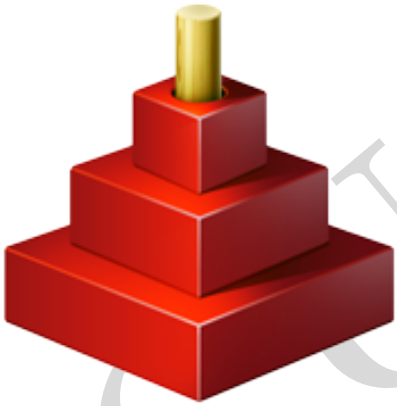

Red Programming Language

Guaracy Monteiro

4 de Outubro de 2017

Conteúdo

1 Introdução

1.1 Atenção

Atenção

Lembre que **Red** é uma linguagem em desenvolvimento e na versão α. Muitas funções ainda não foram implementadas, outras estão parcialmente implementadas e podem conter bugs. Para ter uma melhor visão do que esperar para a versão **1.0**, você pode ver o roteiro da linguagem e andamento do desenvolvimento. Mesmo assim, você poderá ficar espantado com o que já é possível fazer com a linguagem.

Principais restrições:

- Atualmente só está disponível a versão para 32 bits, o que não impede de rodar em um SO de 64 bits.
- GUI disponível apenas para Windows e OSX. Para Android em desenvolvimento pela equipe principal. Para GTK3, testes por terceiros.
- GC ainda não integrado a linguagem.

Para os exemplos e documentação foi utilizado a seguinte versão: `Red for Linux version 0.6.3 built 4-Oct-2017/9:29:04-03:00`

Este documentos está sendo escrito no Emacs com org-mode. Uma das facilidades de entrada de código para os exemplos, sua execução e apresentação do resultado logo a seguir. Como desvantagem, não é possível incluir

texto entre o quadro de código e o dos resultados.

1.2 Motivos para usar Red

Apenas um: **Melhorar suas habilidades de programador.**

Ficar comparando linguagens não tem muito cabimento pois, na grande maioria das vezes, não passa de gosto pessoal. Com maior ou menor esforço, todas fazem a mesma coisa. Algumas são desenhadas para determinadas tarefas e as executam com maior facilidade e menos código. Outras possuem um paradigma que você se adapta melhor. E assim vai. Não é raro opiniões como *"Sua linguagem faz isso em duas linhas mas a minha faz em uma"*, *"A minha já vem com tal facilidade e a sua é necessário programar a facilidade"*, etc..

Entre outras coisas que chamaram a minha atenção com **Red** , o parse foi uma delas. Sempre patinei um pouco com expressões regulares, achando parecidas com um programa em J (quem não conhece não vai achar diferença mesmo :D). O parse de **Red** faz a mesma coisa e até mais com uma estrutura mais legível. A facilidade de criar DSL.

A facilidade para eu desenvolver algo sem sair do Linux, compilar e executar diretamente no Windows (outro SO) ou Raspberry Pi (outra arquitetura). Também existe a possibilidade de executar **Red** de outras linguagens/ferramentas, bastando ter/fazer uma interface com a biblioteca (veja Julia utilizando **Red**).

1.3 Instalação

Na realidade, nem é possível chamar de instalação. Basta você baixar o executável da página e rodar. Atualmente o download está na faixa de 1M. Na página existem outras informações, principalmente algumas dependências se o seu SO for de 64bits.

No início existem os links para baixar a última versão estável (o que não faz muito sentido para uma linguagem alfa). No final da página existem os links para *latest builds* o que faz mais sentido. Sempre que uma nova função seja incluída ou correções/alterações de funções existentes sejam efetuadas, um novo executável será gerado.

Existe a possibilidade de trabalhar com os fontes do GitHub mas seria apenas aconselhável para desenvolvedores ou se você deseja testar novas funcionalidades de outras ramificações.

Formas de execução:

- `red` : Entra no REPL (Read–Eval–Print Loop) permitindo que você digite comandos interativamente. Também permite a execução de outros programas em **Red** .
- `red -c` ou `red -r` : Compila o programa indicado. A opção `-c`, indicada para o desenvolvimento, irá gerar uma biblioeca local agilizando novas compilações após alterações. A opção `-r` (release) é mais demorada e empacota no executável todos os dados necessários. É indicada para distribuir o programa final sem dependências.
- `red -t` : Compila para uma plataforma diferente da original, isto é, um sistema operacional/arquitetura diferente.
- `red -h` : Mostra as opções disponíveis (como compilar uma bibloteca, etc.).

A primeira execução irá demorar um pouco pois irá gerar uma alguma biblioteca que será usada pelo programa bem como a geração do console que será executado em seguida. digite `help` no REPL para maiores informações.

1.4 Duas por uma

A linguagem **Red** é dividida (fornecida) em duas partes:

- Red : A linguagem **Red** é uma linguagem de programação a versão alfa) fortemente baseada em REBOL, compartilhando basicamente, a sintaxe semelhante, a homoiconicidade e o grande número de tipos de dados. É uma linguagem de alto nível no estilo de Ruby, Phyton, etc.. Pode ser executada através do REPL (Read-Evaluation-Print-Loop), usar um arquivo como script ou compilada. Neste último caso, é possível em uma plataforma gerar executáveis para Windows, OSX, Linux e Android (futuramente iOS).
- Red/System : É uma linguagem compilada no mesmo estilo de C mas compartilhando a sintaxe de **Red** , gerando executáveis pequenos e rápidos, permitindo a criação de programas, bibliotecas ou drivers.

