamentos **options, flags** e **no-wait.** Os refinamentos **options** e **flags** são definidos em blocos após o bloco principal do `view` .

Options

No refinamento **options** você pode determinar o offset e o tamanho (size). O tamanho parece poder ser definido de duas maneiras, como uma definição de container ou como uma option.

Offset determina onde sua janela vai aparecer, medido to topo esquerdo da tela.

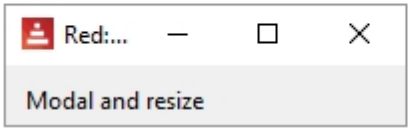
flags

modal - uma janela modal. Demanda atenção, desabilita todas as outras janelas até você fechá-la.

Nota: se você criar uma janela que é modal e no-title/no-border, é bem difícil se livrar dela. Acho que só com o *Task M*anager.

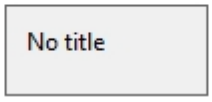
resize - a janela pode ser redimensionada.

```
Red [needs: 'view]
View/flags [ size 200x30 text "Modal and resize" ] [modal resize]
```



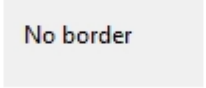
no-title - resulta em uma moldura retangular sem título e sem botões.

```
Red [needs: 'view]
View/flags [ text "No title" ] [no-title]
```



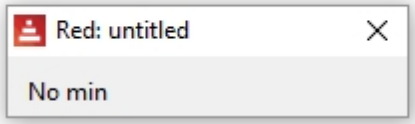
no-border - resulta em uma moldura retangular sem título, sem botões e sem borda.

```
Red [needs: 'view]
View/flags [ text "No border" ] [no-border]
```



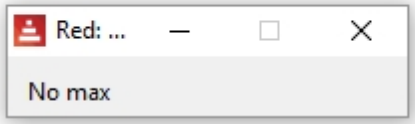
no-min - só o botão de fechar é mostrado no topo da janela.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No min" ] [no-min]
```



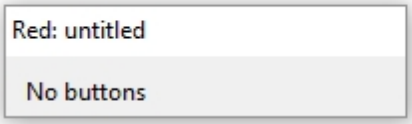
no-max - O botão de maximizar é mostrado como inativo.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No max" ] [no-max]
```



no-buttons - sem botões.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No buttons" ] [no-buttons]
```



popup - só em Windows - faz que a janela seja um popup. Tem um estilo especial (apenas botão de fechar) e permite que outras janelas permaneçam ativas. Fecha se você mudar o foco para outra janela.

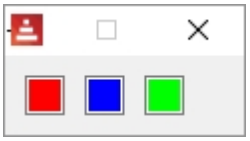
no-wait

Da documentação: "View: Render on screen a window from a face tree or a block of VID code. Enters an event loop **unless /no-wait refinement is used**.

GUI - Layout commands

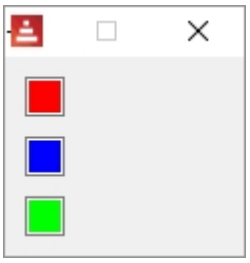
VID DLS across

```
Red [needs: view] ; "needs: view" is needed if the script is going to be compiled

view [
	across
	area 20x20 red
	area 20x20 blue
	area 20x20 green
]
```



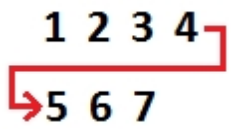
VID DLS below

```
Red [needs: view]

view [
	below
	area 20x20 red
	area 20x20 blue
	area 20x20 green
]
```



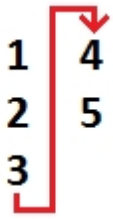
VID DLS return

`return` no modo *across*:

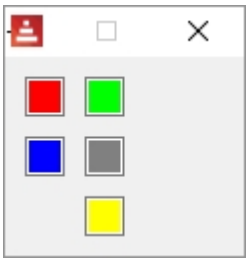



```
Red [needs: view]; "needs: view" is needed if the script is going to be
compiled

view [
	across
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
	area 20x20 yellow
]
```

`return` no modo *below*:



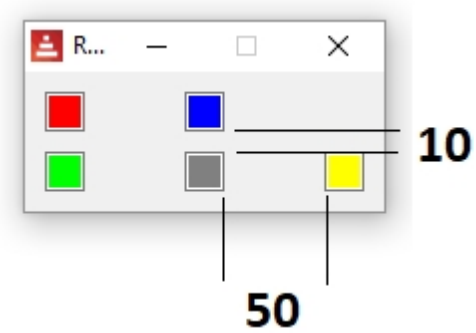

```
Red [needs: view]

view [
	below
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
	area 20x20 yellow
]
```



VID DLS space

Determina o novo offset de espaçamento que vai ser usado para posicionar as próximas faces.

```
Red [needs: view]

view [
	across
	space 50x10
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
	area 20x20 yellow
]
```

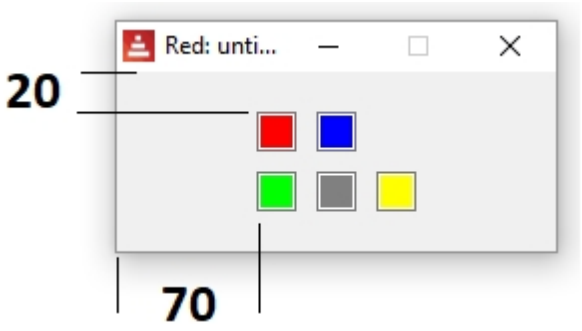


VID DLS origin

Determina o offset da primeira face a aprtir do canto superior esquerdo do painel da janela.

```
Red [needs: view]

view [
	across
	origin 70x20
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
```

```
        area 20x20 yellow
]
```



VID DLS at

Posiciona a próxima face em uma posição absoluta. Este modo de posicionamento apenas afeta a próxima face. Assim, as faces seguintes, colocadas após a face objeto do `at`, vão ser colocadas obedecendo a continuidade do posicionamento.

```
Red [needs: view]

view [
        across
        area 20x20 red
        area 20x20 blue
        return
        area 20x20 green
        at 2x5
        area 20x20 gray
        area 20x20 yellow
]
```

VID DLS pad

Modifica a posição corrente do *layout* com um *offset* relativo. Todas as faces subsequentes são afetadas.

```
Red [needs: view]

view [
        across
        area 20x20 red
        area 20x20 blue
        return
```

```
        area 20x20 green
        pad 10x10
        area 20x20 gray
        area 20x20 yellow
]
```

native! do

Este é o mesmo `do` do capítulo Executando código. Neste caso, é usado para rodar códgio dentro de sua **view**.

Você **pode** fazer isso:

```
Red [needs: 'view]
a: 33 + 12
print a          ;imprime no console
view [
    text "hello"
]
```

Mas isso vai te dar um **error**:

```
Red [needs: 'view]
view [
    text "hello"
    a: 33 + 12        ;ERROR!!!
    print a
]
```

Dentro da view, seu código tem que ficar assim:

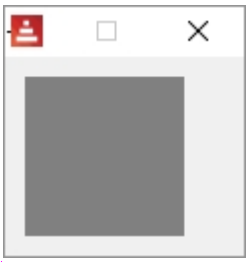
```
Red [needs: 'view]
view [
        text "hello"
        do [a: 33 + 12 print a]       ;OK!
]
```

GUI - Faces

VID DLS base

Face mais básica. Pode ser usada para criar outras faces. Por default, vai mostrar apenas um fundo cinza.

```
Red [needs: view]

view [
      base
]
```



box e image

A bem da verdade, estas não são faces, mas styles da face `base` . box é uma `base` com uma cor transparente e image é uma `base` que espera uma opção `image!` , se nenhuma for forneceida, é mostrada uma imagem vazia com fundo branco.

Nota: o tamanho *default* para `base` e `box` é 80x80, mas para `image`, é 100x100.

```
Red [needs: view]

view [
      base
      box
      image
      image %smallballoon.jpeg
]
```

facets:

Quando o Red interpreta o código e encontra uma **face**, ele procura um dos datatypes a seguir após essa face. Cada um tem um significado que vai mudar a aparência da face mostrada. Seu uso vai ficar claro nos exemplos de faces dados mais adiante.

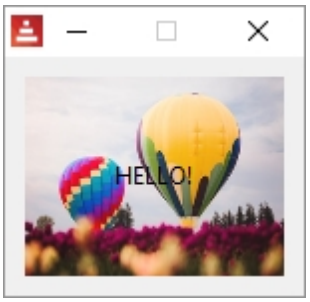
Da documentação:

Datatype	O que faz
integer!	Determina a largura da face.
pair!	Determina a largura e altura face.
tuple!	Especifica a cor de fundo da face.
issue!	Especifica a cor de fundo usando notação hex (#rgb, #rrggbb, #rrggbbaa).
string!	Especifica o texto a ser mostrado pela face.
percent!	Determina a facet `data` (útil para `progress` e `slider`).
logic!	Determina a facet `data` (útil para `check` e `radio`types).
image!	Determina a imagem de fundo.
url!	Carrega o conteúdo apontado pela URL.
block!	Determina a ação para o evento *default* da face.
get-word!	Usa uma função existente como ator.

Uma lista de facets copiada da documentação é dada no final deste capítulo.

Então, usando **facets** com a **face** `base` :

```
Red [needs: view]

view [
      base "HELLO!" 130x100 %balloon.jpeg       ;balloon.jpeg é uma
imagem salva no mesmo...
]                                               ;...diretório que o
executável Red.
```



face text e facet text

Existe uma face chamada text e uma facet também chamada text..

Sobre a facet text: podem ser colocadas na maior parte das faces e podem ser formatadas tanto em estilo como em posição na face. O código a seguir...

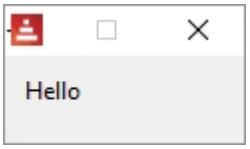
```
Red [needs: view]

view [
      button "hello"
      button "bold" bold
      button "underline" underline
      button "strike" strike
      return
      button "top" 70x70 top
      button "middle" 70x70 middle ;vertical
      button "bottom" 70x70 bottom
      return
      button "left" 70x70 left
      button "center" 70x70 center ;horizontal
      button "right" 70x70 right
      return
      button "mix1" 70x70 top left
      button "mix2" 70x70 top center
      button "mix3" 70x70 top right
      return
      button "No" 70x70 right bold ; does not work!
]
```

... resulta em:

VID DLS text

O evento que dispara a ação defautl é o *click* (veja action facets)

```
Red [needs: view]

view [
      text "Hello"
]
```



Apesar de **h1, h2, h3, h4** e **h5** não serem propriamente **faces** (são styles), Eu acho que devo descrevê-los aqui, já que são faces de texto com tamanhos e fontes diferentes e são bem úteis quando você está trabalhando com texto:

```
Red [needs view]

view [
      below
      h1 "Hello"
      h2 "Hello"
      h3 "Hello"
```

```
        h4 "Hello"
        h5 "Hello"
]
```

O objeto font

Talvez você já tenha tentado atribuir uma cor ao seu texto e notou que simplesmente adicionando, digamos, `blue` após a face `text` apenas faz o fundo ficar colorido, mas nãoa fonte. Para formatar a onte usada para exibir strings emfaces, existe uma coisa na documentação chamada "font object". Pense nela como simplesmente um conjunto de comandos para formatar a fonte. Você os escreve após declarar sua face, junto com as outras facets.

font-name <Nome de fonte válido instalado no sistema operacional>

font-size <Tamano da fonte em *points*>

font-color <Cor da fonte, no formato R.G.B, R.G.B.A ou o nome da cor>

Você também pode adicionar `bold` `italic` `underline` ou `strike`.

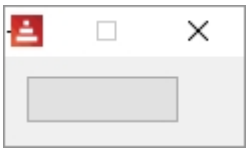
Então:

```
Red [needs: view]

view [
        text "hello" font-name "algerian" font-size 18 font-color red bold
        text "hello" font-name "algerian" font-size 18 font-color blue
        text "hello" font-name "broadway" font-size 15 font-color green
strike
        text "hello" font-name "arial" font-size 12 font-color cyan
underline
]
```

VID DLS button

O evento que dispara a ação default é o *click*.

```
Red [needs: view]

view [
      button
]
```

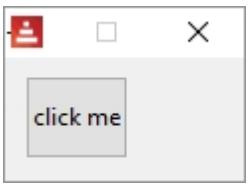


action facets

A maioria das faces permite uma **action facet**, isto é um bloco de comandos que é disparado por um evento. Este evento pode ser um click do mouse (chamado "down" pelo Red) ou outra coisa como pressionar a tecla *enter* ou fazer uma seleção.

Para os buttons, o que dispara a **action facet** é o evento "down" (mouse click) e no próximo exemplo, ele dispara o comando `quit` que encerra o programa.`[quit]` é a action facet (deveria ser chamada **default actor**?, você pode criar seus próprios actors conforme descrito aqui).

```
Red [needs: view]

view [
      button 50x40 "click me" [quit]
]
```



cores

Se você rodar o programa abaixo...

```
Red [needs: view]
view [
      base 30x30 aqua  text "aqua"       base 30x30 beige text "beige"
```

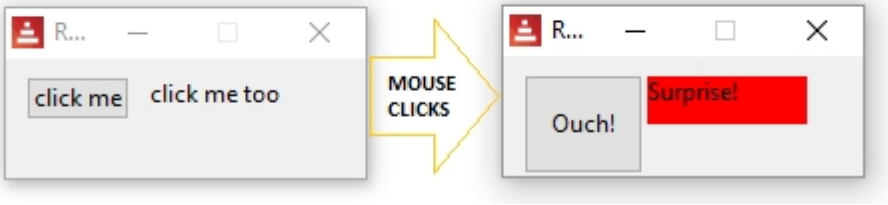
```
        base 30x30 black text "black"     base 30x30 blue   text "blue"

        return
        base 30x30 brick text "brick"      base 30x30 brown text "brown"

        base 30x30 coal text "coal"       base 30x30 coffee   text
    "coffee"
        return
        base 30x30 crimson text "crimson"  base 30x30 cyan text "cyan"

        base 30x30 forest text "forest"   base 30x30 gold text "gold"

        return
        base 30x30 gray text "gray"        base 30x30 green text "green"

        base 30x30 ivory text "ivory"     base 30x30 khaki text "khaki"

        return
        base 30x30 leaf text "leaf"        base 30x30 linen text "linen"

        base 30x30 magenta text "magenta" base 30x30 maroon  text "maroon"

        return
        base 30x30 mint text "mint"        base 30x30 navy text  "navy"

        base 30x30 oldrab text "oldrab"   base 30x30 olive text "olive"

        return
        base 30x30 orange text "orange"    base 30x30 papaya text "papaya"

        base 30x30 pewter text "pewter"   base 30x30 pink text "pink"

        return
        base 30x30 purple text "purple"    base 30x30 reblue text "reblue"

        base 30x30 rebolor text "rebolor"  base 30x30 red text "red"
    ]
```

...você obterá:

faces são objects

Cada face é um clone do objeto padrão `face!` e você pode mudar seus atributos (as facets) durante a execução (runtime):

```
Red [needs: 'view]

view [
    size 180x60
    b: button 50x20 "click me" [b/text: "Ouch!"  b/size: 60x50]
    t: text "click me too"  [t/color: red t/text: "Surprise!"]
    ]
```



Dentro da **action facet**, você pode se referir aos atributos da face usando `face/<atributo>`:

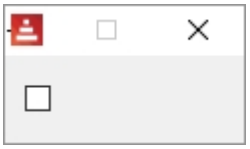
```
Red [needs: 'view]
view [
    button 100 "click me" [face/text: "I was clicked"]
    ]
```

Rode o script abaixo e clique no botão para você ter uma idéia da complexidade de uma face como "button":

```
Red [needs: view] view [b: button [print b]]
```

VID DLS check

```
Red [needs: view]
view [
      check
]
```



O evento que dispara a action facet é o *change.* O estado corrente do está no atributo */data (*`true` *ou* `false`*)*

```
Red [needs: 'view]
view [b: check "unchecked"[either b/data        ;Red's if-else
                              [b/text: "checked"]    ;if "data" is true!
                              [b/text: "unchecked"] ;if "data" is false!
                          ]
    ]
```

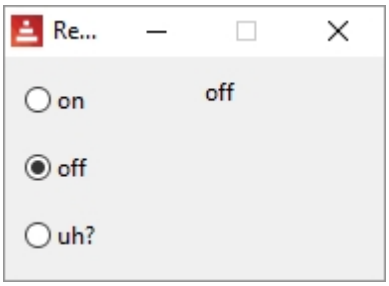
A propósito, o estilo do programa acima não é o correto, apenas me pareceu mais didático. Veja o Red's Coding Style Guide.

VID DLS radio

O evento que dispara a action facet é o *change.* O estado corrente está no atributo */data*

Cria um botão de rádio, com um texto opcional mostrado à esquerda ou a direita. Apenas um botão de rádio por painél pode estar ativado.

```
Red [needs: view]
view [
      r1: radio "on" [t/text: "on"]
      t: text "none"
      return
      below
      r2: radio "off" [t/text: "off"]
      r3: radio "uh?" [t/text: "uh?"]
]
```

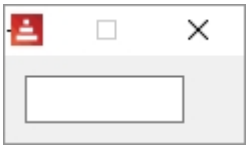


VID DLS field

Campo para entrar com texto.

O evento que dispara a action facet é o *enter*. The events that triggers the action facet is *enter.* O estado corrente (texto no campo) está no atributo */data*. Isto funciona nos dois sentidos, você pode ler o atributo mas também pode mudá-lo durante a execução. Entretanto se você tentar mudar /data durante a execução com códgio dentro do bloco de view mas fora da action facet, será gerado um erro.

```
Red [needs: view]
view [
      field
]
```

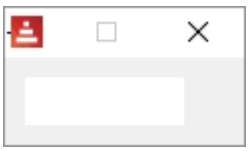


Este exemplo imprime seu input no console quando você aperta enter::

```
Red [needs: view]
view [
      f: field [print f/text]
]
```

`field` permite uma facet `no-border` *:

```
Red [needs: view]
view [
      f: field no-border
]
```

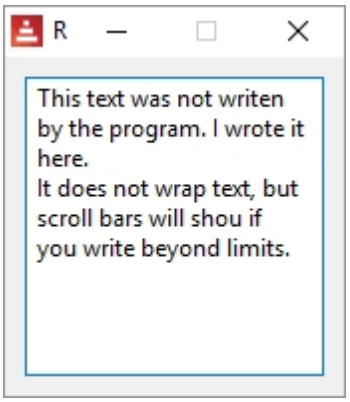


*Para sua informação, a documentação chama `no-border` de "flag", não de facet.

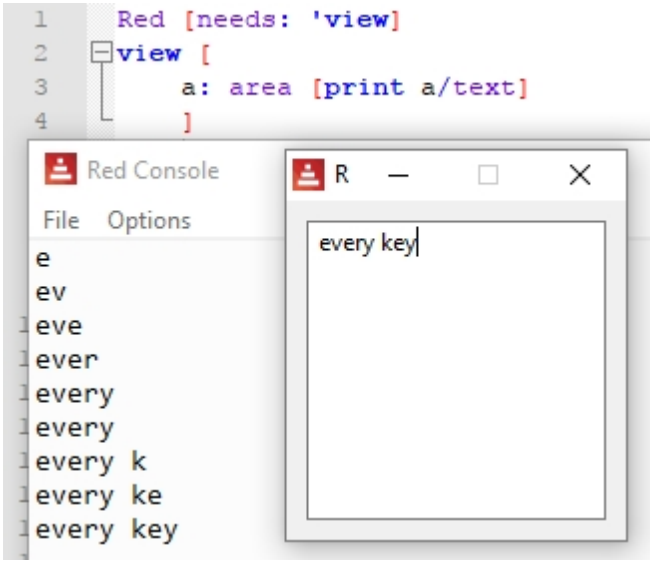
VID DLS area

O evento que dispara a action facet é o *change.* O texto dentro da `area` fica no atributo */text.* Você pode mudar o texto atribuindo strings para */text*.

```
Red [needs: view]
view [
      area
]
```



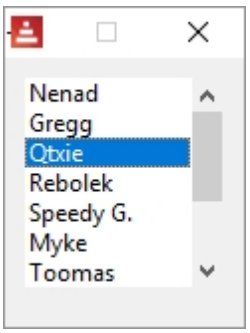
Uma vez que cada *change* (mudança) dispara um evento, cada tecla digitada dentro da `area` executa a action facet:



VID DLS text-list

O evento que dispara a action facet é *selection.* As strings a serem listadas estão no atributo `/data`. O índice dos dados (string) selecionados estão no atributo `/selected`

```
Red [needs: view]
view [
      tl: text-list 100x100 data[
      "Nenad" "Gregg" "Qtxie" "Rebolek"
      "Speedy G." "Myke" "Toomas"
      "Alan" "Nick" "Peter" "Carl"
      ]
      [print tl/selected]
]
```




```
3
```

Para usar a string selecionada, um possível snipped de código seria:

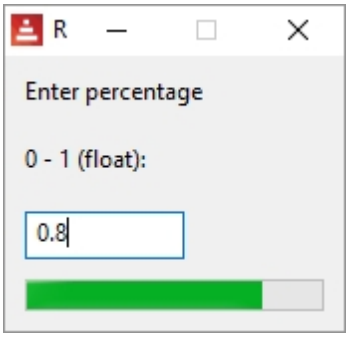
```
pick face/data face/selected
```

Isto é a mesma coisa que: `pick ["Nenad" "Greg" "Qtxie" "Rebolek" (...)] 3`

VID DLS progress

Acho que não permite uma action facet, é só um *display*. O estado corrente está no atributo `/data`, como um `percent!` ou um `float!` entre 0 e 1.

```
Red [needs: view]
view [
	below
	text "Enter percentage"
	text "0 - 1 (float):"
	field [p/data: to percent! face/data]
	p: progress
]
```



VID DLS slider

O evento que dispara a action facet é *change*. A percentagem corrente está no atributo `/data`, como um datatype `percent!`.

```
Red [needs: view]
view [
	title "slider"
	t: text "Percentage"
	slider 100x20 data 10% [t/text: to string! face/data]
]
```

Mova o cursor do slider para ver o valor percentual:

VID DLS panel

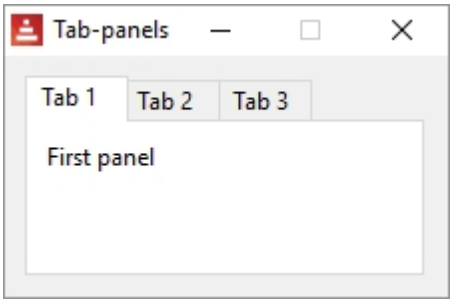
Cria uma nova área onde você pode exibir faces usando a sintaxe explicada até aqui. Acho que o exemplo abaixo é auto-explicativo. Me parece que não permite action facets.

```
Red [needs: view]
view [
	panel red [size 100x120 below text red "Panel 1" check button
"Quit 1" [quit]]
	panel gray [size 100x120 below text gray "Panel 2" check button
"Quit 2" [quit]]
]
```

VID DLS tab-panel

Cria um conjunto de painéis dos quais só um pode ser exibido num dado momento, selecionado por uma aba. Me parece que não permite action facets. Os dados ficm em : `<tab-panel>/data` - Bloco de nomes de tabs (strings).
`<tab-panel>/pane` - Lista de painéis correspondentes a lista de tabs (block!).
`<tab-panel>/selected` - Índice do painel selecionado ou `none` (integer!) (read/write). Isto é, o painel que tem o foco, 1 para o primeiro, 2 para o segundo e assim por diante.

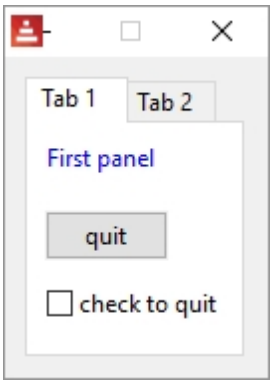
```
Red [needs: view]
view [
	Title "Tab-panels"
	tab-panel 200x100 [
		"Tab 1 " [text "First panel"]
		"Tab 2 " [text "Second panel"]
		"Tab 3 " [text "Third panel"]
	]
]
```



E cada painel permite um conjunto de faces:

```
Red [needs: view]
view [
	Title "Tab-panels"
```

```
        tab-panel 110x140 [
                "Tab 1 " [
                below
                text font-color blue "First panel"
                button "quit" [quit]
                check "check to quit" [quit]
                ]
                "Tab 2 " [text "Second panel"]
        ]
]
```

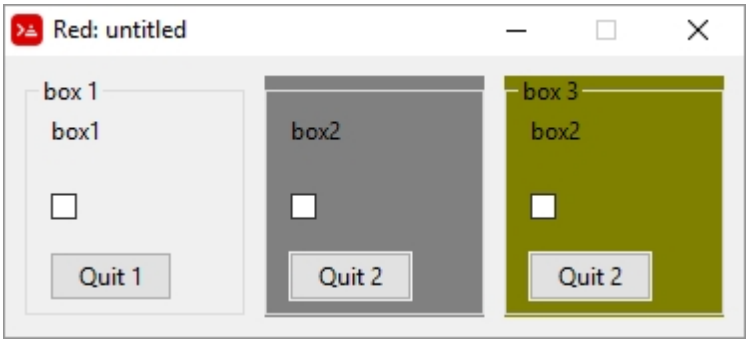


VID DLS group-box

Da documentação: Um group-box é um container de outras faces, com uma moldura visível ao ser redor. Este é um estilo temporário que vai ser removido quando o Red tiver a facet edge.

Me parece ser só um painel com moldura.

```
Red [needs: view]
view [
        group-box "box 1" [size 110x120 below text "box1" check button
"Quit 1" [quit]]
        group-box gray [size 110x120 below text "box2" check button "Quit
2" [quit]]
        group-box "box 3" olive [size 110x120 below text "box2" check
button "Quit 2" [quit]]
]
```



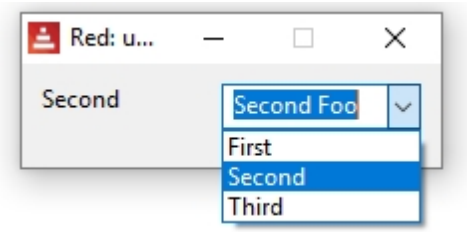
VID DLS drop-down

O evento que dispara a action facet é *enter*.

Da documentação: "Este tipo representa uma lista vertical de strings, mostradas em uma moldura colapsável. Uma scrollbar vertical aparece automaticamente se o conteúdo não cabe na moldura. A facet `data` aceita valores arbitrários, mas somente strings serão adicionados à lista a ser exibida. Valores extras (não strings) podem ser usados para criar arrays associativos, usando as strings como chaves. A facet `selected` é um índice de base 1 indicando a posição da string selecionada na lista, mas não na facet `data`."

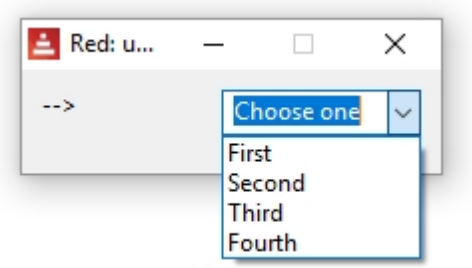
Você pode digitar na caixa de texto. O conteúdo da caixa de texto vai estar no atributo `/text` depois de você digitar "enter".

```
Red [needs: view]
view [
      t: text "-->"
      drop-down "Choose one" data [
            "First"
            "Second"
            "Third"
      ] [ t/text: pick face/data face/selected ]
]     ;precisa apertar enter para mudar o texto
```



Exemplo usando events:

```
      Red [needs: view]
      view [
            t: text "-->"
            drop-down "Choose one" data ["First" "Second" "Third" "Fourth"]
            on-change [ t/text: pick face/data face/selected ]
      ]
```

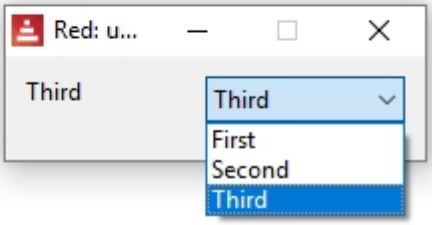


VID DLS drop-list

O evento que dispara a action facet é *change.*

Similar ao drop-down, mas você não pode escrever na caixa de texto e não mostra um texto *default*.

