Red

Programming Language

Guaracy Monteiro
https://github.com/guaracy/red

31 de outubro de 2015

Sumário

Listagens

Lista de Tabelas

1 Introdução

1.1 Objetivo

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

1.2 A Linguagem

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

1.3 Configuração

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

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

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

2 Sintaxe

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

2.1 Delimitadores

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

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

2.2 Sintaxe livre

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

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

Listagem 1: Sintaxe livre

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

2.3 Comentários

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

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

3 REPL

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

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

red>>
```

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

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

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

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

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

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

```
red>> help insert

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

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

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

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

```
        count   [number!]

red>>
```

4 Variáveis

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

4.1 Nomenclatura

4.2 Operação

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

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

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

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

Formato	Significado
`var`	Avalia a variável e retorna o resultado.
`var:`	Atribui um valor a uma variável.
`:var`	Retorna o valor de uma variável sem avaliá-lo.
`'var`	Trata a variável como um simbolo e retorna o valor sem avaliá-lo.

5 Tipos de dados

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

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

No momento, estão disponíveis os seguintes tipos (alguns não serão utilizados em condições normais e, mais importante, alguns ainda não foram definidos comodata e hora. Por enquanto temos:
`action!, binary!, bitset!, block!, char!, datatype!, error!, file!, float!, function!, get-path!, get-word!, hash!, integer!, issue!, lit-path!, lit-word!, logic!, map!, native!, none!, object!, op!, pair!, paren!, path!, percent!, point!, refinement!, routine!, set-path!, set-word!, string!, tuple!, typeset!, unset!, url!, vector!, word!`

5.1 binary!

TBD

5.2 bitset!

TBD

5.3 block!

TBD

5.4 char!

TBD

5.5 datatype!

TBD

5.6 error!

TBD

5.7 file!

TBD

5.8 float!

TBD

5.9 function!

TBD

5.10 get-path!

TBD

5.11 get-word

TBD

5.12 hash!

TBD

5.13 integer!

TBD

5.14 issue!

TBD

5.15 lit-path!

TBD

5.16 lit-word!

TBD

5.17 logic!

TBD

5.18 map!

TBD

5.19 native!

TBD

5.20 none

TBD

5.21 object!

TBD

5.22 op!

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

```
red>> ? op!
     %  =>  Returns what is left over when one value is divided by another
     *  =>  Returns the product of two values
     ** =>  Returns a number raised to a given power (exponent)
     +  =>  Returns the sum of the two values
     -  =>  Returns the difference between two values
     /  =>  Returns the quotient of two values
```

```
     // =>  Compute a nonnegative remainder of A divided by B
     <  =>  Returns true if the first value is less than the second
     <<
     <= =>  Returns true if the first value is less than or equal to the second
     <> =>  Returns true if two values are not equal
     =  =>  Returns true if two values are equal
     == =>  Returns true if two values are equal, and also the same datatype
     =? =>  Returns true if two values have the same identity
     >  =>  Returns true if the first value is greater than the second
     >= =>  Returns true if the first value is greater than or equal to the second
     >>
     >>>
     and        =>  Returns the first value ANDed with the second
     or =>  Returns the first value ORed with the second
     xor        =>  Returns the first value exclusive ORed with the second
red>>
```

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

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

Definimos uma função `pertence` e criamos o operador ∈.

5.23 pair!

TBD

5.24 paren!

TBD

5.25 path!

TBD

5.26 percent!

TBD

5.27 point!

TBD

5.28 refinment!

TBD

5.29 routine!

TBD

5.30 set-path!

TBD

5.31 set-word!

TBD

5.32 string!

TBD

5.33 tuple!

TBD

5.34 typeset!

TBD

5.35 unset!

TBD

5.36 url!

TBD

5.37 vector!

TBD

5.38 word!

TBD