Helpin' Red

**Table of contents**

[Homepage](#_topic_Homepage) 5

[Úvod](#_topic_Uvod) 5

[Začínáme](#_topic_Zaciname) 8

[Notepad++](#_topic_Notepad) 9

[Console 2](#_topic_Console2) 14

["Hello world"](#_topic_Helloworld) 17

[Poznámky ke skladbě](#_topic_Poznamkykeskladbe) 23

[Introspekce a nápověda](#_topic_Introspekceanapoveda) 25

[Vyhodnocení](#_topic_Vyhodnoceni) 27

[Zvýraznění kódu](#_topic_Zvyraznenikodu) 30

[Konzola - vstup a výstup](#_topic_Konzola_vstupavystup) 31

[Exekuce kódu](#_topic_Exekucekodu) 33

[Datové typy](#_topic_Datovetypy) 35

[Další datové typy](#_topic_Dalsidatovetypy) 42

[Hash!, vector! and map!](#_topic_Hashvectorandmap) 46

[Konverze datových typů](#_topic_Konverzedatovychtypu) 48

[Bloky a řady](#_topic_Blokyarady) 49

[Navigace řadami](#_topic_Navigaceradami) 51

[Series "getters"](#_topic_Seriesgetters) 55

[Series "changers"](#_topic_Serieschangers) 63

[Formátování dat](#_topic_Formatovanidat) 74

[Matematika a logika](#_topic_Matematikaalogika) 79

[Konverze bází](#_topic_Konverzebazi) 89

[Kryptografie](#_topic_Kryptografie) 91

[Kopírování](#_topic_Kopirovani) 93

[Opakování](#_topic_Opakovani) 94

[Podmínky](#_topic_Podminky) 97

[Manipulace s textem](#_topic_Manipulacestextem) 101

[Práce s časem](#_topic_Pracescasem) 107

[Ošetření chyb](#_topic_Osetrenichyb) 109

[Soubory](#_topic_Soubory) 110

[Psaní do souboru](#_topic_Psanidosouboru) 114

[Čtení ze souboru](#_topic_Ctenizesouboru) 116

[Funkce](#_topic_Funkce) 118

[Objekty](#_topic_Objekty) 123

[Reaktivní programování](#_topic_Reaktivniprogramovani) 125

[Rozhraní OS](#_topic_RozhraniOS) 128

[I/O](#_topic_IO) 130

[GUI](#_topic_GUI) 131

[Nastavení kontejneru](#_topic_Nastavenikontejneru) 137

[Layout](#_topic_Layout) 141

[Faces](#_topic_Faces) 146

[Události a aktéři](#_topic_Udalostiaakteri) 164

[Introspekce událostí](#_topic_Introspekceudalosti) 169

[Pokročilá témata](#_topic_Pokrocilatemata) 171

[Draw](#_topic_Draw) 179

[Vlastnosti čáry](#_topic_Vlastnosticary) 186

[Barva, gradienty a vzory](#_topic_Barvagradientyavzory) 189

[Transformace v rovině](#_topic_Transformacevrovine) 196

[Sub-dialekt Shape](#_topic_Sub_dialektShape) 202

[Parse](#_topic_Parse) 207

[Matching](#_topic_Matching) 211

[Iterace](#_topic_Iterace) 214

[Extrakce](#_topic_Extrakce) 217

[Úprava vstupu](#_topic_Upravavstupu) 219

[Control flow](#_topic_Controlflow) 220

[Introspekce](#_topic_Introspekce) 222



**Helpin'Red**

**Průvodce s příklady** [**programovacím jazykem Red**](https://www.red-lang.org)

Created by **Ungaretti** - see Helpin´Red

**Do češtiny přeložil Tovim**

Předkládaný text je pokusem o srozumitelný výklad programovacího jazyka Red. Český překlad zatím nedokončené jeho oficiální dokumentace lze číst na <https://doc.red-lang.org/cs> .

Podněty a připomínky k překladu jsou vřele vítány! Můžete je zaslat na adresu [tovim@seznam.cz](mailto:tovim@seznam.cz?subject=HelpingRed)

Tento text byl vytvořen aplikací [HelpNDoc](https://www.helpndoc.com/) dne 25. 9. 2018 a lze jej číst také ve formátu [PDF a](http://helpin.red/cs/Helpin'%20Red-cs.pdf) [Word.](http://helpin.red/cs/Helpin'%20Red-cs.docx)

 You may copy and distribute this work, but you can't make any commercial use or profit from it or any derivative work. Any derivative work must have the same license and give proper credit to the original work.

*Created with the Standard Edition of HelpNDoc:* [*iPhone web sites made easy*](https://www.helpndoc.com/feature-tour/iphone-website-generation)

**Úvod**

**O programovacím jazyku Red**

* Red je programovací jazyk, který se se vším všudy vejde do jednoho spustitelného souboru o velikosti asi 1 MB. Žádná instalace, žádné sestavování.
* Red je 'free' a 'open-source'.
* Red je interpretovaný jazyk ale můžete svůj kód kompilovat a generovat samostatnou prováděčku.
* Red si před interpretací část kódu kompiluje, takže je docela rychlý.
* Red je jednoduchý. Jeho kód je čistý a stručný.
* Red je aktuálně (duben 2018) ve fázi 'alfa' ale má ambiciózní cíle:
  + býti multi-platformní;
  + míti nativní systém GUI s UI dialektem;
  + býti multi-úrovňový (full-stack) programovací jazyk.
* Red je 'open-source' pokračováním jazyka Rebol, vytvořeného tvůrcem Amigy - Carlem Sassentarhem. Chcete-li si vyzkoušet některé procedury, které nejsou zatím v Redu zavedeny, můžete si je vyzkoušet v instalovaném Rebolu.
* Red je vyvíjen skupinou lidí pod vedením Nenada Rakocevice.
* V poslední době posbíral Red nemalý fond z ICO a v Paříži byla ustavena společnost Red Foundation, takže je naděje že tu Red s námi chvíli pobude.