```
Red [needs: 'view]

view [
    t: text "--->"
    drop-list "Choose one" data [
        "First"
        "Second"
        "Third"][
        t/text: pick face/data face/selected
    ]
]
```



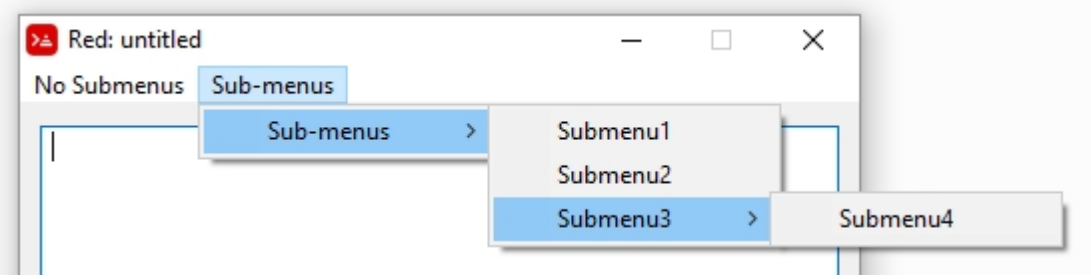
VID DLS menus

`menu` é uma `facet`, mas me parece que quem aprende Red quer saber "quais são as *widgets* disponíveis" e `menu` tem cara e jeito de *widget*. Assim eu acho que deve constar aqui, como `face`, mesmo que tecnicamente seja outra coisa.

É pouco documentada. Toomas Voograid gentilmente forneceu alguns exemplos de menus. O primeiro é a reescrita de um exemplo tirado do site do Nick Antonaccio's [Short Red Code Examples](#).

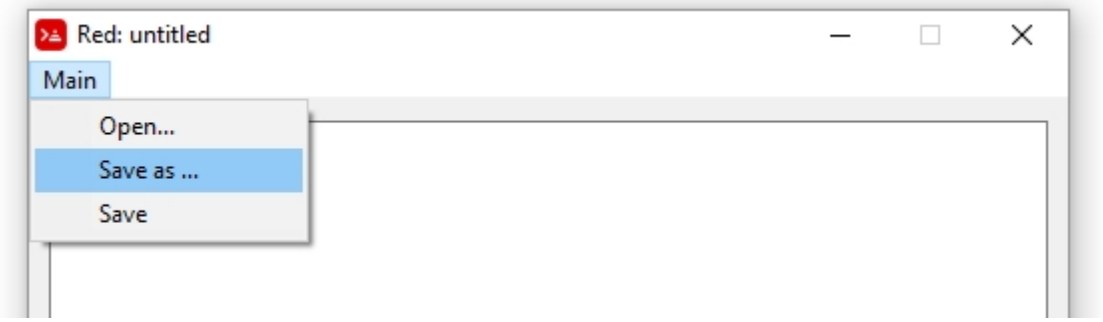
```
Red [needs: view]
view/options [area 400x400] [
    menu: [
        "No Submenus" [
            "Print" prnt
            ---
            "Quit" kwit
        ]
        "Sub-menus" [
            "Sub-menus" [
                "Submenu1" s1
                "Submenu2" s2
                "Submenu3" [
                    "Submenu4" s4
                ]
            ]
        ]
    ]
    actors: make object! [
        on-menu: func [face [object!] event [event!]][
            if event/picked = 'kwit [unview/all]
            if event/picked = 'prnt [print "print menu selected"]
            if event/picked = 's4 [print "submenu4 selected"]
```

```
        ]
    ]
]
```



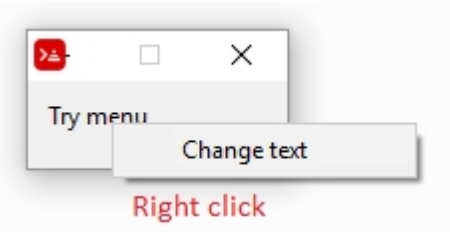
The second example is a simple framework of a text editor using menus:

```
Red [title: "Menus" needs: 'view]

view/options [editor: area 500x300][
   menu: ["Main" ["Open..." open "Save as ..." save-as "Save" save]]
   actors: object [on-menu: func [face event /local new-name][switch
event/picked [
      open [if new-name: request-file [editor/text: read editor/extra:
new-name set-focus editor]]
      save-as [if new-name: request-file/save [write editor/extra: new-
name editor/text]]
      save [write editor/extra editor/text]
]]]]
```



The third example makes a menu appear when you right-click on text:

```
Red [needs: view]
view [text "Try menu" with [
    menu: ["Change text" change]
    actors: object [on-menu: func [f e][
        switch e/picked [change [
            view/flags [text "Please enter new text:" field [
                f/text: face/text unview
            ]][modal]
]]]]]]
```



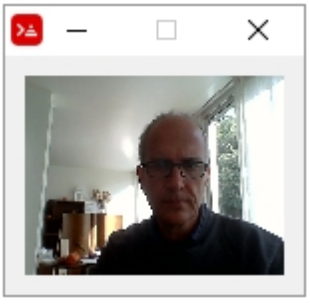
This last example can be rewritten using `on-menu` event:

```
Red [needs: view]
view [
    text "Try menu"
    with [menu: ["Change text" change]]
    on-menu [
        f: face
        if event/picked = 'change [
            view/flags [
                text "Please enter new text:"
                field [f/text: face/text unview]
            ][modal]
        ]
    ]
]
```

VID DLS camera

Acessa um *stream* de camera.

```
Red []
view [
    cam: camera 130x100 select 1
]
```



Este script salva uma foto do stream da câmera como uma imagem .jpeg:

```
Red []
count: 0
snapshot: does [
    load rejoin [mold '% 'picture count: count + 1 '.jpeg]
]
view [
    cam: camera 120x100 select 1
    button "Save picture" [save/as snapshot to-image cam 'jpeg]
```

```
]
```

Facets de acordo com a documentação do Red:

Facet	Datatype	Obrigatório ?	Aplicação	Description
type	word!	yes	all	Tipo de componente gráfico
offset	pair!	yes	all	Posição de *offset* partindo do topo esquerdo da janela-mãe.
size	pair!	yes	all	Tamanho da face.
text	string!	no	all	Texto mostrado na face.
image	image!	no	some	Imagem mostrada no fundo da face.
color	tuple!	no	some	Cor de fundo no formato R.G.B ou R.G.B.A.
menu	block!	no	all	Barra de menu ou menu contextual.
data	any-type!	no	all	Content data da face.
enabled?	logic!	yes	all	Habilitar ou desabilitar eventos de input na face.
visible?	logic!	yes	all	Mostras ou ocultar a face.
selected	integer!	no	some	Para listas, o índice do elemento escolhido corrente.
flags	block!, word!	no	some	Lista de palavras chaves alterando a exibição ou o comportamento da face.
options	block!	no	some	Propriedes extras da face no formato [name: value].
parent	object!	no	all	Retro-referência à face-mãe (se houver).
pane	block!	no	some	Lista de faces-filhas exibidas dentro da face.
state	block	no	all	Informação interna sobre o estado da face (usada só pelo *Viewengine).*

rate	integer!, time!	no	all	Timer da face. Um integer! determina frequencia, um time! determina duração, none encerra.
edge	object!	no	all	*(reservado)*
para	object!	no	all	Referência para o objeto de posicionamento de texto.
font	object!	no	all	Referência para o objeto de formatação da fonte.
actors	object!	no	all	Handler de eventos fornecido pelo usuário.
extra	any-type!	no	all	Uso livre - dados opcionais do usuário ligados à face.
draw	block!	no	all	Lista de comandos de Draw a serem desenhados na face.

GUI - Events e actors

Eventos:

Clicar no mouse, passar o cursor sobre alguma coisa, apertar teclas etc., são eventos que muitas vezes devem ser associados com código. No último capítulo nós vimos que existe algo chamado **action facet** que executa código quando disparado por algum evento. Você pode adicionar mais blocos associados a eventos usando o seguinte leiaute:

```
view [
        face  facet facet [action facet]
                on-event [action]
                on-event [action]
        face2 facet facet [action facet]
                on-event [action]
                on-event [action]
    ]
```

Existe uma extensa lista de eventos possíveis na documentação. Ela está copiada no fim deste capítulo para referência.

Cada face aceita um conjunto de eventos, isto é, nem todos eventos se aplicam a todas as faces.

Eu fiz uma lista curta de eventos. Eu não vejo razão para dar um exemplo de cada evento existente, uma vez que a lógica é a mesma:

down - botão esquerdo do mouse foi pressionado;

over - cursor so mouse passando sobre uma face;

```
Red [needs: view]
view [
        t: area 40x40 blue
        on-down [quit]
        on-over [either t/color = red [t/color: blue][t/color: red]]
]
```

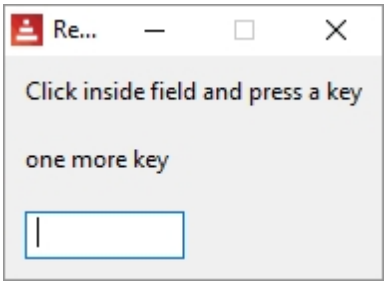
wheel - roda do mouse sendo acionada;

```
Red [needs: view]

list: ["first" "second" "third" "fourth"]
view [
        t: text "Place your cursor over here and roll the wheel"
                on-wheel [
                        t/text: first list
                        list: next list
                        if tail? list [list: head list]
```

```
            ]
]
```

key-down - uma tecla foi pressionada:

```
Red [needs: view]

list: ["key" "another key" "one more key"]
view [
      below
      text "Click inside field and press a key"
      t: text 100
      a: field
            on-key-down [
                  t/text: first list
                  list: next list
                  if tail? list [list: head list]
            ]
]
```



time - o tempo ajustado da facet `rate` da face expirou.

O próximo exemplo "pisca" um texto com um tempo de 1 segundo (veja **rate** no capítulo GUI- Tópicos avançados):

```
Red [needs: view]

view [
      t: text "Now you see..." rate 1
            on-time [either t/text = "" [t/text: "Now you see..."]
[t/text: ""]]
]
```

close - este é um evento de janela (window) disparado quando a janela é fechada. É útil para incluir código que deve ser executado quando o usuário encerra o script.

```
Red [needs: view]

view [
      on-close [print "bye!"]
      button [print "click"]
]
```

Actors

Actors como se chamam as funções de tratamento de eventos (**event handling functions,** ou **event hadlers)** no Red, isto é, o código dentro do bloco depois de `on-<event>` . Então porque não chamá-los de **event handlers** como quase todas as outras linguagens? Eu acho que é porque eles são um objeto dentro da face, como você pode ver se rodar o código abaixo e clicar na **face** `area`:

```
Red [Needs: view]
view [
      t: area 40x40 blue on-down [print t] ;click to quit
      on-over [either t/color = red [t/color: blue][t/color: red]]
]
```

Você vai ver no console, perto do fim da impressão, um objeto com os **actors** descritos:

```
(...)
edge: none
para: none
font: none
actors: make object! [
    on-down: func [face [object!] event [event! none!]][print t]
    on-over: func [face [object!] event [event! none!]][either t/color =
red [t/color: blue] [t/color: red]]
]
extra: none
draw: none
(...)
```

Lista de nomes de eventos names:

Name	Origem	Causa
down	mouse	Botão esquerdo do mouse pressionado.
up	mouse	Botão esquerdo do mouse solto.
mid-down	mouse	Botão central do mouse pressionado.
mid-up	mouse	Botão central do mouse solto.
alt-down	mouse	Botão direito do mouse pressionado.
alt-up	mouse	Botão direito do mouse solto.
aux-down	mouse	Botão auxiliar do mouse pressionado.
aux-up	mouse	Botão auxiliar do mouse solto.

drag-start	mouse	Início do arrastar (drag) de uma face.
drag	mouse	Face sendo arrastada.
drop	mouse	Uma face arrastada foi solta.
click	mouse	Click do botão esquerdo do mouse (apenas na face button).
dbl-click	mouse	Duplo-click esquerdo do mouse.
over	mouse	Cursor do mouse passando sobre uma face. Este evento é disparado uma vez quando o cursor entra na face e outra vez quando o cursor sai da face. Se a facet `flags` tem a flag **all-over** , então todos os eventos intermediários são criados também.
move	mouse	Uma janela foi movimentada.
resize	mouse	Uma janela foi redimensionada.
moving	mouse	Uma janela está sendo movimentada.
resizing	mouse	Uma janela está sendo redimensionada.
wheel	mouse	A roda do mouse está sendo movimentada.
zoom	touch	Um gesto de zoom (pinching) foi reconhecido.
pan	touch	Um gesto de pan (sweeping) foi reconhecido.
rotate	touch	Um gesto de rotação foi reconhecido.
two-tap	touch	Um gesto de duplo toque (double tapping) for reconhecido.
press-tap	touch	Um gesto de press-and-tap foi reconhecido.
key-down	keyboard	Uma tecla foi pressionada.
key	keyboard	Foi dada entrada em um caracter ou uma tecla especial foi pressionada (exceto control, shift ou menu).
key-up	keyboard	Uma tecla pressionad foi solta.
enter	keyboard	Tecla Enter foi pressionada.
focus	any	A face recebeu o foco.
unfocus	any	A face perdeu o foco.
select	any	Uma seleção foi feita em uma face com múltiplas escolhas.
change	any	Uma mudança ocorreu em uma face que aceita entradas de dados do usuário (input de texot ou seleção em uma lista).

menu	any	Uma entrada do menu foi escolhida..
close	any	Uma janela foi fechada.
time	timer	O delay determinado para o `rate` de uma face expirou.

Notas:

- Eventos touch não estão disponíveis para Windows XP.+
- Um ou mais eventos `moving` sempre precedem um do tipo `move` .
- Um ou mais eventos `resizing` sempre precedem um do tipo `resize`.

GUI - Event!, posição do mouse e uso de teclas

Toda a vez que um `event!` acontece em uma face, você pode obter informação sobre ele de `event/`<see list below>.

Mouse position:

Então, no exemplo simples abaixo, nós imprimimos o tipo de evento e as coordenadas do mouse quando um `down` (click) do mouse acontece:

```
Red [needs: view]

view [
    base 100x100
    on-down [
            print event/type
            print event/offset
      ]
]
```

```
down
39x57
down
86x43
```

Key pressed:

Curiosamente, no exemplo acima você só obtém `none!` se você tentar imprimir `event/key`, mas no exemplo abaixo, usando on-key como evento, você obtém não apenas a tecla apertada, mas também as coordenadas do mouse! Na verdade, você obtém as coordenadas do mouse onde quer que ele esteja na tela, referidas ao canto superior esquerdo da face.

```
Red [needs: view]

view [
    area 100x100
    on-key [
            print event/type
            print event/offset
            print event/key
```

```
        ]
]
```

```
key
-59x84
r
key
-36x59
s
key
-116x79
o
```

Algumas faces parecem não gerar eventos de `key`. Por exemplo, se você substituir area por base no exemplo acima, não obterá nenhum resultado no console.

Segue uma lista de eventos da [Documentação oficial do Red]:

Campo	Valor retornado
`type`	Tipo de evento (`word!`).
`face`	Objeto da face onde o evento ocorreu (`object!`).
`window`	Janela onde o evento ocorreu (`object!`).
`offset`	Offset do cursor do mouse relativa à face quando o evento ocorreu (`pair!`). Para eventos de gestos, retorna as coordenadas do ponto central.
`key`	Tecla pressionada (`char! word!`).
`picked`	Novo item selecionado em uma face (`integer! percent!`). Para um evento `down` do mouse em um `text-list`, ele retorna o index do item embaixo do mouse ou `none`. Para um evento `wheel` retorna o número de passos de rotação, um número positivo indica rotação para frente, e vice-versa. Para um evento `menu`, retonra o ID correspondente do menu (`word!`). Para um gesto de zoom, retonra o percentual representando o aumento/redução. Para outros gestos o valor depende do sistema (por enquanto).
`flags`	Retorna uma lista de uma ou mais (`block!`).
`away?`	Retorna `true` se o cursor sai dos limites da face(`logic!`). Só funciona se o evento `over` estiver ativo.
`down?`	Retonra `true` se o botão esquerdo do mouse foi pressionado (`logic!`).

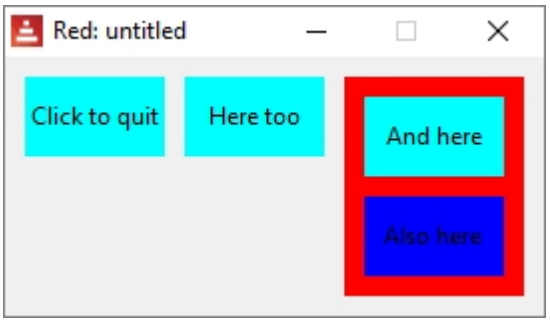
mid-down?	Retonra `true` se o botão central do mouse foi pressionado (`logic!`).
alt-down?	Retonra `true` se o botão direito do mouse foi pressionado (`logic!`).
ctrl?	Retonra `true` se CTRL foi pressionado (`logic!`).
shift?	Retonra `true` se SHIFT foi pressionado (`logic!`).

GUI - Tópicos avançados

VID DLS style

`style` é usado para criar faces personalizadas.

```
Red [Needs: view]

view [
	style myface: base 70x40 cyan [quit]

	myface "Click to quit"
	myface "Here too"
	panel red 90x110 [
		below
		myface "And here"
		myface "Also here" blue
	]
]
```



function! view e function! unview

Múltiplas janelas em uma tela

`view` também pode ser usado para mostrar janelas com faces (a face tree) que foram criadas em outra parte do código. `unview` fecha uma view. O código a seguir cria duas janelas idênticas mas independentes (com face trees diferentes) em partes diferentes da tela:

```
Red [needs: view]
my-view: [button {click to "unview"} [unview]]

print "something"  ;do something else
```

```
print "biding my time" ;do something else

view/options/no-wait my-view [offset: 30x100]
view/options/no-wait my-view [offset: 400x100]
```

`unview` permite o refinamento `/only` para atuar em apenas uma janela:

```
Red [needs: view]

v1: view/options/no-wait [
    backdrop blue
    button "unview blue"[unview/only v1]
    button "unview yellow" [unview/only v2]
][    ;options:
    offset: 30x100
]
v2: view/options/no-wait [
    backdrop yellow
    button "unview blue"[unview/only v1]
    button "unview yellow" [unview/only v2]
][    ;options:
    offset: 400x100
]
```

Refinamentos para `view`:

/tight => offset e origin iguais a zero.
/options =>
/flags =>
/no-wait => Retorna imediatamente, não espera.

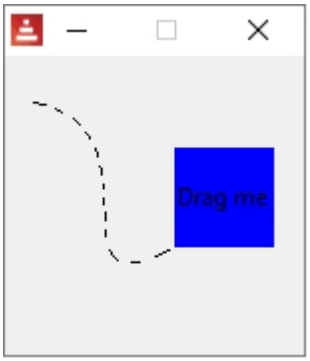
Refinamentos para `unview`:

/all => Fecha todas as views.
/only => Fecha uma dada view.

VID DLS loose

`loose` é uma facet que permite que a face seja arrastada (drag) pelo mouse.

```
Red [needs: view]

view [
      size 150x150
      base blue 50x50 "Drag me" loose
]
```



VID DLS all-over

O evento `on-over` normalmente acontece quando o cursor "entra" ou "sai" da face. Quando usamos a facet `all-over` todo o evento que acontece enquanto o cursor está sobre a face, como movimentos e clicks, gera um evento `on-over`.

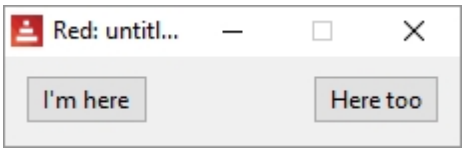
No exemplo a seguir, o quadrado da esquerda só muda de cor quando o cursor entra ou sai, mas o quadrado da direita muda de cor com qualquer movimento sobre ele, bem como com clicks do botão::

```
Red [needs: view]

view [
	a: base 40x40 blue
		on-over [either a/color = red [a/color: blue][a/color: red]]
	b: base 40x40 blue all-over
		on-over [either b/color = red [b/color: blue][b/color: red]]
]
```

VID DLS hidden

Faz a face invisível por default. Um dos usos possíveis é criar uma face oculta com um `rate`, assim você pode ter rotinas temporizadas sem a necessidade de mostrar uma face.

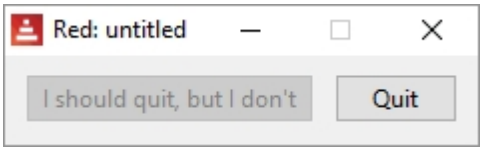
```
Red [needs: view]

view [
	button "I'm here"
	button "I'm not" hidden
	button "Here too"
]
```



VID DLS disabled

Desabilita uma face, ou seja, a face não vai processar nenhum evento até ser habilitada.

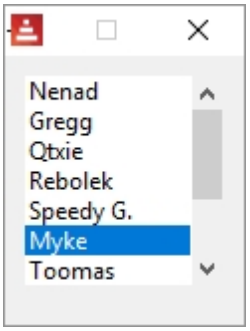
```
Red [needs: view]

view [
      button "I should quit, but I don't" disabled [quit]
      button "Quit" [quit]
]
```



VID DLS select

Determina a facet selecionada na face corrente. Usada geralmente para listas, para indicar qual item é pré-selecionado.

```
Red [needs: view]
view [
      tl: text-list 100x100 data  [
      "Nenad" "Gregg" "Qtxie" "Rebolek"
      "Speedy G." "Myke" "Toomas"
      "Alan" "Nick" "Peter" "Carl"
      ] select 6
      [print tl/selected]
]
```



VID DLS focus

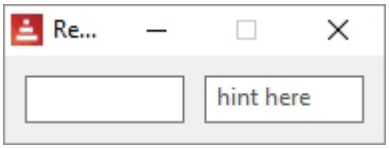
Dá o foco para a face corrente quando a janela é mostrada pela primeira vez. Somente uma face pode ter o foco. Se vários `focus` forem usados em diferentes faces, apenas a última pega o foco.

```
Red [needs: view]
view [
   field
   field
   field focus
   field
]
```

VID DLS hint

Fornece uma mensagem dentro das faces de `field` quando o campo está vazio. Tem a função de orientar o usuário. O texto desaparece quando um novo conteúdo é inserido por digitação do usuário ou mudando o atributo `face/text` faces.

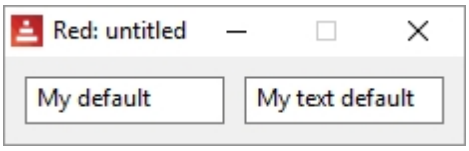
```
Red [needs: view]
view [
   field
   field hint "hint here"
]
```



VID DLS default

Define um valor default para a facet `data` nas faces `text` e `field` .

```
Red [needs: view]
view [
      a: field 100 default "My default"
      b: field 100 "My text default"
      do [
            print a/text
            print a/data      ; "data" was defined by "default" facet
            print b/text
            print b/data      ; this will give you an error, as "data"
was not defined yet
      ]
]
```




```
My default
My default
My text default
*** Script Error: My has no value
*** Where: print
*** Stack: view layout do-safe
```

VID DLS with

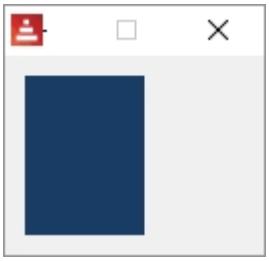
Suponha que você quer criar uma face cujas facets são avaliadas no momento da criação. Você não pode fazer computação nos "argumentos" de sua face, então você usa `with` .

Isto não funciona:

```
Red [needs: view]
a: 2
b: 3
view [
      base a * 30x40 b * 8.20.33
]
```

Isto funciona:

```
Red [needs: view]
a: 2
b: 3
view [
      base with [
            size: a * 30x40
            color: b * 8.20.33
      ]
]
```



VID DLS rate

`rate` é uma facet que tem um timer. Quando termina o tempo do timer um evento `on-time` é gerado. Note que se o argumento de `rate` é um `integer!`, significa "vezes por segundo", então um `rate` de 20 é mais rápido que um `rate` de 5. Você pode dar um argumento tipo `time!` para ajustar o `rate` para um tempo.

Este código faz o texto piscar:

```
Red [needs: view]

view [
      t: text "" rate 2
      on-time [either t/text = "" [t/text: "Blink"] [t/text: ""]]
]
```

Este código faz uma animação tosca onde uma base azul atravessa a janela:

```
Red [Needs: 'View]

view[
    size 150x150
    b: base 40x40 blue "I move" rate 20
        on-time [b/offset: b/offset + 1x1]
]
```

Rates mais lentos:

Para períodos superiores a 1 segundo, use um argumento tipo `time!` para o `rate`:

```
Red [Needs: view]

view[
    t: text "" rate 0:0:3
        on-time [either t/text = "" [t/text: "Blink" print now/time]
[t/text: "" print now/time]]
]
```

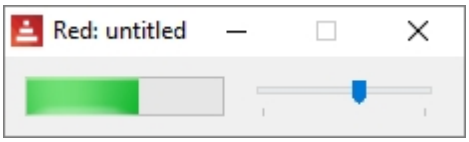
function! react

`react` é uma facet que vincula o comportamento de uma face aos dados de outra face.

O exemplo clássico:

```
Red [Needs: view]

view[
    progress 100x20 20% react [face/data: s/data]
      s: slider 100x20 20%
]
```

A face `progress bar` reage ao ajuste da face `slide` :



/link => vincula objetos usando uma relação reativa.

/unlink => remove uma relação existente.

/later => roda a reação na próxima mudança, ao invés de agora.

/with => <uso interno>

function! layout

`layout` é usado para criar views personalizadas sem mostrá-las. Você atribui o layout a uma palavra e, então, para mostrá-la, você usa `view` ou `unview`. Com `layout` voce pode deixar janelas GUI "prontas" para tarefas específicas.

O código abaixo mostra uma janela, e só mostra a outra quando você fecha a primeira.

```
Red [needs: view]

my-view: layout [button {click to "unview"} [unview]]

print "something"  ;do something else
print "biding my time" ;do something else

view/options my-view [offset: 30x100]
view/options my-view [offset: 400x100]
```

Obter o tamanho da tela:

```
>> print system/view/screens/1/size
1366x768
```

Criar uma view de tela inteira:

```
Red [needs: view]

view [size system/view/screens/1/size]
```

GUI - Rich text

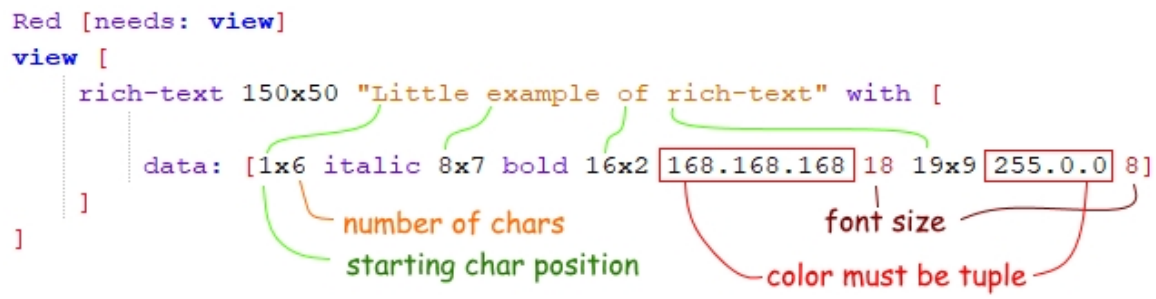
VID DLS rich-text

`rich-text` é um *face* que pode exibir texto em itálico, negrito, colorido e com tamanhos de fonte diferentes. Eu acredito que existem duas maneiras de passar os parâmetros para um rich text:

Primeiro método, usando with :

```
Red[needs: view]
view [
    rich-text 150x50 "Little example of rich-text" with [
        data: [1x6 italic 8x7 bold 16x2 168.168.168 18 19x9 255.0.0 8]
    ]
]
```

Explicando o primeiro método:




```
;número de caracteres
;posição do carater inicial
;tamanho fonte
;cor tem que ser um tuple
```

Se você não quiser usar *tuples* para cores, pode alterar a linha de *data* para::

```
data: reduce [1x6 'italic 8x7 'bold 16x2 gray 18 19x9 red 8]
```

Segundo método, usando function! rtd-layout

`rtd-layout` retorna uma *face* em rich text a partir de um código-fonte RTD. Eu acredito que é mais simples, e permite que você use rich-text a partir de fontes externas, mas você deve ler o capítulo de draw antes, e lembre-se de usar `compose/deep` no `view`. `compose` avalia as coisas entre parênteses, e é usado para "trazer" código de Red para o bloco do dialeto `view` , e é preciso ter um refinamento `/deep` porque os parênteses são aninhados dentro de colchetes..

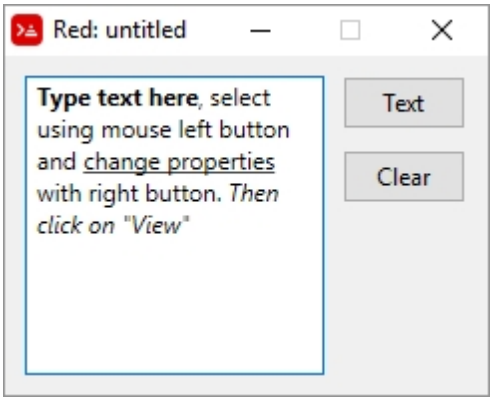
```
Red[needs: view]

myrtf: rtd-layout [i "This " /i b "uses " /b red font 14 "rtd-
layout" /font]

view compose/deep [
    rich-text 200x50 draw [text 0x0 (myrtf)]
    rich-text 200x50 draw [text 20x10 (myrtf)] ;the pair! locates the
text
]
```

Por favor, veja a página de exemplos de rich-text de Toomas Vooglaid. Com a permissão dele, coloquei abaixo alguns. O Toomas também tem um excelente gist com uma variedade de exemplos de Red em vários tópicos.