1.5 Homoicônica

Simplesmente quer dizer que a representação interna e externa são as mesmas, isto é, não faz distinção entre código e dados. Apesar de parecer simples, confere grande poder a linguagem e faz você pensar em novas formas de resolução de problemas.

1.6 Variáveis são tipadas?

Se você leu o parágrafo anterior, a pergunta deveria ser: *"Se não existe distinção entre código e dados, existem variáveis?"* E a resposta seria: *"Não."* Pelo menos não no sentido que estamos acostumados. Existem palavras que fazem referência a um determinado trecho existente na memória e que pode ser qualquer coisa. Se for um trecho de código, ele poderá ser executado e produzir um resultado. Se for um dado, ele poderá ser utilizado para um determinado processamento. Os dados possuem tipo e **Red** é rica em tipos. Entre outros, podemos citar inteiro, string, caractere, percentual, caminho, par, arquivo, url, rotulo, tag, hora, vetor, caminho, função, email, etc..

Mas qual o erro você poderia cometer confundindo uma palavra com uma variável? Veja o trecho abaixo e o resultado produzido.

```
1  teste: func [/local s] [
2      s: ">"
3      append s "foo"
4      print s
5  ]
6
7  print "Conteúdo inicial"
8  probe type? :teste
9  print source teste
10 teste
11 print "Conteúdo após a execução"
12 print source teste
```

```
Conteúdo inicial
function!
teste: func [/local s][s: ">"
    append s "foo"
    print s
]

>foo
Conteúdo após a execução
teste: func [/local s][s: ">foo"
    append s "foo"
    print s
]
```

Na linha 1 dizemos que a palavra **teste** aponta (os dois pontos após a palavra é como fazemos a atribuição) para um trecho do código que é uma função. Na linha 2 , a palavra **s** aponta para um trecho do código que é uma string `">"`. Utilizando o conceito de variável como o conhecemos, a operação da linha 3 apenas adicionaria

a string *foo* ao conteúdo da variável. Como *s* aponta para um endereço do código, o que fazemos foi adicionar a string ao trecho do código. Para a avaliação correta, deveria ser utilizada a forma `s: copy ">"`. O mesmo não acontece com dados numéricos (e alguns outros) que são mantidos na pilha. Mas não se preocupe com isto agora. Tem muita coisa pela frente.

2 Linguagem

2.1 Tipos de dados

Não é porque a linguagem não diferencia dados de código que ela não possui tipos de dados. Pelo contrário, seja para facilitar o trabalho do programador ou para conferir uma maior legibilidade ao programa, existe uma entensa gama de tipos.

É possível definir funções com tipagem dinâmica ou estática, isto é, podemos não indicar o tipo dos parâmetros e a linguagem não fará nenhuma checagem no momento da chamada. Apenas irá verificar a possibilidade da operação no momento em que ela for efetuada. Também é possível definir o tipo dos argumentos na declaração da função. Neste caso, se no momento da chamada algum parâmetro não for conpatível, será acusado um erro.

Abaixo, definimos uma função `soma-din: func[a b]` que aceitará qualquer valor como parâmetros sem acusar nenhum erro durante a chamada. O erro só é acusado quando a operação `a + b` for executada.

```
print "Função sem tipagem dos parâmetros"
soma-din: func[a b][
    a + b
]
print soma-din 1 2
print soma-din 1.5 2.8
print soma-din "yu" "yu"
```

```
Função sem tipagem dos parâmetros
3
4.3
,*** Script Error: + does not allow string! for its value1 argument
,*** Where: +
,*** Stack: soma-din
```

Só por curiosidade, `help +` irá mostrar mais informações sobre a função (no caso operador) e podemos ver que não aceita *string* como argumento.

```
help +
```

```
USAGE:
     value1 + value2

DESCRIPTION:
     Returns the sum of the two values.
\begin{itemize}
\item is an op! value.
\end{itemize}

ARGUMENTS:
     value1        [number! char! pair! tuple! vector! time! date!]
     value2        [number! char! pair! tuple! vector! time! date!]

RETURNS:
     [number! char! pair! tuple! vector! time! date!]
```

Em uma função onde definimos o tipo dos parâmetros como `soma-tip: func[a [integer! percent!] b [integer! percent!]]`, se for chamada com algum argumento que não seja *integer* ou *percent*, irá gerar um erro antes de sua execução. O erro é acusado no momento da chada da função.

```
print "Função com tipagem dos parâmetros"
soma-tip: func[a [integer! percent!] b [integer! percent!]][
    a + b
]
print soma-tip 1 2
try [print soma-tip 1.4 5]
```

```
Função com tipagem dos parâmetros
3
,*** Script Error: soma-tip does not allow float! for its a argument
```

```
,*** Where: soma-tip
,*** Stack: soma-tip
```


print soma-tip 1.4 5

2.1.1 datatype!

2.1.2 unset!

2.1.3 none!

2.1.4 logic!

2.1.5 block!

2.1.6 paren!

2.1.7 string!

2.1.8 file!

2.1.9 url!

2.1.10 char!

2.1.11 integer!

2.1.12 float!

2.1.13 word!

2.1.14 set-word!

2.1.15 lit-word!

2.1.16 get-word!

2.1.17 refinement!

2.1.18 issue!

2.1.19 native!

2.1.20 action!

2.1.21 op!

2.1.22 function!

2.1.23 path!

2.1.24 lit-path!

2.1.25 set-path!

2.1.26 get-path!

2.1.27 routine!

2.1.28 bitset!

2.1.29 point!