```
Red [
  Author: "Toomas Vooglaid"
]

view [rich-text 200x50 "Little example of rich-text" with [
    data: [1x6 italic 8x10 bold 16x2 168.168.168 19x9 255.0.0 18]]
]
rb: rtd-layout [i "And " /i b "another " /b red font 14 "example" /font]
view compose/deep [rich-text 200x50 draw [text 0x0 (rb)]]
```

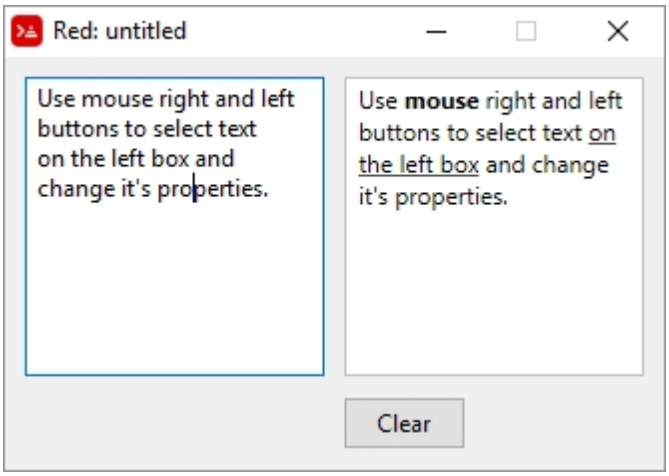

```
Red [
      Purpose: {Relatively simple rich-text demo}
      Help: {Enter text. Select some text, choose formatting from
```

```
contextual menu (alt-click).
            Press "View" to see formatting, "Text" to return to text
editing, "Clear" to clear                          formatting.}
]
count-nl: func [face /local text n x][
      n: 0 x: face/selected/x
      text: copy face/text
      while [all [
            text: find/tail text #"^/"
            x >= index? text
      ]][
            n: n + 1
      ] n
]
view compose [
      src: area wrap with [
            menu: ["Italic" italic "Bold" bold "Underline" underline]
      ]
      on-menu [
            nls: count-nl face
            append rt/data reduce [
                  as-pair face/selected/x - nls face/selected/y -
face/selected/x + 1 event/picked
            ]
      ]
      at 16x12 rt: rich-text hidden with [
            data: copy []
            size: src/size - 7x3
            line-spacing: 15
      ]
      below
      button "View" [
            if show-rt: face/text = "View" [rt/text: copy src/text]
            face/text: pick ["Text" "View"] rt/visible?: show-rt
      ]
      button "Clear" [clear rt/data]
]
```




```
Red [
      Purpose: {Relatively simple rich-text demo}
      Help: {Select some text in first box, choose formatting from
context-menu (alt-click).
            "Clear" clears formatting.}
]
```

```
count-nl: func [face /local text n x][
	n: 0 x: face/selected/x
	text: copy face/text
	while [all [
		text: find/tail text #"^/"
		x >= index? text
	]][
		n: n + 1
	] n
]
view compose [
	below src: area wrap with [
		menu: ["Italic" italic "Bold" bold "Underline" underline]
	]
	on-menu [
		nls: count-nl face
		append rt/data reduce [
			as-pair face/selected/x - nls face/selected/y -
face/selected/x + 1 event/picked
		]
	]
	on-key [rt/text: face/text rt/data: rt/data]
	return
	pnl: panel white with [
		size: src/size
		draw: compose [pen silver box 0x0 (size - 1)]
		pane: layout/only compose [
			at 7x3 rt: rich-text with [
				size: src/size - 10x6 data: copy []
			]
		]
	]
	button "Clear" [clear rt/data]
]
```



GUI - Criando views por programação

VID é o dialeto gráfico do Red. Todos os comandos da GUI (`base`, `across`, `style` etc) são código VID.

FACE TREE - é o object! de uma *view* gráfica. `view` e `show`. só conseguem mostrar este object!

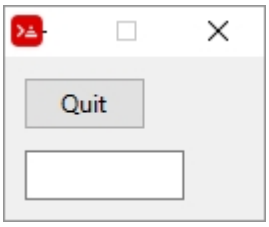
LAYOUT transforma qualquer bloco contendo código VID em uma *face tree*.

VIEW transforma (se necessário) um bloco de código VID em uma *face tree* e exibe como uma GUI.

SHOW mostra a *face tree*. Pode mostrar um `layout` (ou mesmo uma `view`), mas não pode exibir uma GUI fora de um bloco de código VID. Dentro de um bloco VID, ele atualiza um rosto, no entanto, em Red, ao contrário do Rebol, essa atualização é automática, a menos que você altere as configurações em `system/view/auto-sync?`, como explicado aqui.

Então, o argumento para `view` é apenas um bloco de código VID e você pode alterá-lo:

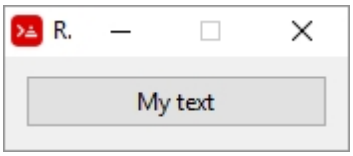
```
Red[needs: view]
board: []
append board [below button "Quit" [quit] field ]
view board
```



Usando variáveis externas como *facets* para *view*

A função interna `compose` computa o conteúdo entre parênteses e permite "passar" parâmetros para a `view`.

```
Red [needs: view]
txt: "My text"
size: 150
view compose [ button (txt) (size)]
```



Alterando uma GUI a partir da própria GUI

Se a GUI for criada a partir de um bloco com `compose` e, em seguida, renderizada por `view` , qualquer alteração nos valores no bloco será refletida imediatamente na GUI :

```
Red[needs: view]

board: compose [
      a: box blue 50x50
      button "Move blue box" [a/offset: (a/offset: a/offset + 5x0)]
]   ; every click increases position of blue box

view board
```

Mostrando e escondendo *faces*

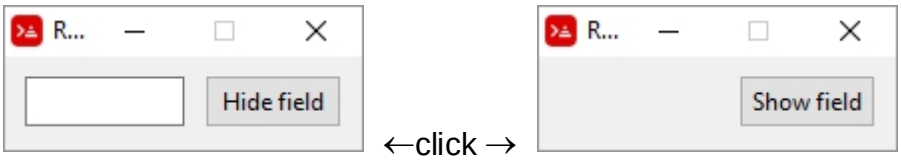
Faces tem o atributo `visible?` que pode ser alterado de `true` (default) para `false` para ocultar uma *face*. No script a seguir, clique no botão para ativar e desativar a visibilidade do `field` :

```
Red [needs: view]
view [
    f: field
    button "Hide field" [f/visible?: not f/visible?]
]
```

Um exemplo mais elegante (por Toomas Vooglaid):

```
Red[needs: view]

view [
   f: field
   button "Hide field" [
      face/text: pick [
         "Hide field" "Show field"
      ] f/visible?: not f/visible?
```

```
    ]
]
```


R...
Hide field
←click →
R...
Show field

Parse

Parse é um "dialeto" do Red (um DSL- domain specific language para ser preciso), isto é, uma mini linguagem embutitda dentro do Red. O interpretador Red que vem com o download já possui algumas destas linguagens: VID para criação de GUIs, DRAW para gráficos e PARSE.

Parse deve ser estudado como uma pequena linguagem de programação.

native! parse

`parse` pega cada elemento de um *input* e o submete a uma regra correspondente em um bloco de regras. Retorna `true` se todas as regras forem atendidas ou `false`, se alguma falhar (fracassar) em atender à respectiva regra.

O exemplo mais básico seria simplesmente checar se cada elemento do bloco de *input* é igual ao elemento correspondente no bloco de regras;

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; bloco de input
print parse a ["fox" "dog" "owl" "rat" "elk" "cat"]
```

```
true
```

Para deixar mais clara a descrição do `parse`, vou reescrever o exemplo acima com outra formatação:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; bloco de input

print parse a [   ;aqui começam as regras:
      "fox"  ; elemento 1 atende regra 1 => sucesso
      "dog"  ; elemento 2 atende regra 2 => sucesso
      "owl"  ; elemento 3 atende regra 3 => sucesso
      "rat"  ; elemento 4 atende regra 4 => sucesso
      "elk"  ; elemento 5 atende regra 5 => sucesso
      "cat"  ; elemento 6 atende regra 6 => sucesso
]
; como todos os elementos atendem às suas regras, o resultado é "true"
```

```
true
```

A correspondência pode ser com datatypes:

```
Red[]
```

```
a:  [33 18.2 #"c" "rat"]      ; input block

print parse a [
      integer!
      float!
      char!
      string!
]
```

```
true
```

Código normal do Red pode ser inserido dentro do bloco de regras usando parêntesis:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk"]       ; input block

print parse a [
      "fox"
      "dog"
      "owl"
      (loop 3 [print "just regular code here!"])
      "rat"
      "elk"
]
```

```
just regular code here!
just regular code here!
just regular code here!
true
```

As regras aceitam o operador lógico "ou" representado por "|":

```
Red[]

a: ["fox" "rat" "elk"]
b: ["fox" "owl" "elk"]

print parse a [
      "fox"
      ["rat" | "owl"]         ;note os colchetes
      "elk"
]
print parse b [
      "fox"
      ["rat" | "owl"| "cat" | "whatever"]
      "elk"
]
```

```
true
true
```

As regras podem ser repetidas adicionando o número de repetições (ou intervalo) antes delas:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
    4 skip  ; see command skip at Parse/Matching
      "elk"
      "cat"
]
```

```
true
```

```
Red[]

a:  ["rat" "rat" "rat" "rat" "elk" "cat"]

print parse a [
    4 "rat"
      "elk"
      "cat"
]
```

```
true
```

Ou intervalo:

```
Red[]

a:  ["rat" "rat" "elk" "cat"]

print parse a [
    0 4 "rat"     ; will return success if there is from zero up to four
"rat"
      "elk"
      "cat"
]
```

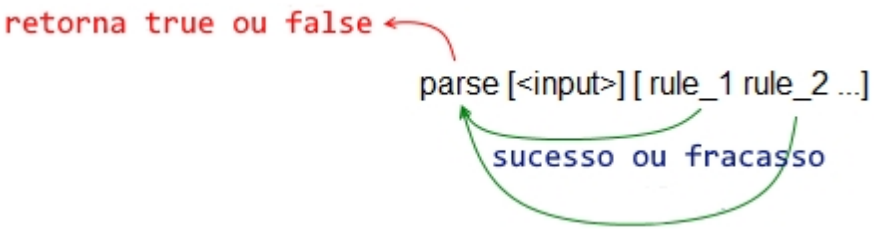
```
true
```

Refinamentos do Parse:

/case =>
/part =>
/trace =>

Esclarecimento importante:

O comando `parse` retorna `true` ou `false`, mas o *matching* (tentativa de ver se a entrada atende à regra) envia ao `parse` sucesso ou fracasso (**success** ou **failure)**. É importante entender isso para não fazer confusão.


retorna true ou false
parse [<input>] [rule_1 rule_2 ...]
sucesso ou fracasso

Debugging Parse

O dialeto Parse é poderoso, mas também é difícil de visualizar e notoriamente difícil de depurar. Antes de prosseguir para os recursos mais avançados da análise, sugiro que você aprenda como depurar seu código. Há duas maneiras que eu conheço: usar a função `parse-trace` e imprimir informações ao longo da computação.

function! parse-trace

Faz o parse da entrada, mas também imprime (rastreia) todas as etapas do processo.

```
Red[]
a:  ["fox" "owl" "rat"]
print parse-trace a ["fox" "owl" "rat"]
```

```
-->
  match: ["fox" "owl" "rat"]
  input: ["fox" "owl" "rat"]
  ==> matched
  match: ["owl" "rat"]
  input: ["owl" "rat"]
  ==> matched
  match: ["rat"]
  input: ["rat"]
  ==> matched
return: true
true
```

```
Red[]
a:  ["fox" "owl" "rat"]
print parse-trace a [["fox" | "cow"] "owl" "rat"]
```

```
-->
  match: [["fox" | "cow"] "owl" "rat"]
  input: ["fox" "owl" "rat"]
  -->
    match: ["fox" | "cow"]
    input: ["fox" "owl" "rat"]
    ==> matched
    match: [| "cow"]
    input: ["owl" "rat"]
  <--
  match: ["owl" "rat"]
  input: ["owl" "rat"]
  ==> matched
  match: ["rat"]
```

```
   input: ["rat"]
   ==> matched
return: true
true
```

usar *print*:

Coloque instruções `print` em locais estratégicos para informar o status da computação:

```
Red[]
a:  ["fox" "owl" "rat"]
print parse a ["fox" (print "reached fox")
               "owl" (print "reached owl")
               "rat" (print "reached the end")
              ]
```

```
reached fox
reached owl
reached the end
true
```

Parse - Matching:

PARSE skip

Pula um elemento:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"] ; input block

print parse a [   ;here the rules begin:
      "fox"
      skip   ; simplesmente pula este elemente
      "owl"
      "rat"
      "elk"
      "cat"
]
```

```
true
```

Outro exemplo, notando que string são séries de caracteres, e são um tipo comum de input para o parse:

```
Red []
a: "XYZhello"
print parse a [skip skip skip "hello"]
```

```
true
```

Ou, mais elegantemente (veja repetição):

```
Red []
a: "XYZhello"
print parse a [3 skip "hello"]
```

```
true
```

PARSE to PARSE thru

Pula elementos até achar uma correspondência. `thru` posiciona o *input* após a correspondência, `to` posiciona antes desta.

```
Red[]
a: "big black cat"
```

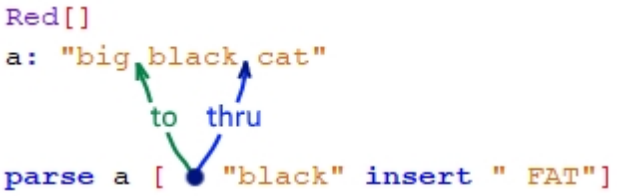
```
parse a [ to "black" insert "FAT "]
print a
```

```
big FAT black cat
```

```
Red[]
a: "big black cat"
parse a [ thru "black" insert " FAT"]
print a
```

```
big black FAT cat
```

Assim:



Exemplo de `to`:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

print parse a [    ;here the rules begin:
      "fox"        ; rule 1 matches element 1 => true
      to "elk"     ; skips all elements until...
                   ; ...it finds a match, but..
      "elk"        ; ... it also checks if the match fits the rule
      "cat"        ; rules for the elements...
      "bat"        ; ... following the match
]
```

```
true
```

Exemplo de `thru`:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

print parse a [    ;here the rules begin:
      "fox"        ; rule 1 matches element 1 => true
      thru "elk"   ; skips all elements until...
                   ; ...it finds a match
      "cat"        ; rules for the elements...
      "bat"        ; ... following the match
]
```

```
true
```

PARSE end

Retorna `true` se todos os elementos do input foram checados pelo parse.

```
Red[]

a:  [33 18.2 #"c" "rat"] ; input block

print parse a [
      integer!
      float!
      char!
      string!
      end
]
```

```
true
```

PARSE ahead

Verifica se o próximo elemento atende à regra:

```
Red[]

a:  ["fox" "dog" "owl" "rat"] ; input block

print parse a [
      "fox"
      "dog"
      ahead "owl" ;verifica se o próximo item atende => ok
      "owl"
      "rat"
]
```

```
true
```

PARSE none

Sempre retorna sucesso.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]  ; input block

print parse a [
      "fox"
      "dog"
      none  ; não faz nada, mas ações podem ser inseridas aqui
      "owl"
      "rat"
]
```

```
true
```

PARSE opt

Se encontra uma correspondência, retorna sucesso e o parse segue para o próximo input. Se o input não atende à regra, simplesmente ignora o input atual e segue testando a correspondência para o próximo input.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]  ; input block

print parse a [
      "fox"  ; ok
      "dog"  ; ok
      opt "owl"  ; item 3 atende regra 3 =>ok
      "rat"  ; item 4 atende regra 4 => ok
]
print parse a [
      "fox"  ; ok
      "dog"  ; ok
      opt "BAT"   ; não tem "BAT" no input, então o parse pula esta
regra...
      "owl"  ; ... e testa o input nesta próxima regra =>ok
      "rat"  ; ok
]
```

```
true
true
```

Outro exemplo:

```
 Red []
 a:  ["Mrs" "Robinson"]
 print parse a [opt "Mrs" "Robinson"] ;TRUE

 a:  ["Robinson"]
 print parse a [opt "Mrs" "Robinson"] ;TRUE, the "Mrs" is OPTional

 a:  ["Miss" "Robinson"]
 print parse a [opt "Mrs" "Robinson"] ; FALSE, "Mrs" is optional, but
"Miss" is wrong!
```

Mais um exemplo:

```
a: [ "elk" "cat" "owl"]

parse a [ opt [ "fig" ] "elk" "cat" "owl" ]  ; never or at least once
```

```
true
```

```
parse a [ opt [ "elk" "cat" ] "owl" ]      ; never or at least once
```

```
true
```

```
parse a [ opt [ "elk" "owl" ]  "cat" ]     ; never or at least once
```

```
false *
```

* Se a entrada não corresponder à regra do `opt` , a análise ignorará essa regra e verificará a mesma entrada pela regra a seguir..

Ainda mais um exemplo de **opt** :

```
hd: "mountaintrack"       ; string
parse hd [ opt "mountain" "track"]   ; == true
parse hd [ opt "mountain" "rights"]   ; == false
```

PARSE not

A definição oficial da regra `not` é de que "inverte o resultado da sub-regra". Para mim, parece ser uma regra que exclui uma possível correspondência da próxima regra. Note que o input não é "consumido" (não "anda").

```
Red[]

a:  ["fox" "dog" "owl" "rat"]

print parse a [
      "fox"
      "dog"
      not "owl" ;não consome input
      skip ;qualquer coisa aqui menos "owl" - fails!
      "rat"
]
print parse a [
      "fox"
      "dog"
      not "COW" ; não consome input
      skip ;qualquer coisa aqui, menos "COW" - success!
      "rat"
]
```

```
false
true
```

PARSE quote

Verifica a correspondência do argumento exatamente como é (literalmente), exceto para `paren!` (coisas entre parêntesis).

Isto dá um erro:

```
>> parse [x] [x]
*** Script Error: PARSE - invalid rule or usage of rule: x
*** Where: parse
*** Stack:
```

Mas isso funciona:

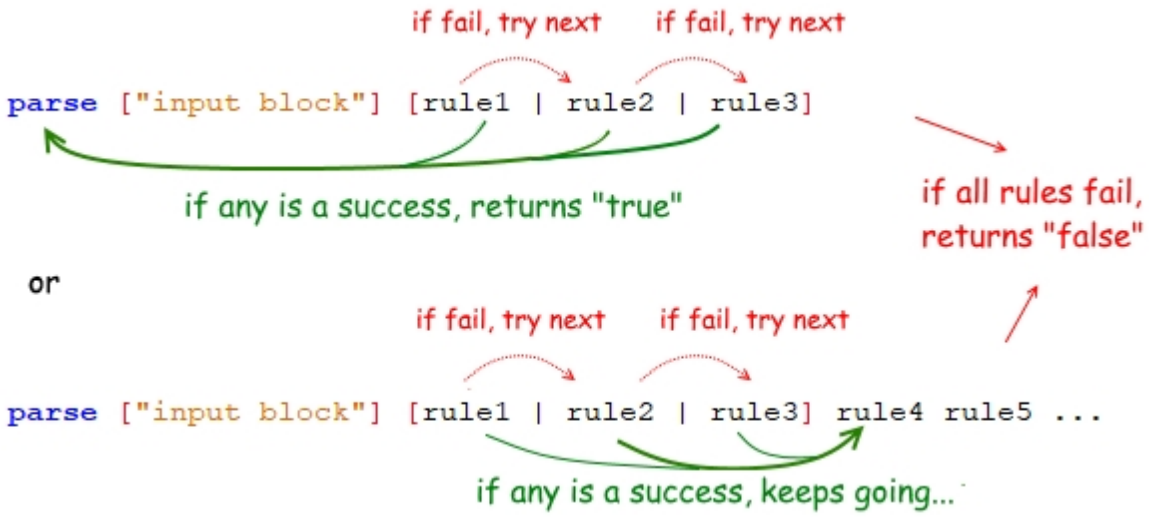
```
>> parse [x] [quote x]
```

```
== true

>> parse ['x] [quote 'x]
== true

>> parse [[x]] [quote [x]]
== true
```

Parse - Ordered Choices

Regras aceitam um operador "escolha ordenada", representado por "|"

Se um bloco de regras separado por "|" é encontrado pelo parse, ele tentará cada regra, da esquerda para a direita até encontrar uma correspondência, retornando sucesso e indo para a próxima regra após este bloco. Se nenhum deles resultar em uma correspondência, é claro, ele falhará e a análise será interrompida retornando `false`.



Isso é semelhante a um operador lógico "ou" , mas a ordem é importante.

Exemplo1:

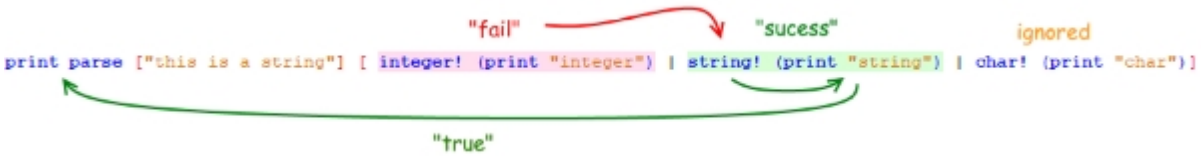
```
Red[]

a: ["fox" "rat" "elk"]
b: ["fox" "owl" "elk"]

print parse a [
      "fox"
      ["rat" | "owl"]         ;notice enclosing brackets
      "elk"
]
print parse b [
      "fox"
      ["rat" | "owl"| "cat" | "whatever"]
      "elk"
]
```

```
true
true
```

Exemplo 2:

```
Red[]
print parse ["this is a string"] [ integer! (print "integer") | string!
(print "string") | char! (print "char")]
```

```
string
true
```



Exemplo3:

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [integer! (print "I") | string! (print "S") | time! (print
 "T")]
```

```
S
false
```

Repetindo o script com `parse-trace` em vez de `print parse` (destaques de cor, novas linhas, fonte em negrito e comentários adicionados por edição):

```
 -->
   match: [integer! (print "I") | string! (print "S") | time
   input: ["string" 3 #"A"]
   ==> not matched

   match: [string! (print "S") | time! (print "T")]
   input: ["string" 3 #"A"]
   ==> matched
                              ;keeps going to execute commands in
parenthesis
   match: [(print "S") | time! (print "T")]
   input: [3 #"A"]
S
   match: [| time! (print "T")]
   input: [3 #"A"]
return: false                 ;too much input and not enough rules ->
false
```

Para obter `true` , podemos adicionar mais regras para a escolha ordenada bem sucedida ...

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [integer! (print "I") | string! (print "S") integer! char!
| integer! (print "T")]
```

```
S
true
```

... ou colocar as escolhas ordenadas entre parênteses e adicionar regras ao bloco de regras principal:

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [[integer! (print "I") | string! (print "S") | time!
(print "T")] integer! char!]
```

```
S
true
```

Repetição e *matching loops*:

Palavras-chave: some, any, opt, while.

Palavra-chave ou valor	Descrição
3 <regra>	repita a regra 3 vezes
1 3 <regra>	repita a regra de 1 a 3 vezes
0 3 <regra>	repita a regra de 0 a 3 vezes
some	repita sua (s) regra (s) enquanto (e se) obtiver um `true` (match) da regra. Retorna `false` se não obtiver pelo menos uma correspondência (faz com que a análise seja `false`).
any	repita sua (s) regra (s) até obter um `false` (sem correspondência) da regra. Sempre retorna `true` para a expressão de análise.
while	veja o texto abaixo.

Número de Repetição Conhecido - Exemplos

```
>> parse "fogfogfog" [3 "fog"]; determined exactly
== true
```

```
>> parse "fogfogfog" [0 5 "fog"]; determined by range
== true
```

Exemplos de script para número exato de repetições:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
    4 skip  ; see command skip at Parse/Matching
```

```
      "elk"
      "cat"
]
```

```
true
```

```
Red[]

a:  ["rat" "rat" "rat" "rat" "elk" "cat"]

print parse a [
    4 "rat"
      "elk"
      "cat"
]
```

```
true
```

Ou um intervalo:

```
Red[]

a:  ["rat" "rat" "elk" "cat"]

print parse a [
    0 4 "rat"     ; will return success if there is from zero up to four
"rat"
      "elk"
      "cat"
]
```

```
true
```

Matching Loops:

PARSE some, PARSE any

Novamente:

`some` - repita a (s) regra (s) enquanto (e se) obtiver um `true` (correspondência) da regra. Retorna `false` se não obtiver pelo menos uma correspondência (faz com que a análise seja `false`).

`any` - repete a (s) regra (s) até obter um `false` (sem correspondência) da regra. Sempre retorna `true` para a expressão de análise

Ambos retornam sucesso enquanto encontrarem correspondências na entrada, a diferença é que `some` requer pelo menos uma ocorrência da entrada (correspondência), enquanto `any` retornará sucesso mesmo sem correspondência.

```
Red[]
```

```
a:  ["fox" "dog" "fox" "dog" "fox" "dog" "elk" "cat"]

print parse a [
      some ["fox" "dog"]
      "elk"
      "cat"
]

print parse a [
      any ["fox" "dog"]
      "elk"
      "cat"
]
```

```
true
true
```

```
Red[]

a:  ["elk" "cat"]

print parse a [
      some ["fox" "dog"]
      "elk"
      "cat"
]

print parse a [
      any ["fox" "dog"]
      "elk"
      "cat"
]
```

```
false
true
```

Exemplo que mostra o comportamento "loop" mais claramente:

```
Red []
txt: {In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?}

print parse txt [some [thru "blue" (print "I found blue!")] to end]
```

```
I found blue!
I found blue!
I found blue!
I found blue!
true
>>
```

Explicando o exemplo:

```
[some
      [thru "blue" (print "I found blue!")] ; essa regra vai se repetir
enquanto encontrar um match
```

```
to end]
```

- primeiro *loop*:

```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```

-> achou um *match*, então repete `[thru "blue" (print "I found blue!")]`

- segundo *loop*:

```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```

-> achou um *match*, então repete `[thru "blue" (print "I found blue!")]`

- terceiro *loop*:

```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```

-> achou um *match*, então repete `[thru "blue" (print "I found blue!")]`

- quarto loop:

```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```

-> achou um *match*, então repete `[thru "blue" (print "I found blue!")]`

-> SEM *match*, então sai do loop de `some` e vai para a próxima regra: `to end`, que é um *match*, porque simplesmente vai para o fim.

Uma vez que todas as regras acharam um *match* (`some` achou mais que um), o parse retorna `true`.

PARSE while

Definitivamento não é para principiantes, como é explicado a seguir por by @9214 do gitter.
Como a explicação é complexa e minha tradução deficiente, acredito que a leitura do original em inglês seja necessária para um completo entendimento. Desculpe.

"

```
>> parse x: [a 1 a 1][while [ahead ['a change quote 1 2] | 'a quote 2]]
== true

>> x
== [a 2 a 2]

>> parse x: [a 1 a 1][any [ahead ['a change quote 1 2] | 'a quote 2]]
== false

>> x
== [a 2 a 1]
```

A principal diferença entre `while` e `any` é que o primeiro continua executando o parse mesmo se o index não avança após uma correspondência bem-sucedida, enquanto o segundo falha tão logo o index permaneça na mesma posição, mesmo que a

correspondência tenha sido bem-sucedida.

Foi por isso que eu usei o `ahead` - é uma regra que faz a correspondência "à frente", mas mantém o index onde está. No exemplo acima, `ahead ['a change quote 1 2]` vai fazer a correspondência com sucesso, e 1 depois de "a" vai ser mudado para 2, *mas a posição do input não vai avançar, porque o* `ahead` *olha para frente, enquando fica no mesmo lugar*. Os resultados são:

- Com `while`, primeiro `ahead ...` muda 1 para 2 sem avançar o input, mas uma vez que `while` não se importa com isso, ele vai para a próxima iteração, na qual uma regra de alto nível vai falhar e dar um passo atrás (uma opção após `|`) para `'a quote 2`, que vai dar uma corrêspondência bem-sucedida (porque nós acabamos de mudar `a 1` para `a 2` e avançar o input, assim nos levanod ao marcador `end` e fazendo o parsing bem sucedido de toda a série.

- Entretanto, com `any`, primeiro `ahead ...` muda `1` para `2` , não avança o input, e o `any`, por ser exigente sobre o avanço do input, resulta em falha sem chegar à segunda iteração.

O uso do `while` é complicado. Na minha experiência, eu o usei para parsing sensível a contexto (isto é, primeiro você olha para a frente e para trás, determinando o contexto de um token, e só então decide o que fazer; "olhar para frente e para trás requer a correspondência de várias regras enquando você não sai do lugar, na posição corrente*) e também em situações onde o input precisa ser modificado durante o parsing (exemplo acima), ou se o parsing depende de algum estado externo. Também se mostrou útil para deep-first traversal of tree-like structures (n.t. não sei traduzi isso) .- a situação é a mesma, você está mexendo com node, fazendo correspondências com regras, mas a posição não deve avançar se você fizer uma correspondência, senão você perde a referência do node corrente.

Portanto, `while` não é amigável para principiantes e você não precisa se preocupar com ele se você está começando. Ele é útil sem situações mais avançadas, onde você precisa de um controle seguro sobre o parsing."

Parse - Guardando o input:

PARSE set e PARSE copy

Ambos pegam o input da próxima regra de parse, se for bem-sucedida.
`set` atribui o input a uma variável e `copy`, atribui uma cópia deste a uma variável.

```
Red[]

a: ["fox" "rat" "elk"]

 parse a [
      "fox"
      set b       ;pronto para atribuir se a próxima regra for bem-
sucedida. Poderia ter usado copy.
      "rat"       ;sucesso, então "rat" => b
      "elk"
]
print b

rat
```

PARSE collect e PARSE keep

Se você tem um bloco de "collect" dentro do seu bloco de regras, o `parse` não vai mais retornar um `true` ou `false`, ao invés disso, vai retornar um bloco com todos os sucessos que foram precedidos da palavra (comando) `keep` .

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block

 print parse a [
      collect[
      keep "fox"  ; sucesso, será mantida
      "dog"
      "owl"
      keep "rat"  ; sucesso, será mantida
      keep "cow"  ; FALHOU! NÃO será mantida
      "cat"
            ]
]
```

```
fox rat
```

PARSE collect set

`parse` vai retornar `true` ou `false`, e inserir todos os sucessos precedidos da palavra `keep` em um novo bloco.

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ;

 print parse a [
      collect set b [                           ; cria b para guardar os
keeps
      keep "fox"  ; sucesso, será mantida
      "dog"
      "owl"
      keep "rat"  ; sucesso, será mantida
      keep "cow"  ; FALHOU! NÃO será mantida
      "cat"
            ]
]

print b
```

```
false
fox rat
```

PARSE collect into

`parse` vai retornar `true` ou `false`, e inserir todos os sucessos precedidos da palavra `keep` em um bloco existente. Parece que faz um append no bloco.

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block
b:""                                            ; must create block
first

 print parse a [
      collect into b [
      keep "fox"  ; sucesso, será mantida
      "dog"
      "owl"
      keep "rat"  ; sucesso, será mantida
      keep "cow"  ; FALHOU! NÃO será mantida
      "cat"
            ]
]

print b
```

```
false
foxrat
```

Coletando o input usando a sintaxe de set-word (dois pontos)

Durante o processamento do `parse`, você pode atribuir a parte restante de um input a uma palavra (variável):

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
      "fox"
      "dog"
      b:
]
probe b
```

```
false
["owl" "rat" "elk" "cat"]
```

Parse - Modifying input:

PARSE insert

Insere um valor no bloco de input na posição corrente.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
      "fox"
      "dog"
      insert 33
      "owl"
      "rat"
]
print a
```

```
true
fox dog 33 owl rat
```

PARSE remove

Remove do bloco de input um valor que tenha uma correspondência bem-sucedida.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
      "fox"
      remove
      "dog"
      remove
      "owl"
      "rat"
]
print a
```

```
true
fox rat
```

PARSE change

Altera uma correspondência bem-sucedida:

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
      "fox"
      "dog"
      change "owl" "COW"
      "owl"
      "rat"
]
print a
```

```
false
fox dog COW rat
```

Controle do fluxo

Parse - Controle de fluxo:

PARSE if

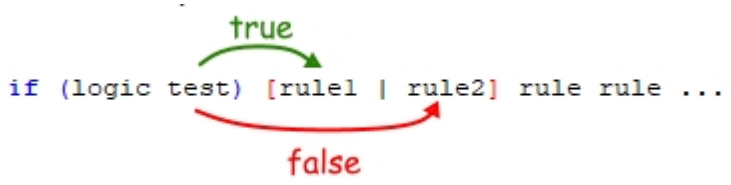
`if` esta o resultado de uma expressão lógica entre parênteses. Geralmente é seguido por `regra1 | regra 2` .

Se não houver uma escolha ordenada (`rule1 | rule 2`) depois do `if`, e o resultado da expressão for `false` ou `none` o parsing é encerrado, retornando `false`.

```
Red[]
block: [6 3 7]
print parse block [integer! integer! if (1 = 1) integer!] ;(1 = 1)
verdadeiro, então continua
print parse block [integer! integer! if (1 = 2) integer!] ;(1 = 2)
falso, então encerra, retornando false
```

```
true
false
```

Com escolhas ordenadas: Se o resultado dessa expressão lógica for `true` , o loop de parse usará `rule1` ; se for `false` ou `none` , usará `rule2` para a próxima tentativa de *match*.




```
Red[]
block: [6 3 7]
print parse block [integer! integer! if (1 = 1) [integer! | string!]] ;
7 is an integer! -> true
print parse block [integer! integer! if (1 = 2) [integer! | string!]] ;
7 is not a string! false
```

```
true
false
```

Outro exemplo simples:

```
Red[]
```

```
block: [1 2]
print parse block [set value integer! if (value = 1) to end]
block: [2 2]
print parse block [set value integer! if (value = 1) to end]
```

```
true
false
```

PARSE then

Independentemente da falha ou sucesso do que segue, pule a próxima regra alternativa. Ou seja, quando um `then` for encontrado, a próxima regra alternativa será desativada.

Não achei bons exemplos para isso.

PARSE into

Muda o input para uma série (string or block) e faz o parse com a regra.
Não achei bons exemplos para isso.

PARSE fail

Força a regra corrent a falhar e faz um backtrack (volta para o input anterior).
Não achei bons exemplos para isso. Acho que é relaciodado com matching loops (any, some and while) apenas.

PARSE break

Sai de um matching loop, retornando success.
Não achei bons exemplos para isso. Acho que é relaciodado com matching loops (any, some and while) apenas, especificamente para oferecer uma maneira de evitar loops infinitos.

PARSE reject

Sai de um matching loop, retornando failure.
Não achei bons exemplos para isso. Acho que é relaciodado com matching loops (any, some and while) apenas.

Uso do parse - Validando inputs

Validando entradas alfanuméricas:

Antes de prosseguir, devo avisá-lo que o datatyping do Red pode causar alguns problemas à programação. Por exemplo, um número de um único dígito em Red pode ser um `integer!`, um `string!`, um `char!` , ou qualquer outra coisa. Portanto, se você tiver alguns erros inexplicáveis em seu script, verifique se os datatypes são compatíveis.

Esse é um script que solicita ao usuário que digite 4 números de um dígito e verifica se a entrada está OK até que a entrada seja "q":

```
Red []
entry: ""
while [entry <> "q"] [
      entry: ask "Enter four digits in the 1-8 range: "
      either (parse entry [some ["1" | "2" | "3" | "4" | "5" | "6" | "7"
| "8"]]) and ((length? entry) = 4) [
            print "OK"]
            [
            print "Not OK!"
            ]
]
```

Isso funciona, mas `["1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"]` pode ser substituido por `charset ["12345678"]`:

```
Red []
entry: ""
validchar: charset ["12345678"]
while [entry <> "q"] [
      entry: ask "Enter four digits in the 1-8 range: "
      either (parse entry [some validchar]) and ((length? entry) = 4) [
            print "OK"]
            [
            print "Not OK!"
            ]
]
```

Como o parse verifica caracter por caracter, `charset ["12345678"]` também pode ser escrito como `charset [#"1" - #"8"]` ! Red entende que essa é uma seqüência de caracteres. Assim, seu programa pode aceitar qualquer caractere ASCII numérico e minúsculo usando `charset [#"0" - #"9" #"a" - #"z"]`.

Simples validador de número de telefone (do manual da Rebol / Core) - Regras referindo-se a outras regras:

```
Red []
digits: charset "0123456789"
area-code: ["(" 3 digits ")"]
phone-num: [3 digits "-" 4 digits]

print parse "(707)467-8000" [[area-code | none] phone-num]
```

```
true
```

Simples validador de e-mail (do blog do Red):

```
Red []

digit:   charset "0123456789"
letters: charset [#"a" - #"z" #"A" - #"Z"]
special: charset "-"
chars:   union union letters special digit
word:    [some chars]
host:    [word]
domain:  [word some [dot word]]
email:   [host "@" domain]

print parse "john@doe.com" email
print parse "n00b@lost.island.org" email
print parse "h4x0r-l33t@domain.net" email
```

```
true
true
true
```

Validando expressões matemáticas escritas como texto (do manual do Rebol/Core):

Observe que este exemplo usa regras recursivas (uma regra que se refere a si mesma).

```
Red []

expr:    [term ["+" | "-"] expr | term]
term:    [factor ["*" | "/"] term | factor]
factor:  [primary "**" factor | primary]
primary: [some digit | "(" expr ")"]
digit:   charset "0123456789"

print parse "1+2*(3-2)/4" expr        ; vai retornar true
print parse "1-(3/)+2" expr           ; vai retornar false
```

```
true
false
```

Uso do parse - Extraindo dados

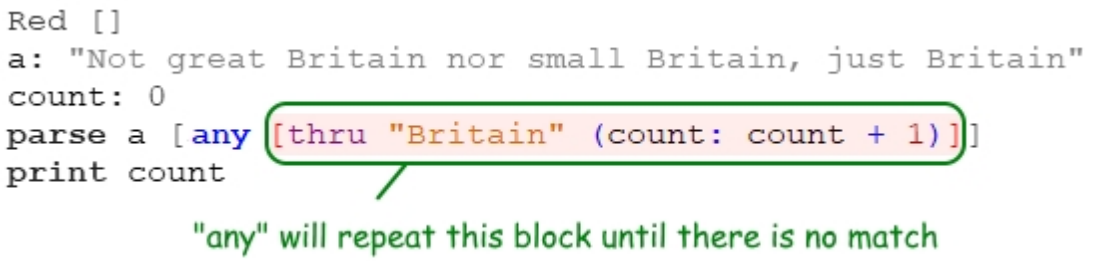
Contando palavras no texto:

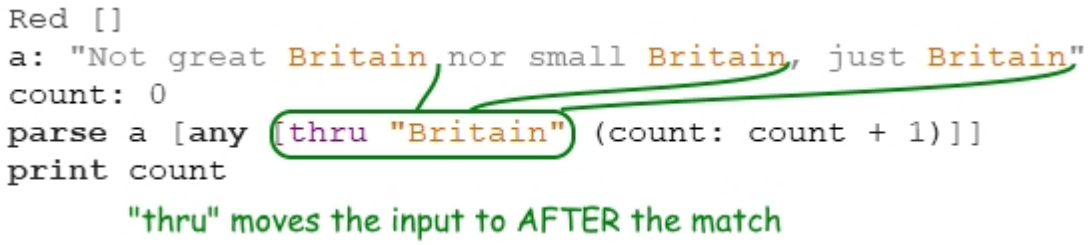
```
Red []
a: "Not great Britain nor small Britain, just Britain"
count: 0
parse a [any [thru "Britain" (count: count + 1)]]
print count
```

```
3
```

Explicando o programa:

Enquanto `thru "Britain"` encontrar um "Britain", `any` vai repetir a regra





Observe que, se você tivesse utilizado `to` em vez de `thru` , o *input* seria movido para ANTES do *match*, criando um loop infinito , já que o parse ficaria repetindo sempre o *match* com "Britain".

Extraindo uma parte de um texto:

Para extrair a parte restante de um texto a partir de um determinado ponto, você pode usar `word:`, como explicado no capítulo Guardando o Input . Para extrair texto entre dois *matches* do parse, você pode usar `copy` :

```
Red []
txt: "They are one person, they are two together"
parse txt [thru "person, " copy b to " two"]
print b
```

```
they are
```

Extraindo dados da Internet:

Este é um exemplo muito básico. Eu criei uma página html em helpin.red: http://helpin.red/samples/samplehtml1.html . O html é muito simples e você pode vê-lo digitando `print read http://helpin.red/samples/samplehtml1.html` no console. Como conhecemos o html, podemos extrair algumas informações com o código abaixo:

```
Red []
txt: read http://helpin.red/samples/samplehtml1.html
parse txt [
      thru "today"
      2 thru ">"
      copy weather1 to "<"
      thru "tomorrow"
      2 thru ">"
      copy weather2 to "<"
      thru "week"
      2 thru ">"
      copy weather3 to "<"
]
print {Acording to helpin.red website weather will be: }
print [] ; just adding an empty line
print ["Today:     "  weather1]
print ["Tomorrow:  "  weather2]
print ["Next week: "  #"^(tab)"  weather3] ; just showing the use of tab
```

```
Acording to helpin.red website weather will be:

Today:      sunny
Tomorrow:   horrible
Next week:        really really horrible
```

Mostrarei como o parse funciona para extrair o tempo de "today" para a variável "weather1":

```
thru "today" ; pula todo o texto até achar o texto "today".
```

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
```

```
    <tr>
      <td style="color: black;">weather today:</td>
      <td style="color: black;">sunny</td>
    </tr>
    <tr>
```

2 thru ">" ; isso faz pular o texto até (depois do) caracter ">". faz isso duas vezes!

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
    <tr>
      <td style="color: black;">weather today:</td>        ; 1
      <td style="color: black;">">sunny</td>               ; 2
    </tr>
    <tr>
```

copy weather1 to "<" ; isso copia para "weather1" tudo que encontra até (antes de) um "<".

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
    <tr>
      <td style="color: black;">weather today:</td>
      <td style="color: black;">sunny</td>                 ; ==>
weather1
    </tr>
    <tr>
```

Uso do parse - Manipulando texto

Inserindo palavras no texto:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [any [to "Britain" insert "blue " skip]]
print a

Not great blue Britain nor small blue Britain, just blue Britain
```

Observe que `skip` foi adicionado à regra para evitar um loop infinito: `to` leva o input para antes do *match*, portanto, "Britain" seria correspondido ininterruptamente se não usássemos o `skip`.

Removendo palavras do texto:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [ any [to remove "Britain"]] ;seems to work the same as [to
"Britain" remove "Britain"]
print a

Not great  nor small , just
```

Explicando o código:

Primeiro:

```
Red []
a: "Not great Britain nor small Britain, just Britain"

parse a [ any [to remove "Britain"]]
print a
```

"any" repeats the rule until no match is found.

; "any" repete a regra até que não encontre mais um *match*

Então:

```
Red []
a: "Not great Britain nor small Britain, just Britain"

parse a [ any [to remove "Britain"] ]
print a
```

"to 'Britain" takes the input to BEFORE the match ('Britain') and "remove" removes it.

; "to 'Britain" leva o input até ANTES do *match* ('Britain') e remove esse *match*

Mudando palavras do texto:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [ any [to "Britain" change "Britain" "Australia"]] ;[to change
"Britain" "Australia"] também funciona!
print a
```

```
Not great Australia nor small Australia, just Australia
```

Links para páginas que podem te ajudar a entender parse:

Links específicos do Red:

http://www.red-by-example.org/parse.html - talvez a melhor fonte de informação disponível.

http://www.red-lang.org/2013/11/041-introducing-parse.html

http://www.michaelsydenham.com/reds-parse-dialect/

https://github.com/red/red/issues/3478 - não exatamente o que você espera, mas informativo de qualquer forma. Discute problemas do parse.

Os links a seguir se referem ao parse em Rebol :

http://video.respectech.com - com editor interativo.

http://www.rebol.com/docs/core23/rebolcore-15.html

http://www.codeconscious.com/rebol/parse-tutorial.html

http://www.codeconscious.com/rebol/r2-to-r3-parse.html

http://www.rebol.com/r3/docs/concepts/parsing-summary.html - muito informativo.

http://www.rebol.com/r3/docs/functions/parse.html

http://blog.hostilefork.com/why-rebol-red-parse-cool/

https://en.wikibooks.org/wiki/Rebol_Programming/Language_Features/Parse/Parse_expressions

http://rebol2.blogspot.com/2012/05/text-extraction-with-parse.html

https://github.com/revault/rebol-wiki/wiki/Parse-Project

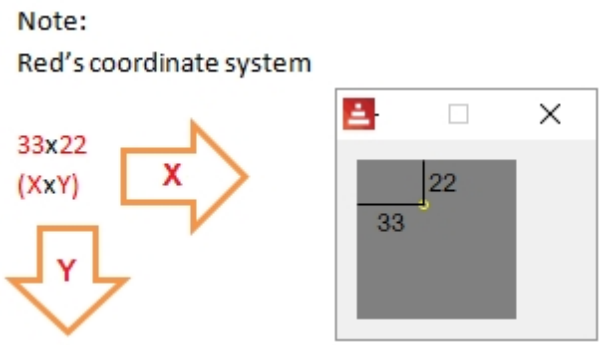
http://www.colellachiara.com/soft/Misc/parse-rep.html - propostas de melhoria no parse.

Draw

Draw é usado para criar gráficos 2D. Assim como o Parse e Vid, Draw é uma DSL, ou seja, um dialeto do Red, uma linguagem dentro da linguagem.

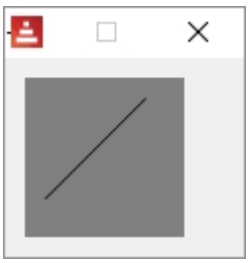
Para usar o `draw`, você tem que usar também a VID, então todo script que usa `draw` tem que ter um bloco `view` , e dentro deste bloco `view` , é preciso ter uma face `base` para desenhar. Os próximos examplos mostram todas os elementos básicos de draw.

Lembrando:

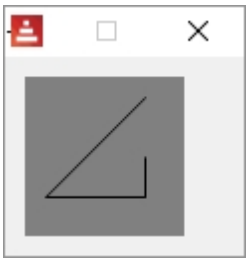


DRAW line

```
Red [needs: view]
view [
	base draw [line 60x10 10x60]
]
```



```
Red [needs: view]
view [
	base draw [line 60x10 10x60 60x60 60x40]
]
```



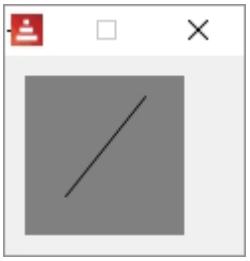
A importância de native! compose para DRAW

Suponha que você queira realizar avaliações nos argumentos do DRAW, como:

```
Red [needs: view]
view [
      base draw [line 60x10 (2 * 10x30)]
]
```

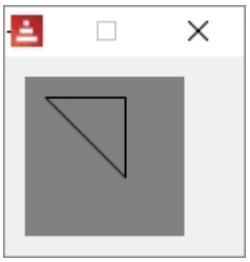
Esta é uma situação muito comum, mas o Red vai te dar um **error** porque o **DRAW não avalia expressões**.
Então você precisa usar `compose`, normalmente com o refinamento `/deep` , para uma execução bem-sucedida.

```
Red [needs: view]
view compose/deep [
      base draw [line 60x10 (2 * 10x30)]
]
```



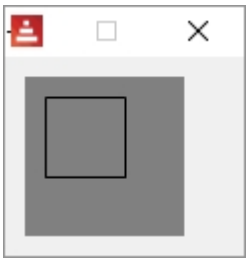
DRAW triangle

```
Red [needs: view]
view [
      base draw [triangle 10x10 50x50 50x10]
]
```



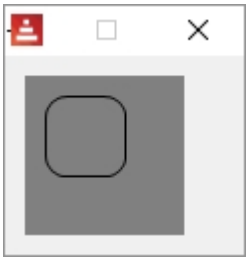
DRAW box

```
Red [needs: view]
view [
      base draw [box  10x10     50x50]
;                    top left  bottom-right
]
```



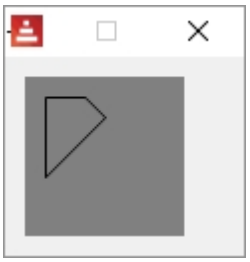
com cantos arredondados:

```
Red [needs: view]
view [
      base draw [box     10x10        50x50      10]
    ;                  top left   bottom-right    corner-radius
]
```



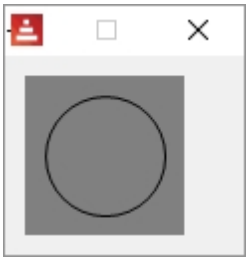
DRAW polygon

```
Red [needs: view]
view [
      base draw [polygon 10x10 30x10 40x20 30x30 10x50]
      ; it closes the polygon automatically
]
```



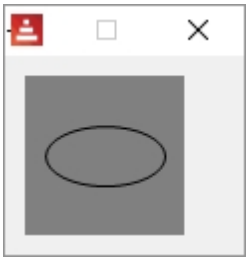
DRAW circle

```
Red [needs: view]
view [
      base draw [circle  40x40     30]
                 ;      center   radius
]
```



modo elipse:

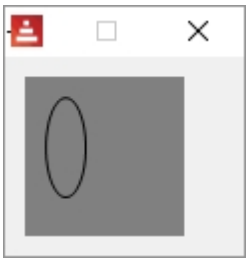
```
Red [needs: view]
view [
      base draw [circle  40x40     30        15 ]
                 ;     center  radius-x  radius-y
]
```



DRAW ellipse

A ellipse é desenhada dentro de retângulo imáginário. Os argumentos são o canto superior esquerdo e e o outro extremo deste retângulo.

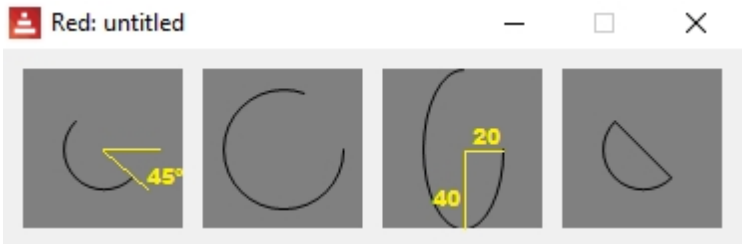
```
Red [needs: view]
view [
      base draw [ellipse 10x10 20x50]
]
```



DRAW arc

Desenha o arco de um círculo do centro (pair!) e raio (também um pair!). O arco é definido por dois ângulos fornecidos em graus. A palavra opcional `closed` pode ser usada para desenhar um arco fechado, com duas linhas partindo do centro.

```
Red [needs: view]
view [
      base draw [arc 40x40       20x20            45             180]
              ;     center  radius-x/radius-y   start angle  finish angle
      base draw [arc 40x40 30x30 0 290]
      base draw [arc 40x40 20x40 0 270]
      base draw [arc 40x40 20x20 45 180 closed]
]
```



DRAW curve

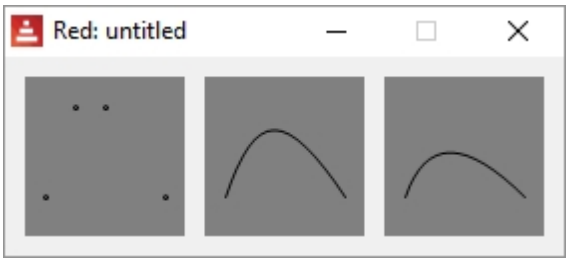
Desenha uma curva de Bezier de 3 ou de 4 pontos:

- 3 pontos: 2 pontos extremos, 1 ponto de controle.
- 4 pontos: 2 pontos extremos, 2 pontos de controle.

A opção de 4 pontos permite a criação de curvas mais complexas.

```
Red [needs: view]
view [
      ;primeiro mostramos os quatro pontos:
      base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle
70x60 1]
      ;então desenhamos as curvas:
      ;4 pontos-  ponto de partida; ponto de controle 1; ponto de
controle2; ponto de chegada
      base draw [curve 10x60 25x15 40x15 70x60]
      ; pontos-  ponto de partida; ponto de controle; ponto de chegada
      base draw [curve 10x60 25x15       70x60]
```

```
]
```

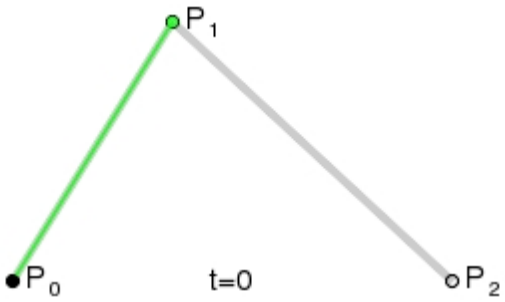


Curvas Bezier

Curvas Bezieer tem um ponto de partida, um ponto de chegada e um ou dois pontos de controle. Se tiver um ponto de controle é uma curva quadrática, se tiver dois é uma curva cúbica.

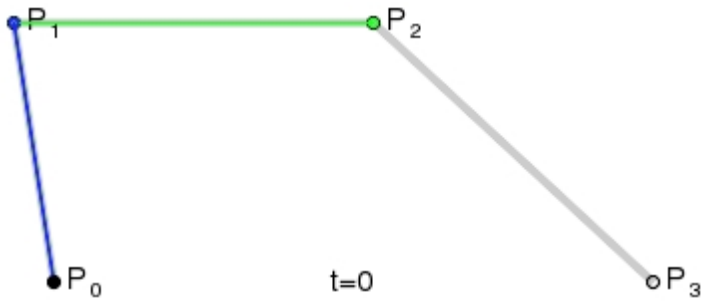
Os gifs animados abaixo foram feitos por Phil Tregoning e colocados em domínio público (obrigado) no Wikimedia Commons. Se você não puder ver a animação, olhe na página da Wikipedia sobre curvas de Bezier :

Bezier quadrática:



Veja também esta ótima demonstração interativa.

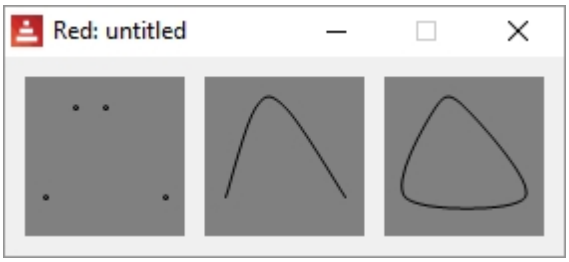
Bezier cúbica:



DRAW spline

Faz uma curva que segue uma sequência de pontos.

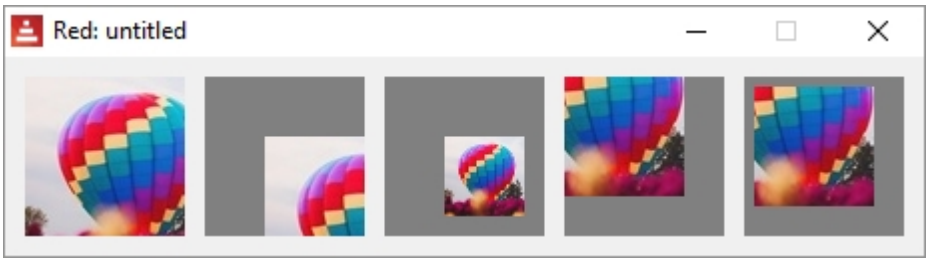
```
Red [needs: view]
view [
      ;primeiro a nos mostramos quatro pontos:
      base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle
70x60 1]
      ;depois desenhamos a curva:
      base draw [spline 10x60 25x15 40x15 70x60]
      base draw [spline 10x60 25x15 40x15 70x60 closed]
]
```



DRAW image

Coloca uma imagem usando uma posição e largura dadas.

```
Red [needs: view]
; O comando image espera uma image! não um file!
; então você precisa primeiro carregar o arquivo
picture: load %smallballoon.jpeg
view [
      base draw [image picture]
      base draw [image picture 30x30]
      base draw [image picture 30x30 70x70]
      base draw [image picture crop 30x30 60x60]
      base draw [image picture 5x5 crop 30x30 60x60]
]
```

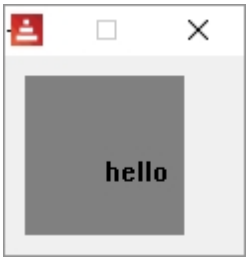


Existe também o comando `color` e o comando `border`, mas não consegui fazê-los funcionar.

```
;base draw [image picture 30x30 70x30 30x70 70x70]
;base draw [image picture 30x30 70x70 red]
;base draw [image picture 30x30 70x70 blue border]
```

DRAW text

```
Red [needs: view]
view [
	base draw [text 40x40 "hello"]
]
```



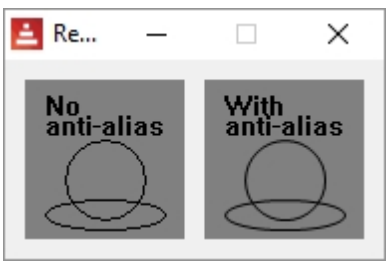
DRAW font

?

DRAW anti-alias

Anti-alias dá uma imagem mais suave, mas exige mais computaçao e, portando, diminui a performance do script. Pode ser `on` (default) ou `off`.

```
Red [needs: view]
view [
	base draw [
		anti-alias off
		text 10x5 "No"
		text 10x15 "anti-alias"
		circle 40x50 20
		ellipse 10x60 60x15

	]
	base draw [
		anti-alias on ; this is the default
		text 10x5 "With"
		text 10x15 "anti-alias"
		circle 40x50 20
		ellipse 10x60 60x15
	]
]
```



DRAW shape

Veja sub-dialeto Shape.

DRAW - Propriedade das linhas:

DRAW line-width

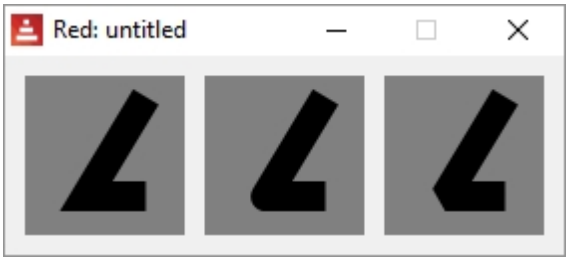
```
Red [needs: view]
view [
	base draw [
		line-width 1
		line 10x10 70x10
		line-width 5
		line 10x30 70x30
		line-width 20
		line 10x60 70x60
	]
]
```

DRAW line-join

Pode ser `miter`, `round` , `bevel` ou miter-bevel*. `miter` é o default

```
Red [needs: view]
view [
	base draw [
		line-width 15
		line-join miter
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-join round
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-join bevel
		line 60x10 30x60 60x60
	]
```

```
]
```



* não consegui fazer a miter-bevel funcionar.

DRAW line-cap

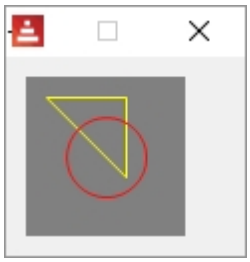
Define o modo de terminação das linhas. Pode ser `flat` (default) `square` ou `round`.

```
Red [needs: view]
view [
	base draw [
		line-width 15
		line-cap flat ;default
		line 10x20 70x20
		line-cap square
		line 10x40 70x40
		line-cap round
		line 10x60 70x60
	]
	base draw [
		line-width 15
		line-cap flat ;default
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-cap square
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-cap round
		line 60x10 30x60 60x60
	]
]
```

DRAW - Cor, gradientes e padrões

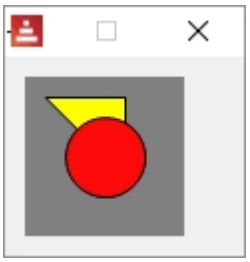
DRAW pen <cor>

```
Red [needs: view]
view [
	base draw [
		pen yellow ; cor como nome
		triangle 10x10 50x50 50x10
		pen 255.10.10 ; cor como tuple!
		circle 40x40 20
	]
]
```



DRAW fill-pen <color>

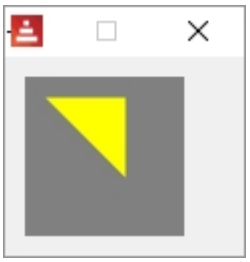
```
	Red [needs: view]
	view [
		base draw [
			fill-pen yellow ; cor como nome
			triangle 10x10 50x50 50x10
			fill-pen 255.10.10 ; cor como tuple!
			circle 40x40 20
		]
	]
```



Desligando o pen e o fill-pen:

```
	Red [needs: view]
	view [
		base draw [
			pen off
```

```
            fill-pen yellow
            triangle 10x10 50x50 50x10
            fill-pen off
            circle 40x40 20
        ]
]
```



DRAW **linear** - gradiente linear de cor

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> linear <color1> <offset> ... <colorN> <offset> <start>
<end> <spread>

<color1/N>       : list of colors for the gradient (tuple! word!).
<offset>         : (optional) offset of gradient color (float!).
<start>          : (optional) starting point (pair!).
<end>            : (optional unless <start>) ending point (pair!).
<spread>         : (optional) spread method (word!).
```

Description

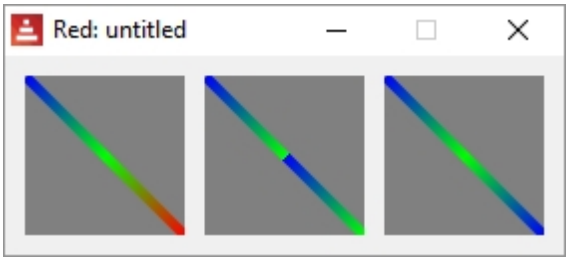
Sets a linear gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same as `repeat` for Windows platform).

When used, the start/end points define a line where the gradient paints along. If they are not used, the gradient will be paint along a horizontal line inside the shape currently drawing.

Pen

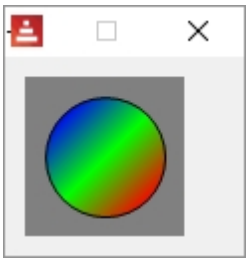
```
Red [needs: view]
view [
      base draw [
            pen linear blue green red 0x0 80x80
            line-width 5
            line 0x0 80x80
      ]
            base draw [
            pen linear blue green 0x0 40x40 pad
            line-width 5
            line 0x0 80x80
      ]
            base draw [
            pen linear blue green 0x0 40x40 reflect
```

```
            line-width 5
            line 0x0 80x80
        ]
]
```



Fill-pen

```
Red [needs: view]
view [
        base draw [
            fill-pen linear blue green red 18x18 62x62
            circle 40x40 30
        ]
]
```



DRAW radial - gradiente radial de cor

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> radial <color1> <offset> ... <colorN> <offset>
<center> <radius> <focal> <spread>

<color1/N> : list of colors for the gradient (tuple! word!).
<offset> : (optional) offset of gradient color (float!).
<center> : (optional) center point (pair!).
<radius> : (optional unless <center>) radius of the circle to paint
along (integer! float!).
<focal> : (optional) focal point (pair!).
<spread> : (optional) spread method (word!).
```

Description

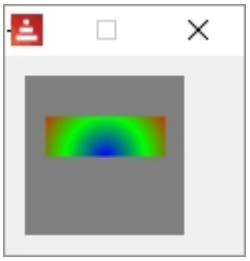
Sets a radial gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same

as `repeat` for Windows platform).

The radial gradient will be painted from focal point to the edge of a circle defined by center point and radius. The start color will be painted in focal point and the end color will be painted in the edge of the circle.

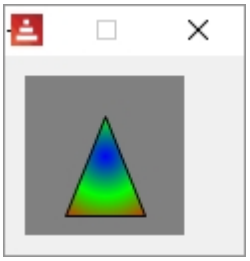
Pen

```
Red [needs: view]
view [
	base draw [
		pen radial blue green red 40x40 40 ; colors center radius
		line-width 20
		line 10x30 70x30
	]
]
```



Fill-pen

```
Red [needs: view]
view [
	base draw [
		fill-pen radial blue green red 40x40 40 ; colors center
radius
		triangle 20x70 60x70 40x20
	]
]
```



DRAW **diamond** - gradiente de cor em forma de diamante

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> diamond <color1> <offset> ... <colorN> <offset> <upper>
<lower> <focal> <spread>
```

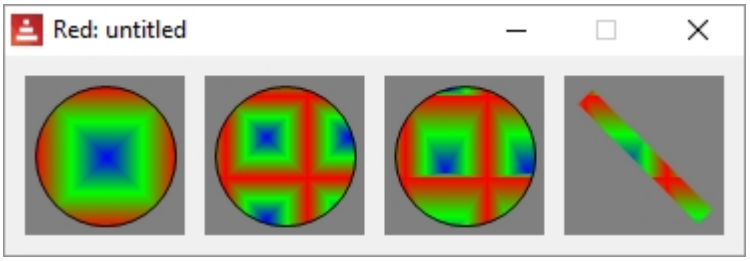
```
<color1/N> : list of colors for the gradient (tuple! word!).
<offset> : (optional) offset of gradient color (float!).
<upper> : (optional) upper corner of a rectangle. (pair!).
<lower> : (optional unless <upper>) lower corner of a rectangle
(pair!).
<focal> : (optional) focal point (pair!).
<spread> : (optional) spread method (word!).
```

Description

Sets a diamond-shaped gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same as `repeat` for Windows platform).

The diamond gradient will be painted from focal point to the edge of a rectangle defined by upper and lower. The start color will be painted in focal point and the end color will be painted in the edge of the diamond.

```
Red [needs: view]
view [
	base draw [
		fill-pen diamond  blue green red ; just centers the gradient
		circle 40x40 35
	]
	base draw [
		fill-pen diamond  blue green red 10x10 50x50 ;added
coordinates of the gradient "box"
		circle 40x40 35
	]
	base draw [
		fill-pen diamond  blue green red 10x10 50x50 30x48; added a
point of focus
		circle 40x40 35
	]
	base draw [
		pen diamond  blue green red 10x10 50x50 30x48
		; a line over the last gradient:
		line-width 10
		line 10x10 70x70
	]
]
```



DRAW **bitmap** - preenchimento bitmap

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> bitmap <image> <start> <end> <mode>

<image> : image used for tiling (image!).
<start> : (optional) upper corner for crop section within image
(pair!).
<end> : (optional) lower corner for crop section within image
(pair!).
<mode> : (optional) tile mode (word!).
```

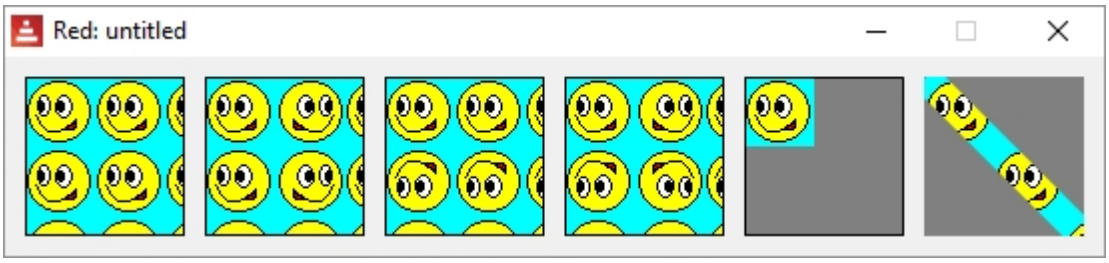
Description

Sets an image as pattern to be used for filling operations. The following values are accepted for the tile mode: `tile` (default), `flip-x`, `flip-y`, `flip-xy`, `clamp`.

Starting default point is 0x0 and ending point is image's size.

O bitmap usado dos exemplos é: 

```
Red [needs: view]
myimage: load %asprite.bmp  ; primeiro carrega o bitmap
view [
      base draw [
            fill-pen bitmap myimage tile ; padrão (default)
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-x
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-y
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-xy
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage clamp
            box 0x0 79x79
      ]
            base draw [
            pen bitmap myimage
            line-width 15
            line 0x0 80x80
      ]
]
```



DRAW **pattern** - preenchimento com padrão desenhado

From Red's official documentation (with eventual minor changes):

Syntax

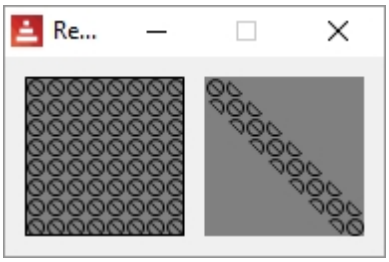
```
<pen-fill-pen> pattern <size> <start> <end> <mode> [<commands>]

<size> : size of the internal image where <commands> will be drawn
(pair!).
<start> : (optional) upper corner for crop section within internal
image (pair!).
<end> : (optional) lower corner for crop section within internal image
(pair!).
<mode> : (optional) tile mode (word!).
<commands> : block of Draw commands to define the pattern.
```

Description

Sets a custom shape as pattern to be used for filling operations. The following values are accepted for the tile mode: `tile` (default), `flip-x`, `flip-y`, `flip-xy`, `clamp`.

Starting default point is 0x0 and ending point is `<size>`.

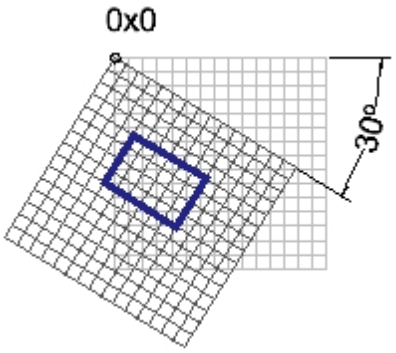
```
Red [needs: view]
view [
	; first we draw a filled box:
	base draw [
		fill-pen pattern 10x10 [
			circle 5x5 4
			line 3x3 7x7
		]
		box 0x0 79x79
	]
	; then we draw a line:
	base draw [
		pen pattern 10x10 [
			circle 5x5 4
			line 3x3 7x7
		]
		line-width 15
		line 0x0 79x79
	]
]
```


Re...

DRAW - 2D transforms

DRAW rotate

Exemplo de uma rotação de 30º centrada em 0x0:



Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
rotate <ângulo> <centro> [<commands>]
rotate 'pen <ângulo> rotate 'fill-pen <ângulo>

<ângulo> :  ângulo em graus (integer! float!).
<centro> : (opcional) centro de rotação (pair!).
<commands> : (opcional) Comandos do dialeto Draw.
```

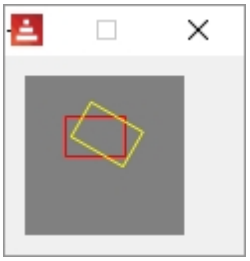
Descrição

Define a rotação no sentido horário em um determinado ponto, em graus. Se o centro opcional não for fornecido, a rotação é sobre a origem do atual sistema de coordenadas do usuário. Números negativos podem ser usados para rotação no sentido anti-horário. Quando um bloco é fornecido como último argumento, a rotação será aplicada apenas aos comandos nesse bloco.

Quando as palavras-chave 'pen' ou 'fill-pen' são usadas, a rotação é aplicada respectivamente à caneta atual ou à caneta de preenchimento atual.

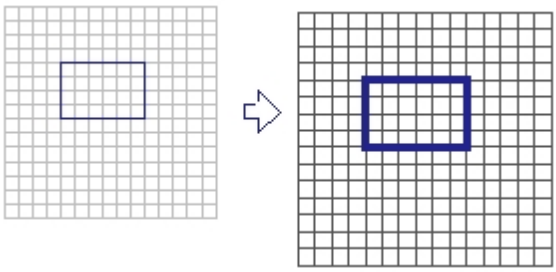
```
Red [needs: view]
view [
      base draw [
          pen red
            box 20x20 50x40 ; retângulo horizontal
            rotate 30 40x40 ; Rotação de 30 graus centrada em 40x40
```

```
            pen yellow
            box 20x20 50x40 ; mesmo comando, outro retângulo
            ]
]
```



DRAW scale

Exemplo de um scale de 1.2 em ambos os eixos, **x** e **y**:



Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
scale <scale-x> <scale-y> [<commands>]
scale 'pen <scale-x> <scale-y>
scale 'fill-pen <scale-x> <scale-y>

<scale-x> : escala em X (number!).
<scale-y> : escala em Y (number!).
<commands> : (opcional) Comandos do dialeto Draw.
```

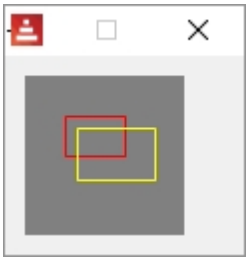
Descrição

Define os valores da escala. Os valores dados são multiplicadores; use valores maiores que um para aumentar a escala; use valores menores que um para diminuí-lo. Quando um bloco é fornecido como último argumento, o *scaling* será aplicado apenas aos comandos desse bloco.

Quando as palavras-chave 'pen' ou 'fill-pen' são usadas, a escala será aplicada, respectivamente, à caneta atual ou à caneta de preenchimento atual.

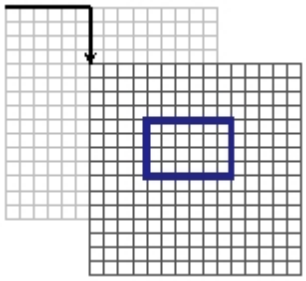
```
Red [needs: view]
view [
      base draw [
          pen red
            box 20x20 50x40 ; horizontal rectangle
            scale 1.3 1.3 ;30% bigger in both x and y
```

```
            pen yellow
            box 20x20 50x40 ; same command, different box
            ]
]
```



DRAW translate

Exemplo de uma translação nos eixos **x** e **y**:

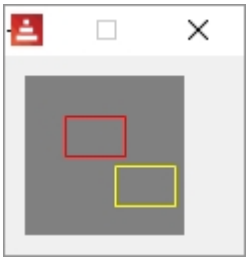


A translação leva todo o sistema de coordenadas para a nova posição.

> Da Documentação oficial do Red (com eventuais pequenas mudanças):
>
> **Sintaxe**
>
> ```
> translate <offset> [<commands>]
> translate 'pen <offset>
> translate 'fill-pen <offset>
>
> <offset> : quantidades de translação (pair!).
> <commands> : (opcional) Comandos do dialeto Draw.
> ```
>
> **Descrição**
>
> Define a origem para os comandos de desenho. Vários comandos de conversão terão um efeito cumulativo. Quando um bloco é fornecido como último argumento, a translação será aplicada apenas aos comandos desse bloco.
>
> Quando as palavras-chave 'pen' ou 'fill-pen' são usadas, a translação é aplicada respectivamente à caneta atual ou à caneta de preenchimento atual.

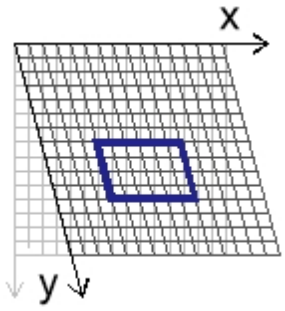
```
Red [needs: view]
view [
      base draw [
          pen red
            box 20x20 50x40 ; horizontal rectângulo
```

```
            translate 25x25
            pen yellow
            box 20x20 50x40 ; same command, different box
            ]
]
```



DRAW skew

Um sistema de coordenadas *skewed* é quando os eixos não são ortogonais.



O comando skew inclina os eixos **x** e/ou **y** um dado número de graus.

Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
skew <skew-x> <skew-y> [<commands>]
skew 'pen <skew-x> <skew-y>
skew 'fill-pen <skew-x> <skew-y>

<skew-x> : inclinação do eixo X em graus (integer! float!).
<skew-y> : (opcional) inclinação do eixo Y em graus (integer!
float!).
<commands> : (opcional) Comandos do dialeto Draw.
```

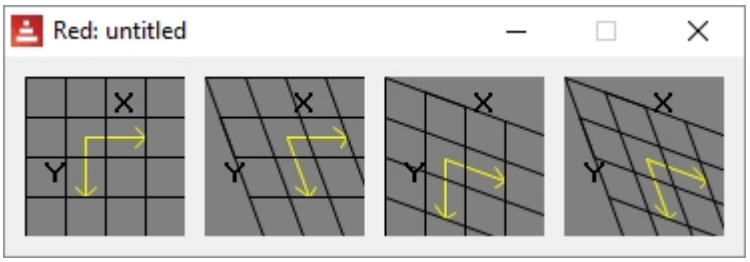
Description

Define um sistema de coordenadas inclinado do original pelo número de graus fornecido. Se <skew-y> não for fornecido, será considerado zero. Quando um bloco é fornecido como último argumento, a inclinação será aplicada somente aos comandos nesse bloco.

Quando as palavras-chave 'pen' ou 'fill-pen' são usadas, a inclinação é aplicada respectivamente à caneta atual ou à caneta de preenchimento atual.

```
Red [needs: view]
view [
      base draw [
            pen yellow        ; só desenha duas flechas
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black         ; só desenha uma grade
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
      ]

      base draw [
            skew 20 0 ;inclina o eixo x 20 graus
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X" ;the text does not follow skew!
            text 10x40 "Y"
      ]
      base draw [
            skew 0 20 ; inclina o eixo y 20 graus
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
      ]
      base draw [
            skew 20 20 ; inclina os dois eixos 20 graus
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
```

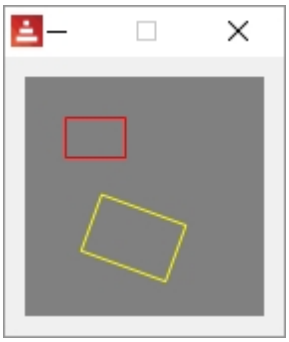
```
        ]
]
```



DRAW transform

Executa a translação, rotação e escala em um único comando. A transformação abaixo usa 0x0 como ponto de ancoragem (ponto de referência), gira 20º, escala para 1,5 em ambos os eixos e translada 20 unidades nos eixos x e y:

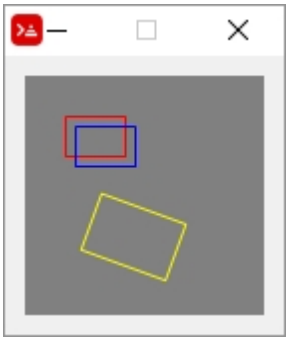
```
Red [needs: view]
view [
      base 120x120 draw [
          pen red
            box 20x20 50x40 ; retângulo horizontal
            transform 0x0 20 1.5 1.5 20x20
            pen yellow
            box 20x20 50x40 ; mesmo comando, outro retângulo
            ]
]
```



Se um bloco for fornecido como último argumento, essas transformações serão aplicadas apenas aos comandos nesse bloco.

```
Red [needs: view]
view [
      base 120x120 draw [
          pen red
            box 20x20 50x40 ; primeiro retângulo, vermelho
            transform 0x0 20 1.5 1.5 20x20 [
                  pen yellow
                  box 20x20 50x40 ; segundo retângulo, amarelo
            ]
```

```
            pen blue
            box 25x25 55x45 ; terceiro retângulo, azul
            ]
]
```



Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
transform <centro> <ângulo> <scale-x> <scale-y> <translation>
[<commands>]
transform 'pen <centro> <ângulo> <scale-x> <scale-y>
<translation>
transform 'fill-pen <centro> <ângulo> <scale-x> <scale-y>
<translation>

<centro> : (opcional) centro de rotação (pair!).
<ângulo> : ângulo de rotação em graus (integer! float!).
<scale-x> : escala em X (number!).
<scale-y> : escala em Y (number!).
<translation> : valores de translação (pair!).
<commands> : (opcional) Comandos do dialeto Draw.
```

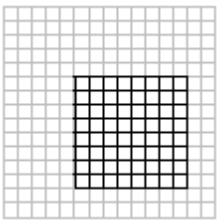
Descrição

Define uma transformação, como translação, escala e rotação. Quando um bloco é fornecido como último argumento, a transformação será aplicada apenas aos comandos nesse bloco.

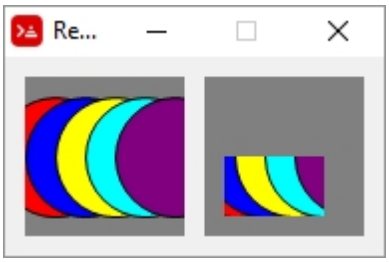
Quando as palavras-chave 'pen' ou 'fill-pen' são usadas, a transformação é aplicada respectivamente à caneta atual ou à caneta de preenchimento atual.

DRAW clip

Limita a área de desenho a um retângulo.



```
Red [needs: view]
view [
      base
      draw [
            pen black
            fill-pen red circle 15x40 30
            fill-pen blue circle 30x40 30
            fill-pen yellow circle 45x40 30
            fill-pen cyan circle 60x40 30
            fill-pen purple circle 75x40 30
      ]
      base
      draw [
            clip 10x40 60x70
            pen black
            fill-pen red circle 15x40 30
            fill-pen blue circle 30x40 30
            fill-pen yellow circle 45x40 30
            fill-pen cyan circle 60x40 30
            fill-pen purple circle 75x40 30
      ]
]
```



Se um bloco é fornecido como último argumento, o recorte é aplicado apenas aos comandos nesse bloco, ou seja, após o bloco, toda a área se torna novamente uma tela.

Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
clip <start> <end> <mode> [<commands>]
clip [<shape>] <mode> [<commands>]

<start> : canto superior esquerdo da área de recorte (pair!)
<end> : canto inf. direito da área de recorte (pair!)
<mode> : (opcional) modo de mistura das áreas recortadas (word!)
<commands> : (opcional) Comandos do dialeto Draw.
<shape> : Comandos do dialeto Shape.
```

Descrição

Define uma região retangular de recorte definida com dois pontos (início e fim) ou uma região com formato arbitrário definida por um bloco de comandos de sub-dialetos Shape. Esse recorte se aplica a todos os comandos subseqüentes do Draw. Quando um bloco é fornecido como último argumento, o recorte será aplicado apenas aos comandos nesse bloco.

Além disso, o modo de combinação entre uma nova região de recorte e a anterior pode ser definido como um dos seguintes:

- `replace` (default)
- `intersect`
- `union`
- `xor`
- `exclude`

De todos este modos, eu só consegui entender `replace` e `exclude`. Você pode tentar os outros.

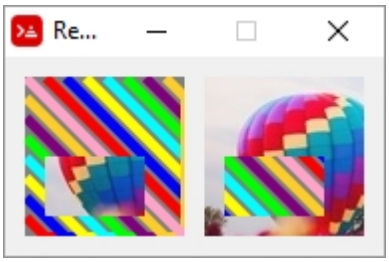
```
Red [needs: view]

view [
	base
	draw [
		line-width 5
		pen red line 0x70 10x80 80x80 80x70 10x0
		pen blue line 0x60 20x80 80x80 80x60 20x0
		pen yellow line 0x50 30x80 80x80 80x50 30x0
		pen cyan line 0x40 40x80 80x80 80x40 40x0
		pen green line 0x30 50x80 80x80 80x30 50x0
		pen purple line 0x20 60x80 80x80 80x20 60x0
		pen gold line 0x10 70x80 80x80 80x10 70x0
		pen pink line 0x0 80x80 80x80

		clip 10x40 60x70 replace ;default

		pen red line 0x10 10x0 80x0 80x10 10x80
		pen blue line 0x20 20x0 80x0 80x20 20x80
		pen yellow line 0x30 30x0 80x0 80x30 30x80
		pen cyan line 0x40 40x0 80x0 80x40 40x80
		pen green line 0x50 50x0 80x0 80x50 50x80
		pen purple line 0x60 60x0 80x0 80x60 60x80
		pen gold line 0x70 70x0 80x0 80x70 70x80
		pen pink line 0x80 80x0 80x80

	]
	base
	draw [
		line-width 5
		pen red line 0x70 10x80 80x80 80x70 10x0
		pen blue line 0x60 20x80 80x80 80x60 20x0
		pen yellow line 0x50 30x80 80x80 80x50 30x0
		pen cyan line 0x40 40x80 80x80 80x40 40x0
		pen green line 0x30 50x80 80x80 80x30 50x0
		pen purple line 0x20 60x80 80x80 80x20 60x0
```

```
            pen gold line 0x10 70x80 80x80 80x10 70x0
            pen pink line 0x0 80x80 80x80

            clip 10x40 60x70 exclude

            pen red line 0x10 10x0 80x0 80x10 10x80
            pen blue line 0x20 20x0 80x0 80x20 20x80
            pen yellow line 0x30 30x0 80x0 80x30 30x80
            pen cyan line 0x40 40x0 80x0 80x40 40x80
            pen green line 0x50 50x0 80x0 80x50 50x80
            pen purple line 0x60 60x0 80x0 80x60 60x80
            pen gold line 0x70 70x0 80x0 80x70 70x80
            pen pink line 0x80 80x0 80x80

      ]
]
```

Ou, usando uma imagem:

```
Red [needs: view]
picture: load %smallballoon.jpeg
view [
      base
      draw [
            line-width 5
            pen red line 0x70 10x80 80x80 80x70 10x0
            pen blue line 0x60 20x80 80x80 80x60 20x0
            pen yellow line 0x50 30x80 80x80 80x50 30x0
            pen cyan line 0x40 40x80 80x80 80x40 40x0
            pen green line 0x30 50x80 80x80 80x30 50x0
            pen purple line 0x20 60x80 80x80 80x20 60x0
            pen gold line 0x10 70x80 80x80 80x10 70x0
            pen pink line 0x0 80x80 80x80

            clip 10x40 60x70 replace ;default

            image picture
      ]
      base
      draw [
            line-width 5
            pen red line 0x70 10x80 80x80 80x70 10x0
            pen blue line 0x60 20x80 80x80 80x60 20x0
            pen yellow line 0x50 30x80 80x80 80x50 30x0
            pen cyan line 0x40 40x80 80x80 80x40 40x0
            pen green line 0x30 50x80 80x80 80x30 50x0
            pen purple line 0x20 60x80 80x80 80x20 60x0
            pen gold line 0x10 70x80 80x80 80x10 70x0
            pen pink line 0x0 80x80 80x80
```

```
            clip 10x40 60x70 exclude

            image picture
        ]
]
```



DRAW - sub-dialeto Shape

O sub-dialeto Shape permite criar formas (desenhos, shapes) como blocos.
Alguns aspectos me lembram de "gráficos de tartaruga" ("turtle-graphics"). Você pode mover a caneta sem desenhar e as coordenadas podem ser absolutas (em relação à face) ou relativas (em relação à última posição).

O sub-dialeto shape também "fecha" as formas para você, permitido o uso de `fill-pen` para adicionar cores e padrões.

Você pode usar `fill-pen` , `pen` , `line-width` , `line-join` e `line-cap` como comandos no bloco de shape, mas apenas o último comando vai ser usado para toda a forma.

O sub-dialeto Shape é baseado em gráficos SVG. Eu achei os links abaixo úteis para entender alguns conceitos:

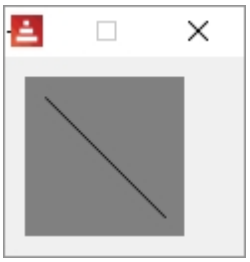
https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths

http://www.w3.org/TR/SVG11/paths.html

⊕ line

O exemplo mais básico:

```
Red [needs: view]

myshape: [line 10x10 70x70]

view compose/deep/only [
      base draw [
          shape (myshape)
      ]
]
```

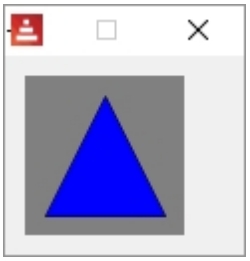


Note o `compose/deep/only` e os parentesis ao redor do nome da shape. Pelo que eu sei, você precisa usá-los quando trabalha com Shapes.

Fechamento automático

No exemplo abaixo, apenas duas linhas são, de fato, desenhadas. A terceira linha, de fechamento, é automática. Eu adicionei o `fill-pen` para ilustrar o conceito melhor:

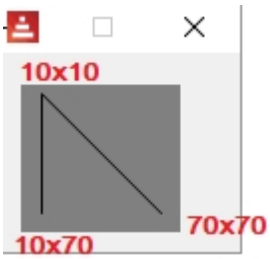
```
Red [needs: view]

myshape: [
	line 10x70 40x10 70x70 ;só duas linhas
]
view compose/deep/only [base draw [ fill-pen blue shape (myshape)]]
```



⊕ move

Exemplo mais basico:

```
Red [needs: view]

myshape: [
	line 10x10 70x70 ;linha de 10x10 para 70x70
	move 10x70 ;move a pen sem desenhar para 10x70
	line 10x10 ;desenha uma linha da posição corrente (10x70) até
10x10
]

view compose/deep/only [base draw [shape (myshape)]]
```

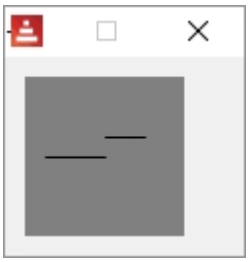


posições relativas

As coordenadas se tornam relativas se você adicionar um apóstrofe (') antes do comando:

```
Red [needs: view]

myshape: [
	line 10x40 40x40 ;linha horizontal no meio
	'move 0x-10 ;nova posição corrente RELATIVA à antiga (acima do
```

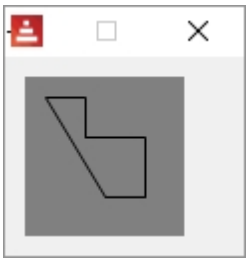
```
meio)
      'line 20x0 ;desenha uma pequena linha horizontal RELATIVA a
posição corrente
]

view compose/deep/only [base draw [shape (myshape)]]
```



⊕ hline e ⊕ vline

Desenham linhas horizontais ou verticais a partir da posição corrente.

```
Red [needs: view]

myshape: [
      move 10x10 ; coloca a caneta em 10x10
      hline 30 ;linha horizontal X =30
      vline 30 ;linha vertical Y = 30
      'hline 30 ;linha horizontal de 30 pixels (maior que a hline acima)
      'vline 30 ;linha vertical de 30 pixels
      'hline -20 ; só para mostrar o uso de distâncias negativas
RELATIVAS
      ; o dialeto shape vai fechar a forma agora.
]

view compose/deep/only [base draw [shape (myshape)]]
```



⊕ arc

Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

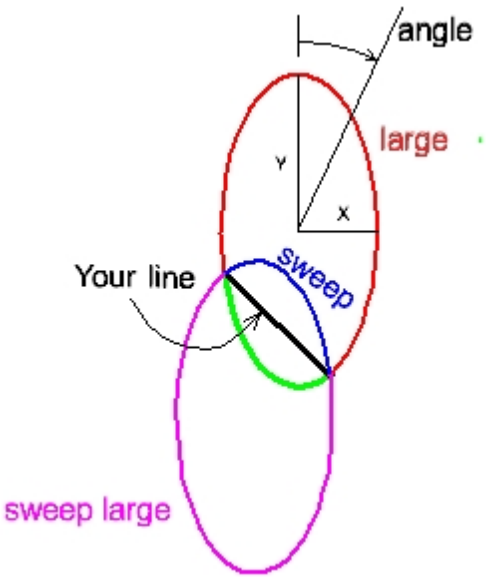
```
arc <end> <radius-x> <radius-y> <angle> sweep large (absolute)
```

```
'arc <end> <radius-x> <radius-y> <angle> sweep large (relative)

<end> : arc's end point (pair!).
<radius-x> : radius of the circle along x axis (integer! float!).
<radius-y> : radius of the circle along y axis (integer! float!).
<angle> : angle between the starting and ending points of the arc
in degrees (integer! float!).
sweep : (optional) draw the arc in the positive angle direction.
large : (optional) produces an inflated arc (goes with 'sweep
option).
```

Descrição

Desenha o arco de um círculo entre a posição atual da caneta e o ponto final, usando valores de raio. O arco é definido por um valor de ângulo

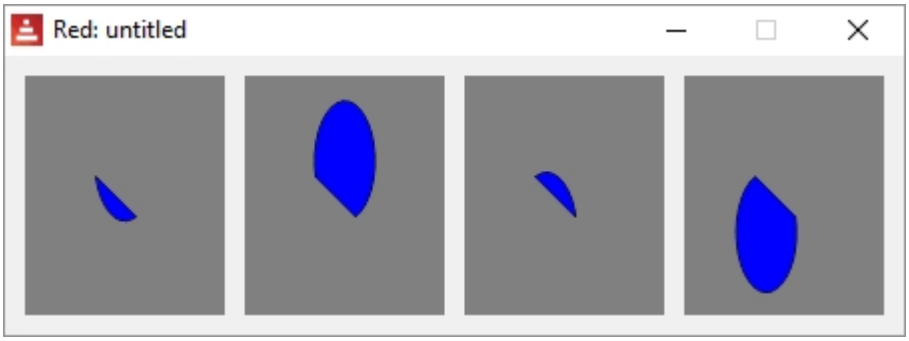
Aqui está uma explicação sobre como o arco funciona. Como você define sua linha (dois pontos) e sua elipse (raio-x, raio-y e ângulo), existem apenas duas posições para a elipse que fazem de sua linha um acorde para ela. As opções `sweep`, `large` e `sweep large` definem qual arco dessas elipses aparecerá em seu desenho. Observe que na ilustração abaixo, o ângulo da elipse é zero.



In the `arc` definition you only inform the arc's end position. That is because the start position is the current pen position. So, if `arc` is your first command in a shape, you must first `move` to the position you want to start at.
Na definição do `arc`, você só informa a posição final do arco. Isso porque a posição inicial é a posição atual da caneta. Então, se `arc` é seu primeiro comando em uma forma, você deve primeiro ir para a posição onde deseja começar.

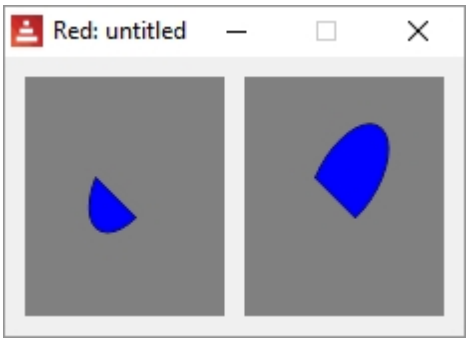
```
Red [needs: view]

myshape_1: [
    move 35x50
    arc  55x70 15 30 0
```

```
]
myshape_2: [
    move 35x50
    arc  55x70 15 30 0 large sweep
]
myshape_3: [
    move 35x50
    arc  55x70 15 30 0 sweep
]
myshape_4: [
    move 35x50
    arc  55x70 15 30 0 large
]
view compose/deep/only [
      base 100x120 draw [fill-pen blue shape (myshape_1)]
      base 100x120 draw [fill-pen blue shape (myshape_2)]
      base 100x120 draw [fill-pen blue shape (myshape_3)]
      base 100x120 draw [fill-pen blue shape (myshape_4)]
]
```



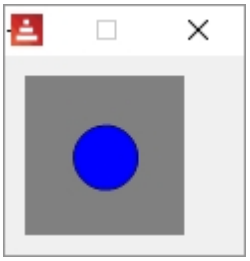
Com um ângulo:

```
Red [needs: view]

myshape_1: [
    move 35x50
    arc  55x70 15 30 30
]
myshape_2: [
    move 35x50
    arc  55x70 15 30 30 large sweep
]

view compose/deep/only [
      base 100x120 draw [fill-pen blue shape (myshape_1)]
      base 100x120 draw [fill-pen blue shape (myshape_2)]
]
```



Um círculo:

```
Red [needs: view]
myshape_1: [
    move 56x40
    arc  56x41 16 16 0 large
]
view compose/deep/only [base  draw [fill-pen blue shape (myshape_1)]]
```



⊕ qcurve

Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
qcurve <point> <point> ... (absolute)
'qcurve <point> <point> ... (relative)

<point> : coordinates of a point (pair!).
```

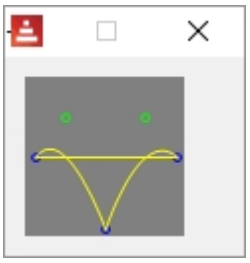
Descrição

Desenha uma curva quadrática de Bézier a partir de uma sequência de pontos, a partir da posição atual da caneta. Pelo menos 2 pontos são necessários para produzir uma curva (o primeiro ponto é o ponto de partida implícito).

Desenha uma curva Bezier quadrática a partir de uma sequência de 3 pontos. O script a seguir desenha duas curvas uasndo <ponto de partida> <ponto de controle > <ponto de chegada/ponto de partidat> <ponto de controle > <ponto de chegada>. Permite cordenadas apsolutas ou relativas (para relativas usar `'qcurve`) .

```
Red [needs: view]
myshape: [
    move 5x40
    qcurve  20x20 40x76 60x20 76x40
]
view compose/deep/only [
      base  draw [
            pen blue
            circle 5x40 2 ;mostra ponto de partida 1
            circle 40x76 2 ;mostra ponto de chegada 1/ponto de partida 2
            circle 76x40 2 ;mostra ponto de chegada 2
            pen green
            circle 20x20 2 ;mostra ponto de controle 1
            circle 60x20 2 ;mostra ponto de controle 2
            pen yellow
            shape (myshape)
            ]
]
```

Eu adicionei a localização aproximada dos pontos fixos (azul) e os pontos de controle (verde) na imagem abaixo. Eles não são gerados pelo programa, eu editei a imagem.



⊕ curve

Da Documentação oficial do Red (com eventuais pequenas mudanças):

Sintaxe

```
curve <point> <point> <point> ... (absolute)
'curve <point> <point> <point> ... (relative)

<point> : coordinates of a point (pair!).
```

Descrição

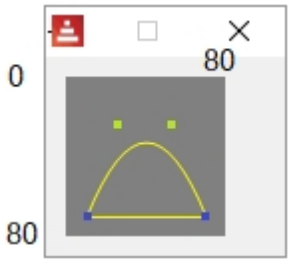
Desenha uma curva cúbica de Bezier a partir de uma sequência de pontos, a partir da posição atual da caneta. Pelo menos 3 pontos são necessários para produzir uma curva (o primeiro ponto é o ponto de partida implícito).

Desenha uma curva Bezier cúbica usando <ponto de partida (posição corrente)> <ponto de controle 1 (argumento)> <controlponto de controle 2 (argument0)> <ponto de chegada (argumento)> . Permite coordenadas absolutas ou relativas (para relativa `'curve`).

```
Red [needs: view]
```

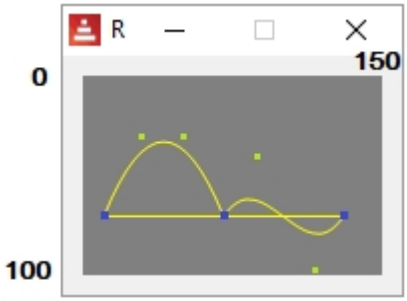
```
myshape_1: [
      move 10x70 ; ponto de partida
      curve 30x20 50x20 70x70 ; ponto de controle; ponto de controle;
ponto de chegada
]
view compose/deep/only [base draw [ pen yellow shape (myshape_1)]]
```

Eu adicionei a localização aproximada dos pontos fixos (azul) e os pontos de controle (verde) nas imagens abaixo. Eles não são gerados pelo programa, eu os editei.



Você pode adicionar mais pontos ao comando `curve` eles vão criar uma nova curva independente:

```
Red [needs: view]
myshape_1: [
      move 10x70 ; ponto de partida
      curve 30x20 ;primeiro ponto de controle
            50x20 ;segundo ponto de controle
            70x70 ;ponto de chegada da primeira curva/ponto de partida
da segunda curva
            90x40 ;primeiro ponto de controle da segunda curva
            110x100 ;segundo ponto de controle da segunda curva
            130x70 ponto de chegada da segunda curva
]
view compose/deep/only [base 150x100 draw [ pen yellow
shape(myshape_1)]]
```



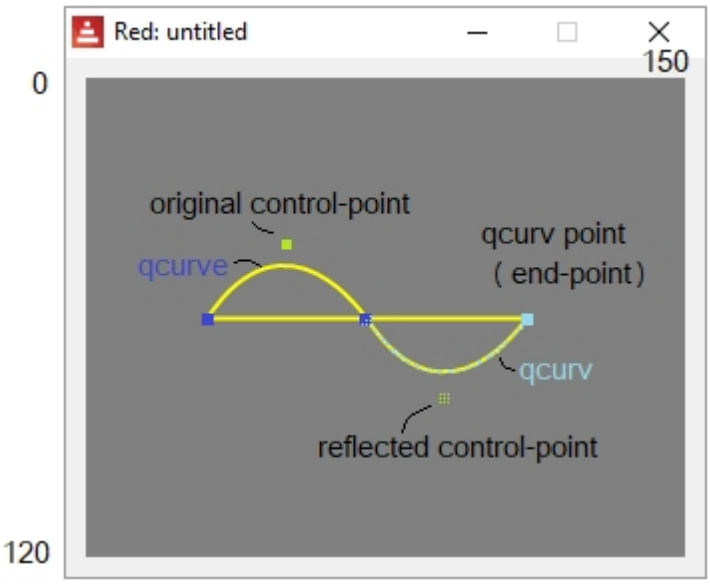
⊕ qcurv

Sintaxe

```
qcurv <point> (absolute)
'qcurv <point> (relative)

<point> : coordenadas do ponto de chegada (pair!).
```

`qcurv` desenha um Bezier quadrático suave da posição atual da caneta até o ponto especificado.
Você não precisa fornecer o ponto de controle entre o ponto de partida e o ponto final, `qcurv` cria esses pontos de controle como um reflexo do último ponto de controle dado no bloco de shape, portanto, você deve ter um comando que use um ponto de controle antes de usar o `qcurv`.

```
Red [needs: view]

myshape_1: [
	move 30x60 ;ponto de partida da qcurve
	qcurve 50x30 70x60 ;ponto de controle; ponto de chegada da qcurve
	qcurv 110x60 ; ponto de chegada da qcurv
]
view compose/deep/only [
	base 300x240 draw [
		scale 2 2 ; só aumenta a escala para visualisar melhor
		pen yellow
		shape (myshape_1)
		]
]
```



Pelo menos até abril de 2018, `qcurv` só funciona com um ponto de chegada como argumento.

⊕ curv

Desenha uma curva Bezier cúbica suave a partir de uma sequência de pontos.

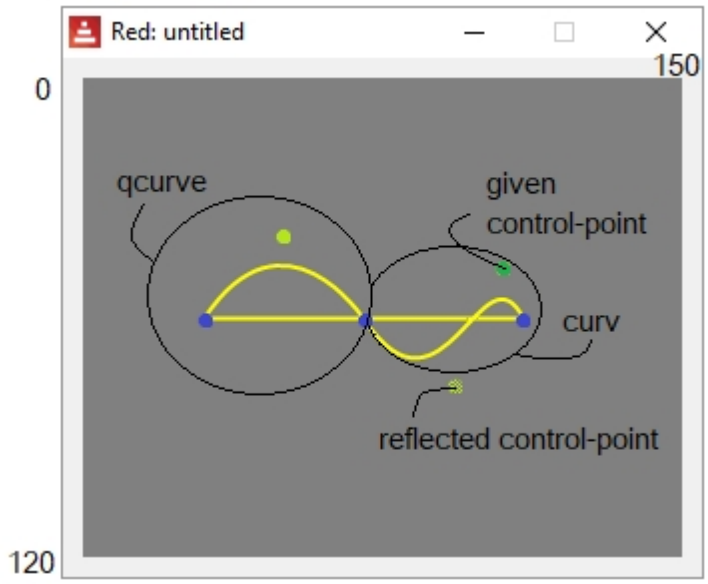
Assim como o `qcurv`, o `curv` cria pontos de controle refletidos em relação ao último ponto de controle no bloco de forma. Mas como os Béziers cúbicos requerem 2 pontos de controle, você deve fornecer o segundo para cada segmento. Este segundo ponto de controle será refletido como o primeiro ponto de controle do próximo segmento.

> Da Documentação oficial do Red (com eventuais pequenas mudanças):
>
> **Sintaxe**
>
> ```
> curv <point> <point> ... (absolute)
> 'curv <point> <point> ... (relative)
>
> <point> : coordinates of a point (pair!).
> ```
>
> **Descrição**
>
> Desenha uma curva Bezier cúbica suave a partir de uma sequência de pontos, a partir da posição atual da caneta. Pelo menos 2 pontos são necessários para produzir uma curva (o primeiro ponto é o ponto de partida implícito).
>
> "O primeiro ponto de controle é considerado o reflexo do segundo ponto de controle no comando anterior em relação ao ponto atual. (Se não houver um comando de curva anterior, o primeiro ponto de controle será o ponto atual.)"

Então, `curv` desenha uma Bezier usando <posição corrente como ponto de partida >< ponto de controle 1 criado automaticamente><ponto de controle 2> <ponto de chegada>.

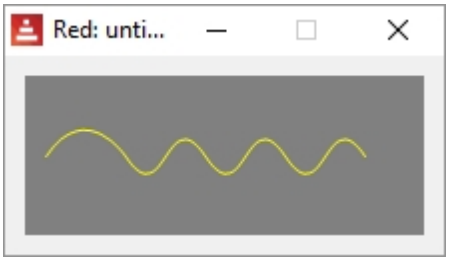
Assim, os argumentos que você efetivamente passa para `curv` são só: <ponto de controle 2> <ponto de chegada>[...]

```
Red [needs: view]

myshape_1: [
      move 30x60 ;ponto de partida da qcurve
      qcurve 50x30 70x60 ;ponto de controle; ponto de chegada
      curv 100x40 110x60 ; segundo ponto de controle da curv e ponto de
chegada
]
view compose/deep/only [
      base 300x240 draw [
            scale 2 2 ; aumentando só para visualização
            pen yellow
            shape (myshape_1)
            ]
]
```



`curv` pode usar muitos pontos de controle e pontos consecutivos:

```
Red [needs: view]

      ;segundo ponto de controle    ponto

myshape_1: [
      move 10x40
      qcurve 30x10 50x40
      curv  70x10 90x40 110x10 130x40 150x10 170x40
      move 10x40
]
view compose/deep/only [base 200x80 draw [ pen yellow shape (myshape_1)]
]
```



DRAW - Desenhos e animação programáticos

Executar desenhos usando ferramentas de programação Red (loops, matemática, ramificações, etc.) requer alguma estruturação do script. Eu sugiro a seguinte estrutura como regra geral:

```
Red [needs: view]
draw-changing: function [ ]
view compose/deep/only [
        face focus
        draw [commands (arguments)]
        on-event [ draw-changing ]
]
```

draw-changing - Estas são as funções a serem chamadas de um *event* para fazer cálculos e, em seguida, alterar o campo "draw" do objeto da *face*. Você deve alterar este campo daqui porque não pode mudá-lo de dentro do bloco do dialeto draw.

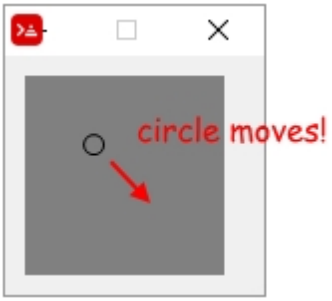
face focus - Alguns *events* (como `key`) parecem só ser gerado se houver `focus` nas faces como `base` ou `box`, cuidado.

draw - Executa o dialeto draw. Qualquer argumento calculado (variável) deve estar entre parênteses para ser computado por `compose/deep/only`.

on-event - Chama a função apropriada de *draw-changing*, considerando o tipo de evento.

Animação simples:

```
Red [needs 'view]

position: 0x0

update-canvas: func [] [
    position: position + 1x1
    canvas/draw:  reduce ['circle position 5]
]

view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```



Explicando o código:

```
Red [needs 'view]

{ "position" é o centro do círculo que vai se mover
aqui ele está no canto superior esquerdo}

position: 0x0

{a função "update-canvas" faz todo o
processamento necessário e "passa" a
rotina de draw para o objeto de "canvas".
Observe três coisas no código abaixo:
1- Sim, draw é um campo de um objeto!
2- Você deve usar "reduce" para enviar o
valor atual da posição;
3- Deve haver um apóstrofo antes
"circle". "circle" é um comando do
dialeto draw, e por isso deve ser passado "como é"}


update-canvas: func [] [
    position: position + 1x1
    canvas/draw:  reduce ['circle position 5]
]

{A rotina do view cria uma base chamada
"canvas" que se atualiza 25 vezes
por segundo}


view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```

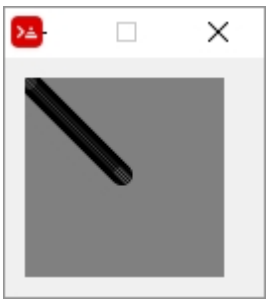
Para mostrar que a **canvas** é um objeto!, feche a visualização gráfica depois uns instantes, mas deixe o console aberto. Digite `? canvas` no console. Você vai ter:

```
>> ? canvas
CANVAS is an object! with the following words and values:
     type              word!         base
     offset            pair!         10x10
     size              pair!         100x100
     text              none!         none
     image             none!         none
```

```
     color             tuple!        128.128.128
     menu              none!         none
     data              none!         none
     enabled?          logic!        true
     visible?          logic!        true
     selected          none!         none
     flags             none!         none
     options           block!        length: 6  [style: base vid-align:
top at-o...
     parent            object!       [type offset size text image color
menu dat...
     pane              none!         none
     state             none!         none
     rate              integer!      25
     edge              none!         none
     para              none!         none
     font              none!         none
     actors            object!       [on-time]
     extra             none!         none
     draw              block!        length: 3  [circle 37x37 5]
     on-change*        function!     [word old new /local srs same-pane?
f saved]
     on-deep-change*   function!     [owner word target action new index
part]
```

No próximo exemplo, em vez de alterar o bloco de `draw` , vamos fazer um `append` com novos comandos de `draw` . O resultado é que todos os desenhos anteriores são mantidos (na verdade, são redesenhados, mas ...), criando um rastro de desenhos:

```
Red [ needs 'view ]
position: 0x0
command: [] ; initialized to prevent error.

update-canvas: func [] [
    position: position + 1x1
    append command reduce ['circle position 5]
    canvas/draw:  command
]

view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```

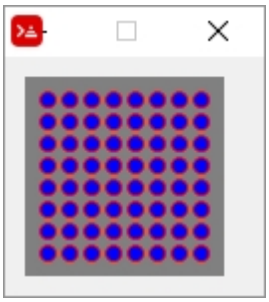


Note que se você fechar a janela gráfica e digitar `? canvas` no console, você verá um longo bloco como o valor de *draw*:

```
>> ? canvas
...
     draw     block!   length: 84  [circle 1x1 5 circle 2x2 5 circle 3x3
5 circle 4x4 5 ...
...
```

Um exemplo de programa desenhado:

```
Red [needs: view]

drawcircles: does [
    command: [pen red fill-pen blue]
    repeat x 8 [
        repeat y 8 [
            position:(x * 11x0) + (y * 0x11)
            append command reduce ['circle position 4]
        ]
    ]
    canvas/draw:  command
]

view [
    canvas: base 100x100
    do [drawcircles]
]
```



Você poderia ter escrito o programa acima sem usar uma função, mas você precisaria do refinamento `no-wait` para `view`, assim:

```
Red [needs: view]

command: [pen red fill-pen blue]

view/no-wait [
    canvas: base 100x100
]
{o refinamento "no-wait" acima permite
script criar a view (base) e, em seguida
continuar direto para o bloco do"repeat".
Sem "no-wait", o script permaneceria no
bloco do "view"}
```

```
repeat x 8 [
    repeat y 8 [
        position:(x * 11x0) + (y * 0x11)
        append command reduce ['circle position 4]
    ]
]

canvas/draw:  command
probe command {apenas para mostrar o que foi enviado para draw.
Você deve usar o probe em vez de print, porque print
tenta computar (avaliar), e "caneta" e "círculo" não têm
valor, o que gera um erro}
```

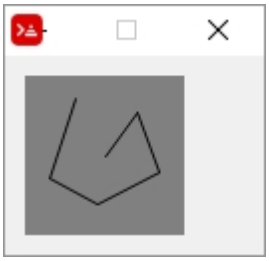
```
[pen red fill-pen blue circle 11x11 4 circle 11x22 4 circle 11x33 4
circle 11x44 4 circle 11x55 4 circle 11x66 4 circle 11x77 4 circle 11x88
4 circle 22x11 4 circle 22x22 4 circle 22x33 4 circle 22x44 4 circle
22x55 4 circle 22x66 4 circle ...
```

Você vê que Red atualiza automaticamente a `base` com os desenhos gerados pelo bloco do `draw` , mesmo depois que a *face* já foi criada por View. Isso acontece porque no Red, ao contrário do Rebol, o padrão é que sempre que você alterar algum campo do objeto *face*, esta é atualizada automaticamente. Isso não teria acontecido se você adicionasse a instrução `system/view/auto-sync?: off` no início do script, conforme descrito aqui .

O programa Paint mais simples do mundo:

```
Red [needs: view]
newposition: 40x40              ;desculpe, mas sempre começa no centro.
linedraw: func [offset] [       ;func, não function. Variáveis são
globais.
    oldposition: newposition
    newposition: offset
      ; agora vamos adicionando linhas ao bloco de draw:
    append canvas/draw reduce['line oldposition newposition]
]

view [
canvas: base draw[]             ;crea um campo draw no objeto.
    on-down [                   ;quando um botão é clicado...
        do [linedraw event/offset]  ;envia a posição do mouse.
    ]
]
```

Toda vez que você clica no mouse na `base`, um novo segmento de linha é desenhado:



Aqui está uma versão muito melhorada do script que, no entanto, não usa a estrutura "regra geral":

```
Red [needs: view]
EnableWrite: false
view [
    canvas: base 150x150 white all-over
    draw[]
    on-down [                       ;quando clica o mouse...
        EnableWrite: true           ;... habilita desenho...
        startpoint: event/offset    ;...e pega a posição do cursor
    ]
    on-up [EnableWrite: false]      ;quando o botão é solto, desabilita
o desenho
    on-over [                       ;quando o cursor está sobre a
base...
        if EnableWrite [
            endpoint: event/offset  ;pega a posição atual
                ; agora vai adicionando linhas...
            append canvas/draw reduce['line startpoint endpoint]
            startpoint: endpoint
        ]
    ]
]
```

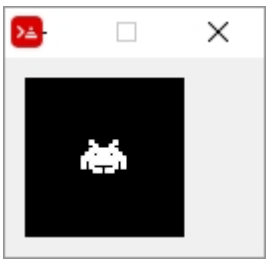
Observe que o *flag* `all-over` permite que o evento `over` crie eventos para cada movimento do mouse, conforme explicado aqui .

Movimentando um *shape* com as setas do teclado

Esse script desenha um "alien" no centro de uma `base` e permite que as teclas de seta movam a *shape* para cima, para baixo, para a esquerda e para a direita. Ele usa a transformação `translate` para fazer o movimento. Observe o uso da **`compose`** para computar o que está entre parênteses.

```
Red [needs: view]
pos: 28x31          ; This is the initial position of the "alien"

{O seguinte bloco é apenas a forma de um "alien"}
```

```
alien: [line 4x0 4x2
 'hline 2 'vline 2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -2 'vline 6
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline 2 'hline 4 'vline -2
 'hline -4 'vline -2 'hline 10 'vline 2
 'hline -4 'vline 2 'hline 4 'vline -2
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline -6 'hline -2  'vline -2
 'hline -2 'vline -2 'hline -2 'vline -2
 'hline 2 'vline -2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -6 'vline -2
 'hline -2 'vline -2 'hline -2
 move 6x6 'hline 2 'vline 2 'hline -2 'vline -2
 move 14x6 'hline 2 'vline 2 'hline -2 'vline -2
]

{A próxima função atualiza o bloco 'draw' do objeto cosmos.
Remove a palavra 'translate e o par seguinte!
a partir do início do bloco e, em seguida, insere 'translate
 novamente e o par da posição atualizado!}

update-cosmos: func [] [
      remove/part cosmos/draw 2
      insert cosmos/draw reduce ['translate pos]
]

view compose/deep/only [
      cosmos: base black focus ; use focus para ter evento on-key
      draw [
            translate (pos)    ; translação inicial. Centraliza o
"alien"
            pen white
            fill-pen white
          shape (alien)
      ]

      on-key  [
            switch event/key [
                  up    [pos: pos - 0x1]  ; decreases y axis
                  down  [pos: pos + 0x1]  ; increases y axis
                  left  [pos: pos - 1x0]  ; decreases x axis
                  right [pos: pos + 1x0]  ; increases x axis
            ]
            update-cosmos
      ]
]
```



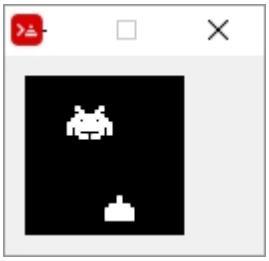
Sugiro você alterar o program para testar a transformação `rotate` .

Movendo duas ou mais *shapes* separadamente

O script a seguir usa a seta para a esquerda e para a direita para mover a "nave espacial", e "z" e "x" para mover o "alien".Observe o escopo de `reduce` e `compose` :

```
Red [needs: view]
;======= initial positions: ========
alienposition: 28x15
shipposition: 32x60

;======= just the shapes ===========
alien: [line 4x0 4x2
 'hline 2 'vline 2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -2 'vline 6
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline 2 'hline 4 'vline -2
 'hline -4 'vline -2 'hline 10 'vline 2
 'hline -4 'vline 2 'hline 4 'vline -2
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline -6 'hline -2  'vline -2
 'hline -2 'vline -2 'hline -2 'vline -2
 'hline 2 'vline -2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -6 'vline -2
 'hline -2 'vline -2 'hline -2
 move 6x6 'hline 2 'vline 2 'hline -2 'vline -2
 move 14x6 'hline 2 'vline 2 'hline -2 'vline -2
]
spaceship: [move 0x12 'hline 14 'vline -6
'hline -2 'vline -2 'hline -4 'vline -4 'hline -2
'vline 4  'hline -4 'vline 2 'hline -2 'vline 6
]
;======= The draw block updating function ======
; this time we create the whole block and just replace
; the original cosmos/draw
update-cosmos: does[
    drawblock: reduce compose/deep[
        'pen white
        'fill-pen white
        'translate alienposition [shape [(alien)]]
        'translate shipposition [shape [(spaceship)]]
    ]
    ;probe drawblock ;uncomment if you want to see it
    cosmos/draw: drawblock
]

view compose/deep/only [
    cosmos: base black focus
;this "draw" be "executed" only once:
    draw [
        pen white
        fill-pen white
        translate (alienposition) [shape (alien)]
        translate (shipposition) [shape (spaceship)]
    ]
; now the draw block will be recreated on every key press
    on-key  [
            switch event/key [
                  #"z"  [alienposition: alienposition - 1x0]      ;
decreases x axis
                  #"x"  [alienposition: alienposition + 1x0]      ;
increases x axis
                  left [shipposition: shipposition - 1x0]       ;
decreases x axis
```

```
                right [shipposition: shipposition + 1x0]  ; increases
x axis
            ]
            update-cosmos ; calls the "draw block recreating function"
        ]
]
```



O que existe em "system"

Se você digitar `? system` no console, você obtém:

```
>> ? system
SYSTEM is an object! with the following words and values:
     version     tuple!        0.6.3
     build       object!       [date git config]
     words       object!       [datatype! unset! none! logic!...
     platform    function!     Return a word identifying the operating
system.
     catalog     object!       [datatypes actions natives accessors
errors]
     state       object!       [interpreted? last-error trace]
     modules     block!        length: 0  []
     codecs      block!        length: 8  [png make object! [title:...
     schemes     object!       []
     ports       object!       []
     locale      object!       [language language* locale locale* months
days]
     options     object!       [boot home path script cache thru-cache
...
     script      object!       [title header parent path args]
     standard    object!       [header error file-info]
     lexer       object!       [pre-load throw-error make-hm make-msf...
     console     object!       [prompt result history size running?
catch? ...
     view        object!       [screens event-port metrics fonts
platform ...
     reactivity  object!       [relations stack queue eat-events? debug?
...
```

Você pode explorar esses *paths* usando tanto `?` como `probe`.

Coisas interessantes que você pode fazer:

Acessar *words*, não só as do sistema, mas as suas próprias.

Se você digitar `? system/words`, você obtém um lista muito, muito longa de todas as palavras da sua sessão Red:

```
>> ? system/words
```

```
SYSTEM/WORDS is an object! with the following words and values:
     datatype!                datatype!     datatype!
     unset!                   datatype!     unset!
     none!                    datatype!     none!
   ...
   ...
     right-command            unset!
     caps-lock                unset!
     num-lock                 unset!
```

Digite uma nova palavra no seu console, como `banana` , aperte enter (dá um erro) então digite `? system/words` novamente. Você verá que `banana` foi adicionada à lista de palavras de sua sessão:

```
>> banana
*** Script Error: banana has no value
*** Where: catch
*** Stack:

>> ? system/words
SYSTEM/WORDS is an object! with the following words and values:
     datatype!                datatype!     datatype!
     unset!                   datatype!     unset!
   ...
   ...
     caps-lock                unset!
     num-lock                 unset!
     banana                   unset!
```

Se você atribuir um valor a `banana` e repetir `? system/words` você verá que o valor foi vinculado à palavra (word):

```
>> banana: "hello"
   ...
   ...
     caps-lock                unset!
     num-lock                 unset!
     banana                   string!       "Hello"
```

Mudar o *prompt* do console:

```
>> ? system/console/prompt
SYSTEM/CONSOLE/PROMPT is a string! value: ">> "

>> system/console/prompt: "@*=> "
== "@*=> "
@*=>                    ;este é o novo prompt agora
```

Ver o histórico dos comandos:

```
>> probe system/console/history
["probe system/console/history" "?
system/console" {system/console/prompt: "@*=> "} "
" {system/console/prompt: "@*"} "? system/console/prompt" "?
console/prompt" "? system" "? system/standard/error" "? system" "probe
last system/word" "probe last system" "probe last a" "a: [a b c]" "probe
last sys ...
```

Alterar as mensagens de erro:

```
>> ? system/catalog/errors/script
SYSTEM/CATALOG/ERRORS/SCRIPT is an object! with the following words and
values:
     code               integer!      300
     type               string!       "Script Error"
     no-value           block!        length: 2  [:arg1 "has no value"]
    ...
     lib-invalid-arg    block!        length: 2  ["LIBRED - invalid
argument for" :arg1]

>> system/catalog/errors/script/type: "Don't be silly!! "
== "Don't be silly!! "

>> nono
*** Don't be silly!! : nono has no value
*** Where: catch
*** Stack:
```

Escolher caminhos conforme o Sistema Operacional:

```
>> either system/platform = 'Windows [print "Do this"] [print "Do that"]
Do this
```

Note o apóstrofo antes de "Windows". É uma `word!` não uma `string!`

Obter o tamanho da tela:

```
>> print system/view/screens/1/size
1366x768
```

Debugar View:

Use `system/view/debug?: yes` , como explicado em GUI Tópicos Avançados .

Apêndice I - Enquanto esperamos pela porta serial...

(capítulo temporário)

Aviso 1: Esta informação é principalmente para usuários do Windows;

Aviso 2: A comunicação serial pode ser complicada, com caracteres ocultos e detalhes de configuração. Se você não estiver familiarizado com o assunto, sugiro que você comece com um tutorial mais amigável.

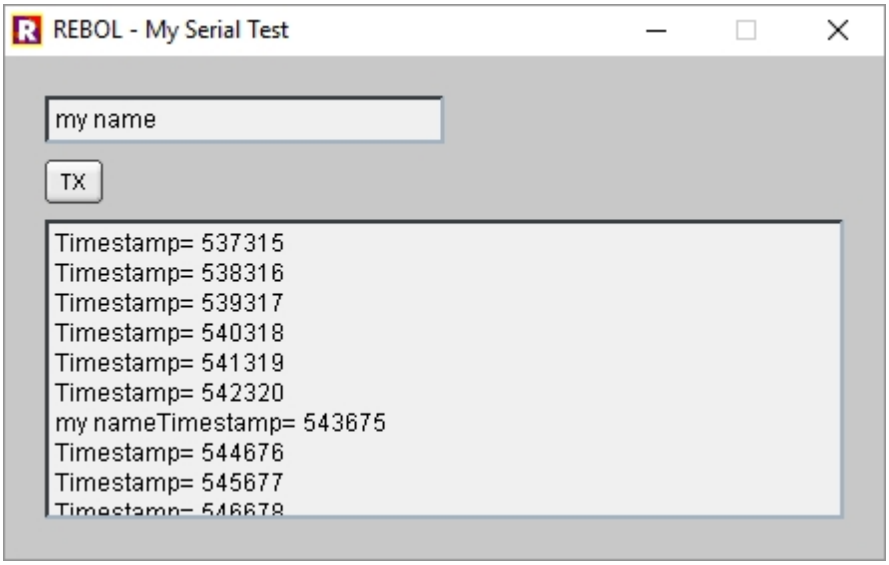
O Red ainda não suporta o acesso à porta serial (outubro de 2018). Isso pode ser decepcionante se você planeja usar o Red com Arduino, IoT, ESP8266 e hardware em geral. Assim, enquanto esperamos pelo suporte da porta serial, descrevo aqui alguns truques e dicas que achei úteis. Eles estão relacionados principalmente ao envio de comandos para o cmd do Windows usando `call` , mas os usuários do Linux também podem encontrar informações interessantes aqui.

Como funciona em Rebol. Red deverá ser parecido:

Veja a doucumentação do Rebol;

Parece-me que em Rebol você tem que dizer qual é a sua porta COM, criar uma "coisa serial" (chamada "ser" no exemplo abaixo) e configurá-la. Então, para enviar mensagens para serial, você `insert` suas mensagens nessa "coisa", e para ler o que é recebido, você `copy` , `pick` ou `first` essa "coisa".

```
Rebol []

System/ports/serial: [ com7 ]
ser: open/direct/no-wait serial://port1/9600/none/8/1
ser/rts-cts: false

view/title layout [
      f: field 200
      btn "TX" [insert ser f/text update ser]
      t: area
      rate 20 feel[engage: [append t/text copy ser show t]]
] "My Serial Test"
```



Neste exemplo, o que é enviado pelo dispositivo é mostrado na `area` e, quando você pressiona TX, o que quer que tenha escrito no `field` será enviado para o dispositivo.

Eu testei com um programa ESP8266 que envia um timestamp a cada segundo, mas também transmite de volta o que recebe. O programa também envia um ctrl-z (0x1A) a cada 10 mensagens. Caso você esteja interessado, aqui está o *sketch* do Arduino:

```
long interval = 1000;     //milliseconds between sending timestamps
long previousMillis = 0;
long count = 0;
void setup(){
  Serial.begin(9600);
}

void loop()
{ // this first part "echoes" whatever is sent
  // when characters arrive over the serial port...
  if (Serial.available()) {
    // ...wait a second and send them back.
    delay(1000);
    while (Serial.available() > 0) {
    Serial.write(Serial.read());
    }
  }

  // this second part sends a timestamp every interval
  long currentMillis = millis();
  if(currentMillis - previousMillis > interval) {
      if (count > 10){
        count =  0;
        Serial.print("stop\x1A"); // string "stop" & ctrl-z
      }
    previousMillis = currentMillis;
    Serial.print("Timestamp= ");
    Serial.println(currentMillis);
    count = count +1;
  }
}
```

E agora, algumas dicas para usar o Red como está...

Uma função para obter as portas COM disponíveis::

Envia o commando `mode` para o cmd e faz o parsing (sem usar `parse`) do valor retornado:

```
Red []
funcGetComPorts:
; Usa o  cmd do Windows para obter as COM ports disponíveis
      has[cmdOutput com-ports b c i] [

      cmdOutput: ""     ;aqui fica a resposta do cmd, como texto
      com-ports: []   ;esta série vai ter as COM ports
; agora mandamos o comando "mode" para o cmd
; guardamos a resposta do sistema em "cmdOutput"
      call/output "mode" cmdOutput
; retiramos todos os "-"
      trim/with cmdOutput "-"
; dividimos o cmdOutput em uma série
      cmdOutput: split cmdOutput " "
; fazemos um pouco de edição para ter um bom formato
      foreach i cmdOutput [
            b: copy/part i 3
            if b = "COM" [
                  c: copy/part i 4
                  append com-ports c
            ]
      ]
      return com-ports
]

probe funcGetComPorts
```

```
["COM7" "COM3"]
```

Configurando uma porta serial:

O comando completo do cmd para configurar uma porta COM seria:

```
mode COM7 BAUD=9600 PARITY=n DATA=8
```

Então, essa seria uma função de configuração de porta COM:

```
Red []
SerialConfig: function [COMport baud parity datasize][
      command: ""
      command: rejoin [command "mode " COMport " BAUD=" baud
                        " PARITY=" parity " DATA=" datasize]
      print command
      call/shell form command
]
SerialConfig "COM7" "9600" "n" "8"
```

Você pode verificar se funciona digitando `mode` cmd antes e depois de executar o script acima. `mode` mostra a configuração atual de suas portas.

Using ComPrinter.exe and SerialSend.exe :

Esses pequenos programas (disponíveis para download aqui) podem ser acessados usando um comando `call` dentro de um script Red para enviar e receber dados da porta serial. Eles são programas *open source* feitos por Ted Burke (obrigado!). Eles são ótimos programas que, com um pouco de criatividade, podem permitir fazer muitas coisas com Red!

Os exemplos dados aqui assumem que esses executáveis estão na mesma pasta que o script. Simplesmente copie e cole lá.

ComPrinter *

*procure pela versão atualizada que você encontra nos comentários da página do ComPrinter (link de download direto).

Da página da Web: "O ComPrinter é um aplicativo de console (ou seja, um programa de linha de comando) que abre uma porta serial e exibe caracteres de texto recebidos no console. Ele apresenta várias opções muito úteis."

Opções de ComPrinter.exe:

/devnum - Use para especificar uma porta COM. O padrão é a porta de número mais alto, incluindo portas> = 10. Por exemplo, para definir COM7 use `/devnum 7`

/baudrate - Use para especificar a taxa de transmissão. O padrão é 2400 bits por segundo. Por exemplo, para definir a taxa de transmissão para 9600, use `/baudrate 9600`

/keystrokes - Use para simular um pressionamento de tecla para cada caractere de entrada, por exemplo, para digitar texto em um aplicativo..

/debug - Use para exibir informações adicionais ao abrir a porta COM..

/quiet - Use para suprimir o texto da mensagem de boas vindas e outras informações. Apenas texto recebido pela porta COM será exibido.

As seguintes opções estão disponíveis apenas na versão atualizada:

/charcount - Para sair de um certo número de caracteres. Por exemplo, para sair após 6 caracteres, use `/charcount 6`

/timeout - Sair após um tempo limite - ou seja, nenhum dado recebido por um número especificado de milissegundos. Por exemplo, para sair após 2 segundos sem dados, use `/timeout 2000`

/endchar - Sair quando um determinado personagem é recebido. Por exemplo, para sair quando a letra 'x' for recebida, use `/endchar x`

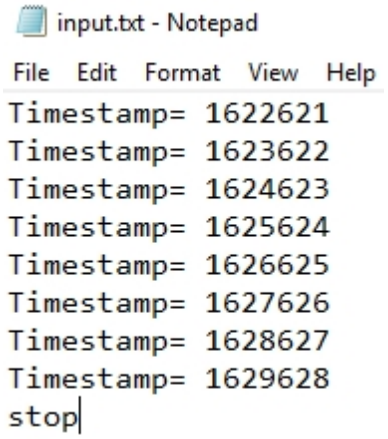
/endhex - Sair quando um determinado byte hexadecimal for recebido. Por exemplo, para sair quando o valor hexadecimal 0xFF for recebido, use `/endhex FF`

Exemplo:

O exemplo abaixo envia o que recebe em COM7 em 9600 baud para o arquivo "input.txt" até que ele receba um ctrl-z. Cria o arquivo automaticamente ou faz *append* de um arquivo existente. O esboço do Arduino acima envia um ctrl-z só de vez em quando, então o arquivo que você vai gerar pode ser maior ou menor:

```
Red[]

call/output form "ComPrinter.exe /devnum 7 /baudrate 9600 /endhex 1A"
%"input.txt"
;     ComPrinter.exe     - o executável chamado
;     /devnum 7          - seleciona COM7
;     /baudrate 9600     - seleciona baud rate 9600
;     /endhex 1A         - pára o ComPrinter quando recebe um ctrl-z
(0x1A)
;     %"input.txt"       - o arquivo de output (lembra do
refinamento /output ?)
```

Conteúdo do arquivo input.txt após rodar o script:



Caso você queira que seu script Red faça outra coisa enquanto cmd lê a porta serial, você pode usar um redirecionamento cmd (">") para enviar a saída para um arquivo. Nesse caso, parece funcionar apenas com `call/shell`:

```
Red[]

call/shell form "ComPrinter.exe /devnum 7 /baudrate 9600 /endhex 1A >
input.txt"
print "This is printed immediately, while the input.txt file is still
being created"
```

Infelizmente, você não pode escrever na porta serial enquanto o cmd estiver recebendo dados. E, a propósito, o Windows leva alguns segundos para atualizar o arquivo, então se você abrir "input.txt" muito rapidamente, pode estar vazio. Claro, também pode estar vazio porque algo deu errado ...

SerialSend

Da página web: "SerialSend é um pequeno aplicativo de linha de comando que criei para enviar strings de texto através de uma porta serial. Eu o uso principalmente para enviar informações para circuitos de microcontroladores através de um conversor USB para serial, então ele foi projetado para funcionar bem nesse contexto "

O seguinte comando envia os caracteres “abc 123” através da porta serial mais alta disponível na taxa de transmissão padrão (38400 baud).

```
SerialSend.exe "abc 123"
```

Opções par SerialSend.exe:

/devnum - Use para especificar uma porta COM. O padrão é a porta com maior disponibilidade, incluindo portas> = 10. Por exemplo, para definir COM7 use `/devnum 7`

/baudrate - Use para especificar a taxa de transmissão. O padrão é 38400 bits por segundo. Por exemplo, para definir a taxa de transmissão para 9600, use `/baudrate 9600`

/hex - Bytes arbitrários, incluindo caracteres não imprimíveis, podem ser incluídos na string como valores hexadecimais usando a opção de linha de comando “/ hex” e a seqüência de escape “\ x” no texto especificado. Por exemplo, o comando a seguir envia a string “abc” seguida por um caractere de alimentação de linha (valor hexadecimal 0x0A) - ou seja, 4 bytes no total: `SerialSend.exe /hex "abc\x0A"`

Exemplo:

```
Red[]
call form {SerialSend.exe /devnum 7 /baudrate 9600 "abc 123"}
```

Exemplo que envia variáveis e funções:

```
Red[]
myVariable: "Time now is: "   ; a string
txt: rejoin [{"} myVariable now {"}] ; now returns time and date
command: form rejoin ["SerialSend.exe /devnum 7 /baudrate 115200 " txt]
print command     ;just to help you see what will be sent to cmd
call command
```

Note que eu aumentei o baudrate para 115200 neste segundo exemplo. Isso porque eu estava tendo problemas em 9600 baud: por algum motivo, a mensagem estava sendo truncada para cerca de uma dúzia de caracteres. Depois de muitas horas tentando isolar o bug (um *null modem cable* teria ajudado, mas não tenho um no momento), desisti e aumentei a velocidade, tanto no script Red quanto no sketch do Arduino. Isso não resolveu completamente, mas eu consigo enviar strings com mais de 200 caracteres, o que é bom o suficiente por enquanto.

Um utilitário semelhante ao SerialSend e ComPrinter, baseado no trabalho de Ted Burke, é o comsniff - Este utilitário não apenas imprime o que recebe no console do cmd, mas também envia o que você digita, conforme você digita, para a porta serial. Eu tive muitos problemas tentando usá-lo, mas é open source e vale a pena mencionar aqui.

Outras informações úteis (?) Caso você não queira usar executáveis externos:

Enviando caracteres para uma porta COM: (não extensivamente testado)

Eu encontrei informações úteis sobre o envio de caracteres para a porta serial no Windows aqui. asicamente, você pode enviar dados para a porta serial usando:

- `echo hello > COM1`

Mas este comando também envia um espaço extra, um CR e um LF. Além disso, não reconhece números de porta mais altos (acima de 9?). Você pode optar por enviar um comando mais universal como este:

- `set /p x="hello" <nul >\\.\COM22`

Aqui está uma função que usa o primeiro comando:

```
Red []
SerialSender: function [stringtosend COMport][
	command: []
	append command "e "
	append command stringtosend
	append command " > "
	append command COMport
	call/shell form command
]

SerialSender "hello world" "COM7"
```

Você pode enviar arquivos inteiros para a porta serial usando `copy yourfile.txt com1`, ou, para portas de número >= 10, `copy yourfile.txt \\.\COM21`

(Supostamente deveria) redirecionar o input serial para um arquivo: (bem, testado mas...)

Estes comandos deveriam enviar os dados recebidos da porta serial para um arquivo:

- `COPY COM4 data.txt`
- `type com1: >> data.txt`

Eu tive resultados muito ruins com isso. O cmd do Windows parece começar a ler quando quer e isso pode levar dezenas de segundos, até minutos, ou nunca. De qualquer forma, se você for corajoso, não se esqueça de combinar a taxa de transmissão, a paridade e o tamanho dos dados primeiro!

A propósito, para fazer o cmd parar de gravar dados, o dispositivo tem que enviar um ctrl-z. Você consegue isso no Arduino usando `Serial.write ("26")` ou `Serial.print("<Stuff>\x1A")` . Isso parece funcionar com `copy` (quando `copy` funciona) mas não com `type`.

Terminais:

Terminal - com port development tool - Muito bom e completo, mas tem que se acostumar um pouco.

PuTTY pode ser configurado para funcionar como um terminal serial. Permite salvar a sessão em um arquivo de log.

Para ser honesto, eu uso quase sempre o Serial Monitor da IDE do Arduino.

Apêndice II -CGI e RSP usando o servidor Cheyenne

O Red não tem suporte completo ao CGI (novembro de 2018). Os primeiros capítulos aqui cobrem os passos básicos usando o Rebol. Eu acredito que o comportamento do Red será muito similar, se não o mesmo. Isso não significa que você não pode usar Red para CGI. Você pode encontrar uma boa referência de como usá-lo aqui

Há muitas informações sobre CGI na Internet. No entanto, eu tive dificuldade com os primeiros passos, especialmente como usar o servidor Cheyenne no meu próprio computador, como "cobaia" para meus testes. Então eu escrevi este texto como um "guia para iniciantes". Não é um tutorial completo sobre CGI e RSP.

O que é CGI?

Common Gateway Interface (CGI), é um protocolo que permite aos servidores executar programas que geram páginas web dinamicamente, isto é: programas que geram código HTML em tempo real, "adaptados" à entrada do usuário.

O CGI foi substituído por uma grande variedade de tecnologias de programação web. Atualmente, a maioria dos desenvolvedores usa linguagens de script como o PHP para fazer o que o CGI faz.

Então por que você deveria se interessar? Bem, talvez você não queira ser um desenvolvedor da Web, apenas conectar seus scripts Red / Rebol a navegadores da Web, criar alguns aplicativos da Web, ou qualquer coisa assim. Os navegadores da Web são uma ótima maneira de exibir informações e fazer interface com o usuário. E, claro, você também pode acessar a Internet.

O que é o RSP?

Eu posso estar errado sobre isso, mas acredito que o RSP é uma coisa só do Cheyenne. É uma maneira simplificada de fazer CGI, usando o Rebol embutido no código HTML. O que acontece é que Cheyenne tem um interpretador Rebol embutido em seu código, então, ao contrário do CGI normal, no qual tem que chamar algum interpretador de script (um executável) para rodar seu script e criar o HTML, RSP são arquivos interpretados por uma espécie de Rebol nativo em Cheyenne. Além disso, Cheyenne oferece APIs de RSP para trabalhar com seus scripts.

Por que Cheyenne?

Porque é incrivelmente pequeno, cerca de 500 KB! Tem um único arquivo de configuração e é totalmente portátil. Além disso, foi escrito em Rebol por Nenad Rakocevic e, como mencionado, interpreta essa linguagem nativamente. Você pode

facilmente empacotar tudo e seus scripts em um projeto e ainda assim ficar em menos de 1 MB.

Link básico de informações HTTP:

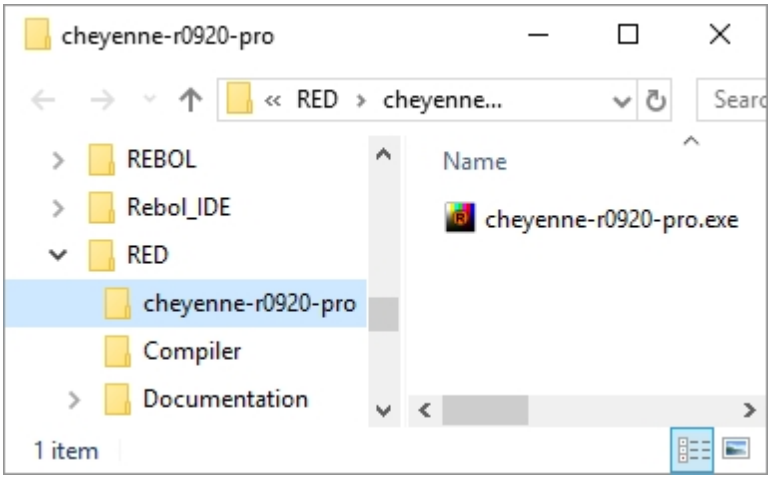
Um primer sobre HTTP - Muito bom, e tem links para informações mais detalhadas.

.

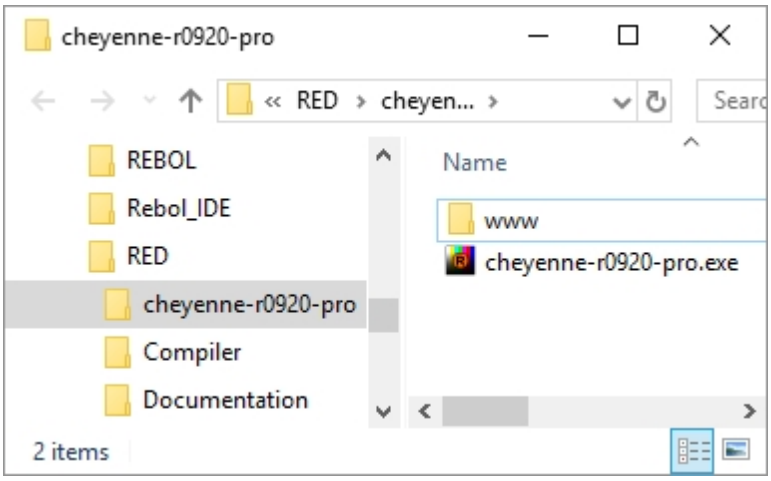
Instalando e configurando o Cheyenne

Vá para https://www.cheyenne-server.org/download.shtml e baixe o zip. Eu escolhi **Cheyenne Pro** porque é menor, mas você pode receber **Cheyenne Command** se você quiser alguns extras.

Descompacte-o em qualquer lugar no seu computador. Eu descompactei em uma pasta chamada RED, então eu tenho isso:



Agora crie uma pasta chamada "www" dentro da pasta do Cheyenne, assim:

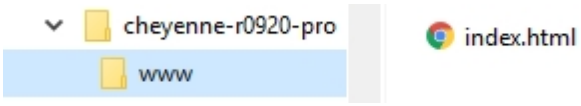


Agora copie o HTML abaixo em algum editor de texto e salve-o como index.html dentro da pasta www :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
```

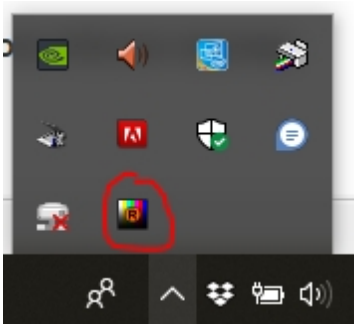
```
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title></title>
</head>
<body>
<h2 style="text-align: center;">Congratulations! Your
Cheyenne server is working!</h2>
<div style="text-align: center;">Have a nice day!</div>
</body>
</html>
```

Fica assim:

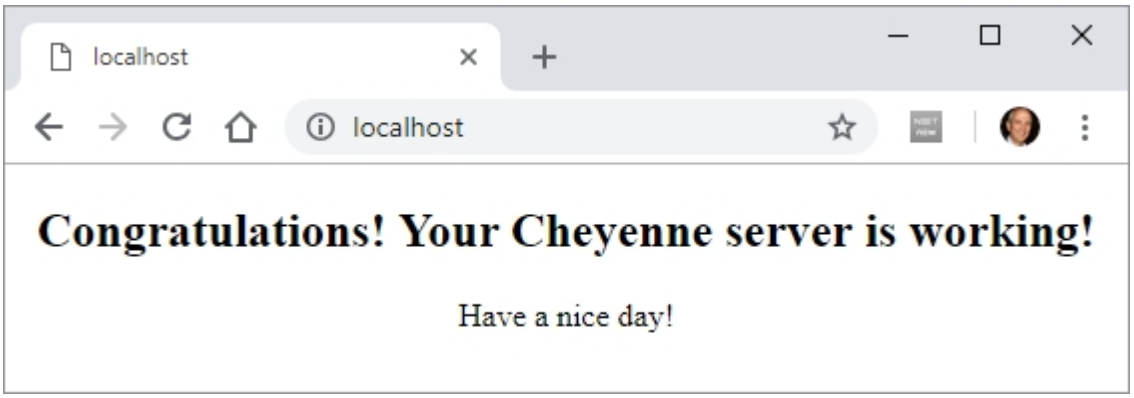


Agora dê um duplo clique no executável Cheyenne. Eu tive um par de avisos do Windows Defender e escolhi **mais informações / executar de qualquer maneira** .

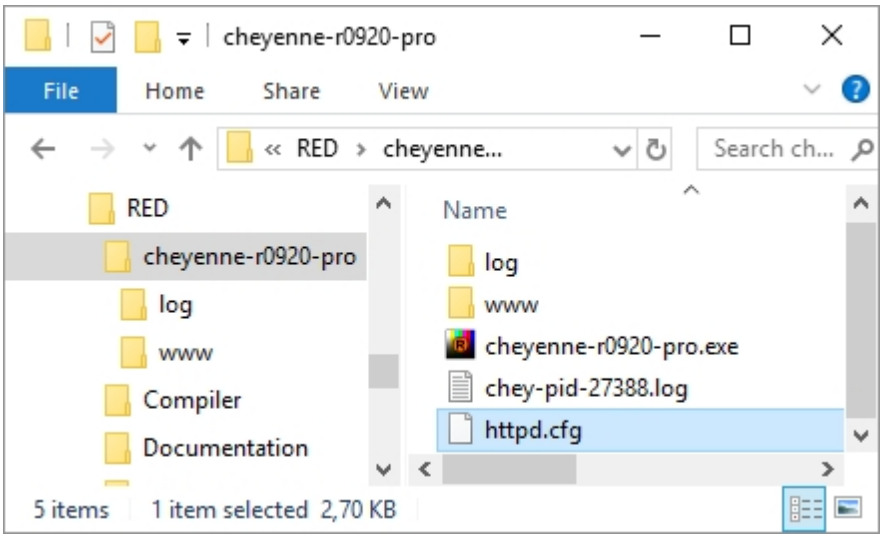
Na barra de tarefas, um pequeno ícone do Rebol avisa que Cheyenne está a funcionar:



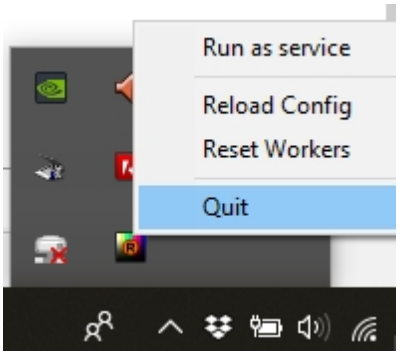
Agora abra seu navegador favorito, digite "localhost" na barra de endereços e pressione enter. O browser deve ir para a página html que você acabou de criar:



Após essa primeira execução, Cheyenne cria alguns arquivos e pastas extras e deve ficar assim:



Você pode sair do Cheyenne clicando com o botão direito do mouse no ícone da barra de tarefas e escolhendo **Quit** :



Portas são os "canais" da comunicação do computador. Por padrão, o Cheyenne escuta a porta 80, mas você pode querer mudar isso, seja para evitar conflitos ou, talvez (?), adicionar alguma segurança extra. Um número de porta é um inteiro de 16 bits, variando de 0 a 65535, mas sugiro que você escolha um número aleatório em torno de 30000.

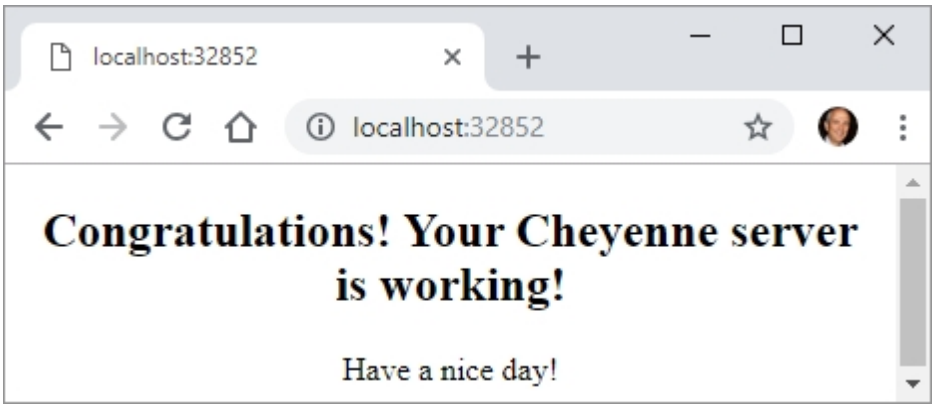
A propósito, usar o Cheyenne como descrito neste texto deve ser seguro, a menos que você abra explicitamente suas portas no modem DSL e no firewall no seu PC.

Para alterar a porta que o Cheyenne escuta para, por exemplo, 32852, abra o arquivo **httpd.cfg** com um editor de texto simples e adicione a seguinte linha:

```
...
;--- define alternative and/or multiple listen ports (by default, cheyenne will run on 80)

;listen [80 10443]

listen [32852]

bind SSI to [.shtml .shtm]

bind php-fcgi to [.php .php3 .php4]
...
```

Ou você pode apenas descomentar a linha de cima e mudar os números das portas (Cheyenne pode ouvir mais de uma porta).

Agora você pode acessar seu **index.html** digitando na barra de endereços do seu **host localhost: <número da porta>** :



Apenas lembrando que os números de porta comuns (evite-os) são:

20: Transferência de dados do protocolo de transferência de arquivos (FTP)
21: Controle de Comando do Protocolo de Transferência de Arquivo (FTP)
22: Login Seguro do Secure Shell (SSH)
23: serviço de login remoto Telnet, mensagens de texto não criptografadas
25: roteamento de email SMTP (Simple Mail Transfer Protocol)
53: serviço Sistema de Nomes de Domínio (DNS)
80: HTTP (Hypertext Transfer Protocol) usado na World Wide Web - *padrão Cheyenne*
110: protocolo postal (POP3)
119: Protocolo de transferência de notícias de rede (NNTP)
123: protocolo de tempo de rede (NTP)
143: IMAP (Internet Message Access Protocol) Gerenciamento de correio digital
161: Protocolo simples de gerenciamento de rede (SNMP)
194: Internet Relay Chat (IRC)
443: HTTP Seguro (HTTPS) HTTP sobre TLS / SSL

Se você removesse todas as linhas comentadas do arquivo **httpd.cfg** (não faça isso), você obteria o texto abaixo, que eu acho que é uma configuração simples e auto-explicativa:

```
modules [
        userdir
        internal
        extapp
        static
        upload
        action
        fastcgi
```

```
        rsp
        ssi
        alias
        socket
]

globals [
        bind SSI to [.shtml .shtm]
        bind php-fcgi to [.php .php3 .php4]
        bind-extern CGI to [.cgi]
        bind-extern RSP to [.j .rsp .r]
]

default [

        root-dir %www/
        default [%index.html %index.rsp %index.php]
        on-status-code [
                404     "/custom404.html"
        ]
        socket-app "/ws.rsp"        ws-test-app
        socket-app "/chat.rsp"      chat
        webapp [
                virtual-root "/testapp"
                root-dir %www/testapp/
                auth "/testapp/login.rsp"
        ]
]
```

RSP "Hello world"

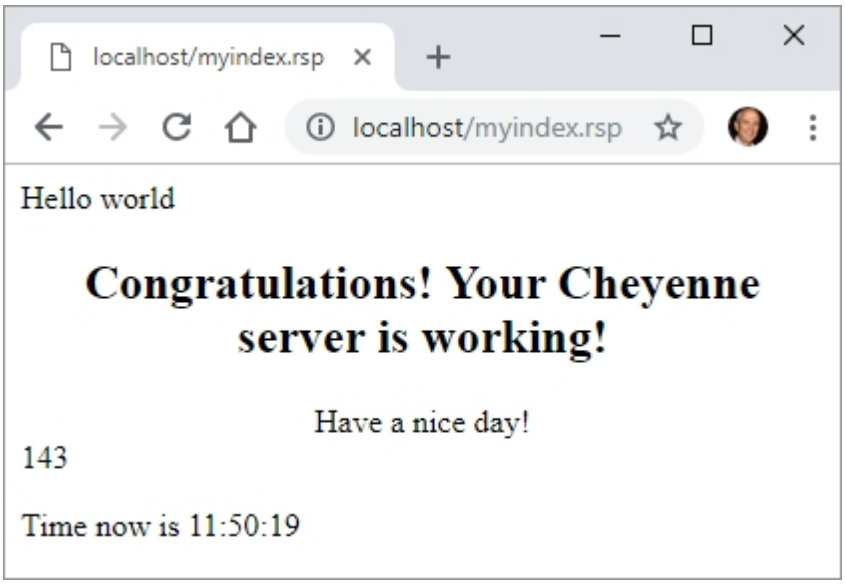
Veja também a página do Cheyenne's sobre RSP

Nos scripts RSP, o Cheyenne interpreta tudo que está entre "**<%**" e "**%>**" como código de Rebol !

Abra seu **index.html** (aquele que você criou no capítulo "Instalando e configurando ...") com um editor de texto simples, adicione as seguintes linhas destacadas e salve-o na pasta www como **myindex . rsp** .

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<% print "Hello world" %>
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title></title>
</head>
<body>
<h2 style="text-align: center;">Congratulations! Your
Cheyenne server is working!</h2>
<div style="text-align: center;">Have a nice day!</div>
<% print 55 + 88 %>
</br>
</body>
</html>
<% print rejoin ["Time now is " now/time] %>
```

Com o Cheyenne em execução (ouvindo a porta padrão 80), digite **localhost/myindex.rsp** na barra de endereços do seu navegador. Você deve ver isso:



RSP - Request e Response

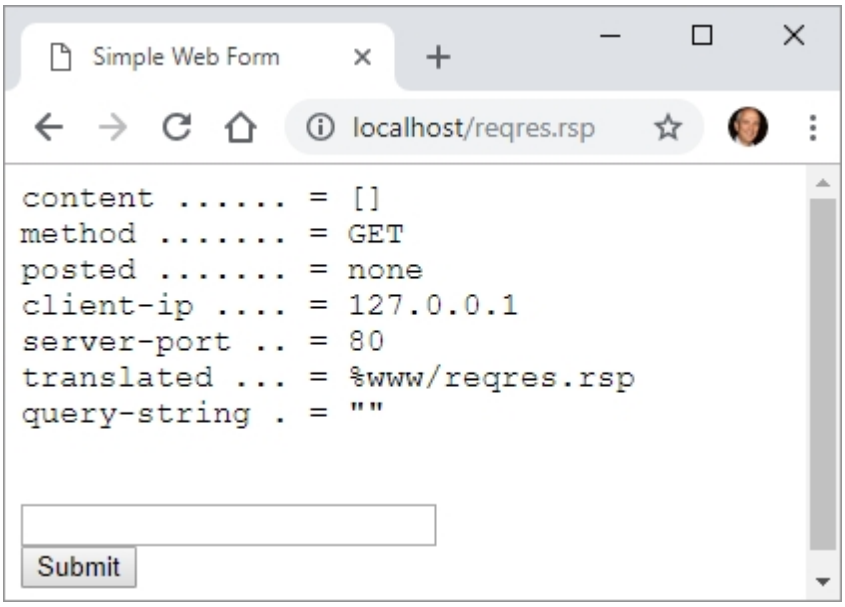
Use essa página como referência para este texto.

Requests:

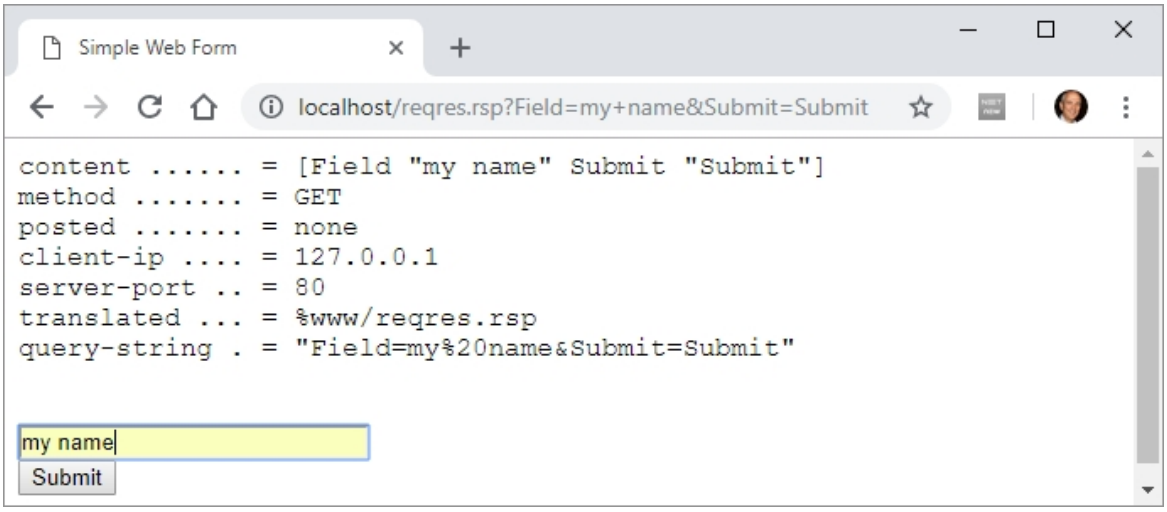
Use um editor de texto simples para criar o script abaixo e salve na pasta www como **reqres.rsp**.

```
<%
print {<font face="courier">}
print "content ...... = " probe request/content      print "<br>"
print "method ....... = " probe request/method       print "<br>"
print "posted ....... = " probe request/posted       print "<br>"
print "client-ip .... = " probe request/client-ip    print "<br>"
print "server-port .. = " probe request/server-port  print "<br>"
print "translated ... = " probe request/translated   print "<br>"
print "query-string . = " probe request/query-string print "<br>"
%>
<br><br>
<HTML>
<TITLE>Simple Web Form</TITLE>
<BODY>
<FORM ACTION="reqres.rsp">
<INPUT TYPE="TEXT" NAME="Field" SIZE="25"><BR>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

Com o Cheyenne rodando (porta 80), digite **localhost/reqres.rsp** na barra de endereços do navegador. Você deve ver isso:



Digite alguma coisa no campo e aperte o botão "Submit". Seu navegador deve mostrar isso:



O que acontece:

Parece claro que o Cheyenne pega o *request* do cliente (navegador), decodifica-o e armazena todos os valores importantes nas variáveis internas do objeto **request**.

Quando você clica em "Submit", `ACTION="reqres.rsp"` remete você para a mesma página, mas atualizada! Mas, para fazer isso, o navegador envia um **request** que é dividido e armazenado nas variáveis do objeto **request ,** que são mostradas na página atualizada.

Responses:

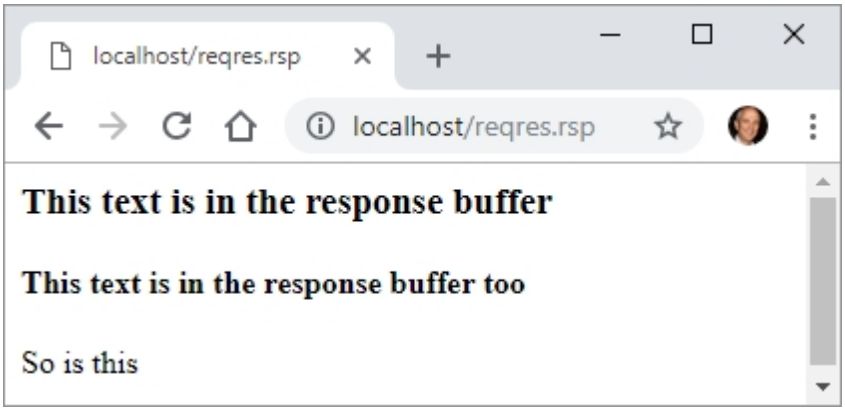
Da mesma forma que os requests têm o **objeto request**, as responses têm o **objeto**

response. No entanto, a maioria dos campos deste objeto são funções (ações). A exceção mais relevante é o **response/buffer**, que é onde o RSP do Cheyenne armazena tudo o que deve ser enviado ao cliente. É um bloco, e você pode manipulá-lo como qualquer série.

Se você mudar o código do **reqres.rsp** para:

```
<%
append response/buffer "<HTML>"
append response/buffer "<h3>This text is in the response buffer</h3>"
append response/buffer "<h4>This text is in the response buffer
too</h4>"
append response/buffer "<p>So is this</p>"
%>
```

Você obtém:

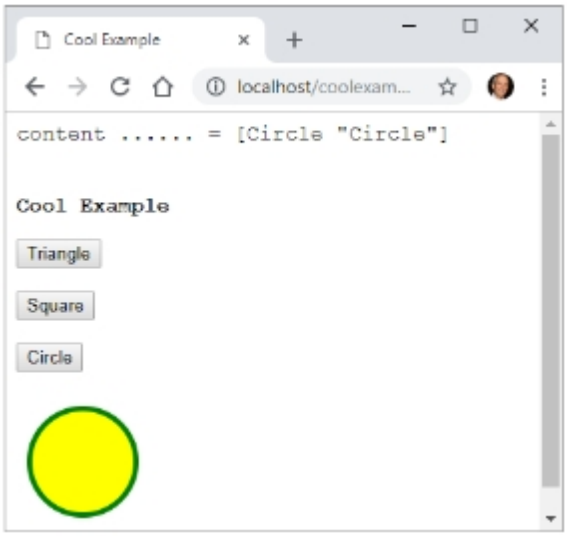


Exemplo legal:

Crie e salve o seguinte script RSP como **coolexample.rsp** na pasta www do Cheyenne . Abra **localhost/coolexample.rsp** no seu navegador e clique em um botão. Se o seu navegador suporta SVG HTML (a maioria suporta), uma imagem correspondente deve aparecer sob o botão.

```
<%
print {<font face="courier">}
print "content ..... = " probe request/content      print "<br>"

%>

<HTML>
<br><br>
<TITLE>Cool Example</TITLE>
<BODY>
<b>Cool Example</b><p>
<FORM ACTION="coolexample.rsp">
<INPUT TYPE="SUBMIT" NAME="Triangle" VALUE="Triangle"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Triangle  [
        print {<svg width="100" height="100">
        <polygon points="0,100 50,0 100,100"
        style="fill:lime;stroke:purple;stroke-width:5;fill-
rule:evenodd;" />
```

```
        </svg> <br>}
    ]
]
%>
<INPUT TYPE="SUBMIT" NAME="Square" VALUE="Square"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Square  [
        print {<svg width="100" height="100">
        <rect width="100" height="100" style="fill:rgb(0,0,255);stroke-
width:10;stroke:rgb(0,0,0)" />
        </svg> <br>}
    ]
]
%>
<INPUT TYPE="SUBMIT" NAME="Circle" VALUE="Circle"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Circle  [
        print {<svg width="100" height="100">
        <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4"
fill="yellow" />
        </svg> <br>}
    ]
]
%>
</FORM>
</BODY>
</HTML>
```



CGI "Hello world"

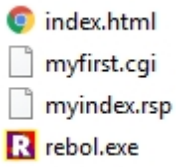
Veja também: Quick and Easy CGI - A Beginner's Tutorial and Guide

Faça o download do interpretador "rebol core" da página de downloads do Rebol . Salve esse executável na pasta **www** do seu Cheyenne.

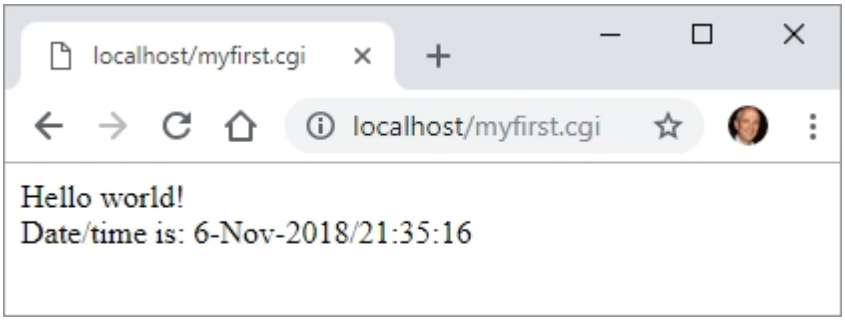
Agora crie o seguinte script em um editor de texto simples e salve-o como **myfirst.cgi** na mesma pasta **www.**

```
#!www/rebol.exe -c
REBOL []
print "Hello world!"
print "<br/>"
print ["Date/time is:" now]
```

Sua pasta **www** agora deve estar assim:



Agora, se o seu servidor está rodando (porta 80) e você digitar **localhost/myfirst.cgi** na barra de endereço do seu navegador, você obtém:



Explicando o script:

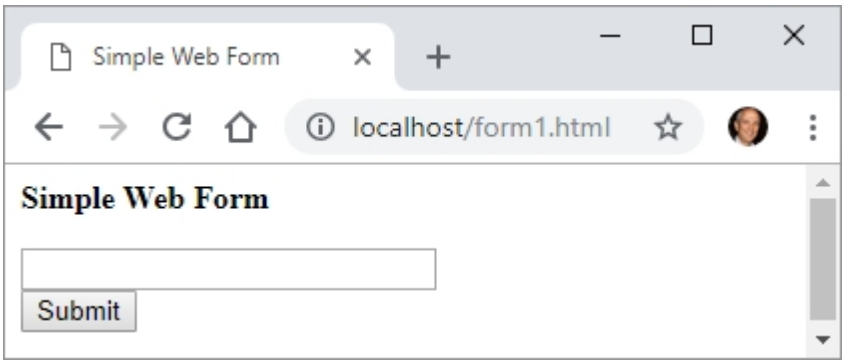
```
#!www/rebol.exe -c            ; Essa linha é importante
                              ; ela diz ao servido o
                              ; path do interpretador.
                              ; A opção -c diz ao Rebol
                              ; para funcionar no modo CGI.
REBOL []
print "Hello world!"          ; Manda "Hello world!" para o browser.
print "<br/>"                 ; carriage return.
print ["Date/time is:" now]   ; Manda data e hora.
```

CGI - Processing web forms

Veja também: Creating and Processing Web Forms with CGI (Tutorial)

Crie o seguinte arquivo **form1.html** na pasta **www**:

```
<HTML>
<TITLE>Simple Web Form</TITLE>
<BODY>
<b>Simple Web Form</b><p>
<FORM ACTION="formhandler.cgi">
<INPUT TYPE="TEXT" NAME="Field" SIZE="25"><BR>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```



Agora crie e salve na mesma pasta o script **formhandler.cgi**:

```
#!www/rebol.exe -c
Rebol []
print [<HTML><PRE> mold system/options/cgi </HTML>]
```

Quando você escreve "My Name" no campo e aperta o botão "Submit", o seu **form1.html** vai chamar o **formhandler.cgi**, e este script vai imprimir o que o protocolo CGI passa para o Rebol e este armazena em **system/options/cgi**:

```
make object! [
   server-software: "Cheyenne/1.0"
   server-name: "Ungaretti"
   gateway-interface: "CGI/1.1"
   server-protocol: "HTTP/1.1"
   server-port: "80"
   request-method: "GET"
   path-info: "/formhandler.cgi"
   path-translated: "www\formhandler.cgi"
   script-name: "/formhandler.cgi"
   query-string: "Field=My+Name&Submit=Submit"
```

```
    remote-host: none
    remote-addr: "127.0.0.1"
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: "0"
    other-headers:
["HTTP_ACCEPT" {text/html,application/xhtml+xml,application/...
]
```

É bom saber isso, mas o Rebol oferece uma função para decodificar o CGI, chamada `decode-cgi` que converte os dados brutos em um bloco de Rebol que contém palavras seguidas pelos seus valores. A informação que queremos (o conteúdo do campo) estão na variável **query-string**. Então, altere script **formhandler.cgi** como se segue:

```
#!www/rebol.exe -c
Rebol []
print [<HTML><PRE> decode-cgi system/options/cgi/query-string </HTML>]
```

O navegador agora vai mostrar :

```
 Field My Name Submit Submit
```

Exemplo legal de CGI

Esta é a versão CGI do exemplo legal de RSP. Salve como **coolexample.cgi** na pasta **www** do Cheyenne. Abra com o navegador usando *localhost/coolexample.cgi.*

```
#!www/rebol.exe -c
Rebol []
; First, a not very elegant way of avoiding crashes:
either system/options/cgi/query-string = none [
    system/options/cgi/query-string: ""
    decoded: ""
    ][
    decoded: second decode-cgi system/options/cgi/query-string
]

; Lets show what's in "decoded":
print {<font face="courier">}
print "decoded = " probe decoded      print "<br>"

; Here we start HTML
print {
    <HTML>
    <br><br>
    <TITLE>Cool Example</TITLE>
    <BODY>
    <b>Cool Example</b><p>
    <FORM ACTION="coolexample.cgi">}

print {<INPUT TYPE="SUBMIT" NAME="Triangle" VALUE="Triangle"><br><br>}
 if decoded = "Triangle"  [
     print {<svg width="120" height="120">
    <polygon points="0,100 50,0 100,100"
    style="fill:lime;stroke:purple;stroke-width:5;fill-rule:evenodd;" />
    </svg> <br>}
```

```
]


print {<INPUT TYPE="SUBMIT" NAME="Square" VALUE="Square"><br><br>}
if decoded = "Square"  [
    print {<svg width="120" height="120">
    <rect width="100" height="100" style="fill:rgb(0,0,255);stroke-
width:10;stroke:rgb(0,0,0)" />
    </svg> <br>}
]


print {<INPUT TYPE="SUBMIT" NAME="Circle" VALUE="Circle"><br><br>}
if decoded = "Circle"  [
    print {<svg width="120" height="120">
    <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4"
fill="yellow" />
    </svg> <br>}
]

; Now we finish HTML
print {
    </FORM>
    </BODY>
    </HTML>}
```

CGI usando Red

Hello World!

Veja também: *Using Red as CGI*

Faça uma cópia do interpretador Red e salve esse executável na pasta **www** do seu Cheyenne, assim como você fez com a Rebol.

Renomeie o executável do Red para algo como **redcgi.exe** . Eu descobri que isso é importante porque eu já tenho o Red "instalado" no meu computador (onde meu servidor está rodando - localhost), e o sistema operacional tenta apenas executar o script normalmente, não como CGI.

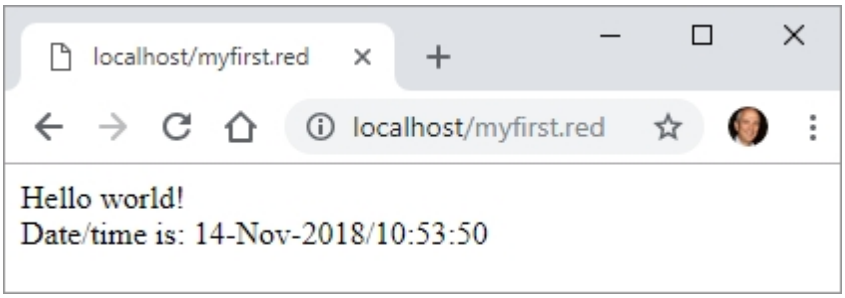
Abra o arquivo **httpd.cfg** em um editor de texto simples e inclua **.red** no bloco "bind-extern CGI to", conforme mostrado:

```
globals [
	;--- define alternative and/or multiple listen ports (by default, cheyenne
will run on 80)
	;listen [80 10443]
	bind SSI to [.shtml .shtm]
	bind php-fcgi to [.php .php3 .php4]
	bind-extern CGI to [.cgi .red]
	bind-extern RSP to [.j .rsp .r]
```

Agora crie o seguinte script em um editor de texto simples e salve-o como **myfirst.red** na mesma pasta **www** . --cli é importante, se não usá-lo, o Red pode tentar compilar e abrir o console GUI.

```
#!www/redcgi.exe --cli
Red []
print "Hello world!"
print "<br/>"
print ["Date/time is:" now]
```

Agora, se o seu servidor está em execução (porta 80) e você digitar **localhost/myfirst.red** na barra de endereços do seu navegador, você obtém:



Processando web forms.

Como mencionado, o Red ainda não tem suporte completo para o CGI. No entanto, acredito que seja possível recuperar e decodificar mensagens HTTP no **Linux** , usando o http-tools.red de Boleslav Březovský . Eu não sei como fazer isso no Windows.

Apêndice III -MQTT usando Red

O MQTT se tornou protocolo mais popular para comunicação de IoT (Internet of Things). No *Internet Protocol Stack*, ela funciona na mesma camada que o HTTP, mas o MQTT é mais leve, usa menos banda e permite manter uma conexão fixa com dispositivos e comunicação quase em tempo real.

Ao contrário do suporte para porta serial ou CGI, o MQTT não é uma prioridade no desenvolvimento do Red, e dependerá da comunidade para criar bibliotecas nativas. No entanto, é possível publicar e subscrever tópicos (como cliente) usando Red e alguns executáveis e DLLs externos.

Não vou entrar em detalhes sobre o MQTT, presumo que você saiba o básico disso. Caso você não saiba , a melhor informação que encontrei está nos tutoriais tutoriais do Hivemq .

Para monitorar mensagens MQTT, você pode usar qualquer uma das ferramentas listadas aqui . Eu uso o MQTT-spy, mas qualquer utilitário do cliente serve, incluindo alguns aplicativos Android que você pode instalar no seu telefone (pesquise o Google-Play).

Eu usei uma conta gratuita "Cute cat" no CloudMQTT para meus testes.

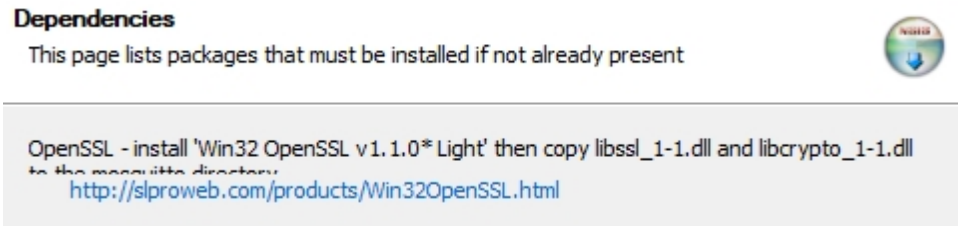
O que você precisa:

Você precisa ter os seguintes arquivos dentro da pasta do seu script::

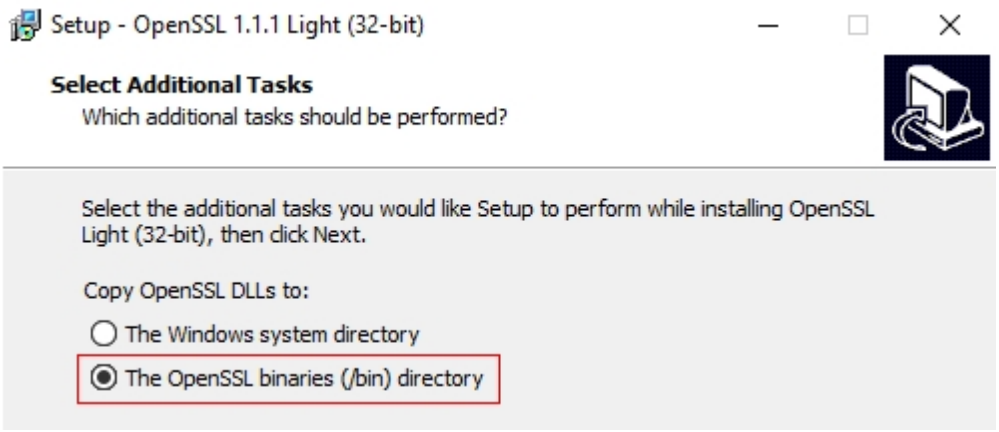
- mosquitto_pub.exe
- mosquitto_sub.exe
- mosquitto.dll
- libssl-1_1.dll
- libcrypto-1_1.dll

Eu obtive **mosquitto_pub.exe, mosquitto_sub.exe** e **mosquitto.dll** instalando o mosquitto obtido aqui. Eu usei a instalação 32 bits. Esses arquivos estão na pasta "mosquitto" criada na instalação.

Durante a instalação, você recebe o seguinte aviso:



Os arquivos **libssl-1_1.dll** e **libcrypto-1_1.dll** pertencem ao OpenSSL toolkit. Assim, conforme recomendado, eu baixei o OpenSSL de http://slproweb.com/products/Win32OpenSSL.html e instalei. Durante a instalação, você deve optar por colocar as DLLs na pasta do OpeSSL, vai ser mais fácil encontrá-las depois:



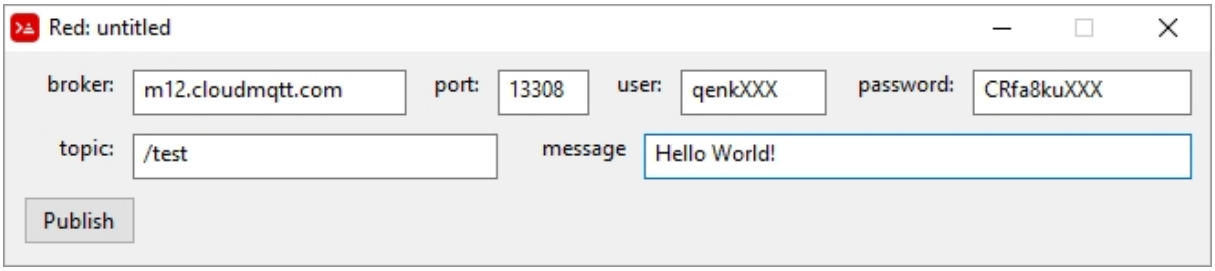
Então copiei **libssl-1_1.dll** e **libcrypto-1_1.dll** não apenas para o diretório do mosquitto, mas também para a pasta do meu script.

Para entender o uso do **mosquitto_pub.exe** olhe esta página, e para o **mosquitto_sub.exe** existe esta outra página. Uma boa página com exemplos é Using The Mosquitto_pub and Mosquitto_sub MQTT Client Tools- Examples, e o respectivo video.

Publicando:

O script a seguir é um simples "publicador" de MQTT. Não oferece muitas opções, mas é suficiente mostrar como criar uma linha de comando para o mosqutto_pub :

```
Red [needs view]
view [
    text "broker:" 50 right broker: field "m12.cloudmqtt.com" 150
    text "port:" 30 right port: field "13308" 50
    text "user:" 30 right user: field "qenkXXX"
    text "password:" 60 right password: field "CRfa8kuXXX" 120
    return
    text "topic:" 50 right topic: field 200 "/test"
    text "message" 60 right message: field 300 "Hello World!"
    return
    button "Publish" [
        call rejoin ["mosquitto_pub.exe -h " broker/text " -p "
port/text " -u " user/text
        " -P " password/text { -t "} topic/text {"} { -m "} message/text
 {"}
        ]
    ]
]
```



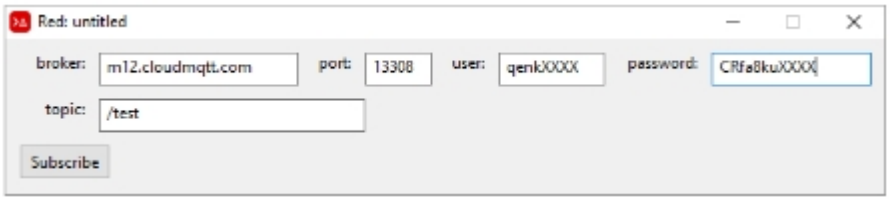
Você pode usar `print` no lugar de `call` no script acima para ver o comando completo enviado para **mosquitto_pub.exe.**

Subscrevendo:

Subscrever usando o **mosquitto_sub.exe** é um pouco mais complexo, porque as mensagens recebidas são escritas no console CLI do cmd. Eu não descobri como repassar isso constantemente para um script de Red. Minha solução até agora é redirecionar a saída do mosquitto_sub.exe para um arquivo de texto e verificar constantemente para detectar qualquer alteração no tamanho do arquivo. Se ele mudar, o script o lê para obter as novas mensagens.

Este script subscreve o tópico e redireciona as saídas para **mqttlog.txt** usando o comando de redirecionamento do cmd **">"**:

```
Red [needs view]
view [
    text "broker:" 50 right broker: field "m12.cloudmqtt.com" 150
    text "port:" 30 right port: field "13308" 50
    text "user:" 30 right user: field "qenkXXXX"
    text "password:" 60 right password: field "CRfa8kuXXXX" 120
    return
    text "topic:" 50 right topic: field 200 "/test"
    return
    button "Subscribe" [
        call/shell rejoin ["mosquitto_sub.exe -h " broker/text " -p "
port/text " -u " user/text
        " -P " password/text { -t "} topic/text {" > mqttlog.txt}
        ]
    ]
]
```



E este script constantemente verifica se houve alteração no tamanho de **mqttlog.txt** e, se houver, copia o texto do arquivo para uma `area`:

```
Red [needs: view]
oldsize: 0
view [
    mqttlog: area rate 2 ;verifica o arquivo texto duas vezes por
```

```
segundo
    on-time [
        newsize: size? %"mqttlog.txt"
        if newsize <> oldsize [
            mqttlog/text: read %"mqttlog.txt"
            oldsize: newsize
        ]
    ]
]
```