Ukázka Redu:

Red [needs: **view**]

**view** [

f1: field "First name"

f2: field "Last name"

button "Greet Me!" [

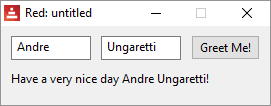
t1/text: rejoin ["Have a very nice day " f1/text " " f2/text "!"]

]

**return**

t1: text "" 200

]



Pokud vás to zaujalo, můžete nahlédnout do [Short Red Examples](http://redprogramming.com/Short%20Red%20Code%20Examples.html) od Nicka Antonaccia, případně do českého překladu prozatimní oficiální dokumentace [Programovací jazyk Red.](https://doc.red-lang.org/cs/)

**Introductory notes by Ungaretti:**

This is an evolution of the [Red Language Notebook](https://www.gitbook.com/book/ungaretti/red-language-notebook/details).   
I chose to use [HelpNDoc](https://www.helpndoc.com/) software to develop a more friendly and useful interface.

Notes:

* I use Windows, so this work is based on this Operating System.
* I'm not an experienced or even a good Red programmer, in fact, I'm not a programmer at all.
* English is not my native language.
* This isn't a complete reference for the Red language (yet?).
* I did not use the best coding style in many examples. Please, take a look at [Red's coding style guide](https://doc.red-lang.org/v/v0.6.0/Coding-Style-Guide.html).
* I try to make my work original, but some text was copied and pasted from [Red's official documentation](https://doc.red-lang.org/en/) and I based some examples on what I found at:
  + [red-by-example.org](http://www.red-by-example.org/index.html)by Arie van Wingerden and Mike Parr
  + [mycode4fun.com.uk](http://www.mycode4fun.co.uk/red-beginners-reference-guide) by Alan Brack
  + [redprogramming.com](http://redprogramming.com/Home.html) by Nick Antonaccio

Also, a lot of help came from the Red community at [gitter.im/red/home](https://gitter.im/red/home)

**Nezbytná poznámka překladatele:**

Mám zato, že programátor nebo zájemce o programování musí alespoň částečně umět anglicky a mít znalost alespoň nezbytné zásoby anglických slov.

Vycházeje z tohoto předpokladu, pokusím se některé termíny nepřekládat, pokud budou rozumně použitelné v české větě.

Přivítám pomoc při překladu termínu **face** (tvář), pro který jsem použil výraz **piškot**, jenž některé Redisty (Rebolisty) poněkud irituje. Slovo face zde nemá s tváři mnoho společného, neboť je to zkrácenina slova interface. Face i interface mi nepřijdou jako označení pro daný objekt jednoznačně přiléhavá a proto se jich nedržím. Alternativní označení k face je widget, jež se mi líbí ale špatně se s ním v české větě pracuje. Také se mi líbí slovo **grafion**. Tož uvidíme.

Použítí upraveného editoru **Notepad** ++ vřele doporučuji - spolu s interaktivní konzolou Redu v aplikaci [Console 2](https://sourceforge.net/projects/console-devel/files/console-releases/2.0b148a/) pro jednodušší příklady.

*Created with the Standard Edition of HelpNDoc:* [*Easily create EPub books*](https://www.helpndoc.com/feature-tour)

**Začínáme**

**Instalace prostředí Red**

Nejprve je zapotřebí stáhnout si do počítače vhodnou [instalaci Redu](https://www.red-lang.org/p/download.html). Když ji (dvojklikem) spustíte, vytvoří a otevře ve vašem monitoru interaktivní konzolu Red Console.

Cestu ke složce s binárním souborem red~.exe lze uložit do systémové proměnné System PATH. To umožní otevírat interpretační prostředí Red Console odkudkoliv v počítači prostým zápisem slova **red** do příkazového řádku systémové konzoly. Tento způsob lze doporučit pouze věcí znalým, neboť systémová proměnná rozezná pouze bazální název red.exe. Z toho důvodu je nutné buď instalační soubor na disku přejmenovat nebo vytvořit soubor red.bat.

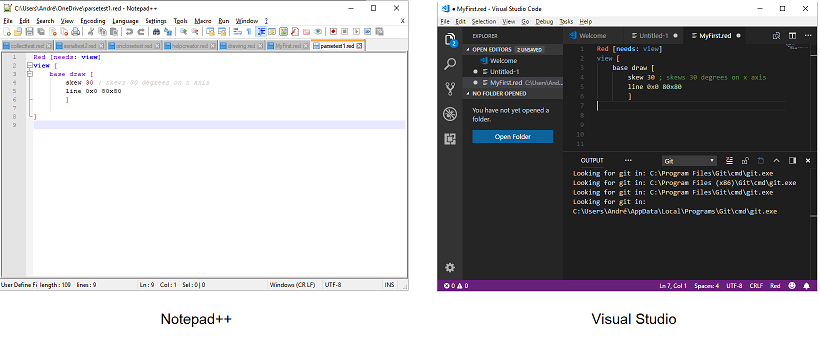
Další přístup k interpretačnímu prostředí Redu vytvoříme nastavením doporučovaného editoru [Notepad++](#_topic_Notepad) a ještě další přístup nastavením rovněž doporučované konzoly [Console2](#_topic_Console2).

Pokyny pro spouštění skriptů jsou popsány v kapitole [Hello world](#_topic_Helloworld), nejprve si však musíte vybrat textový editor

**Výběr editoru**

Skripty lze psát v jakémkoli textovém editoru, který produkuje čisté textové soubory. V našem Průvodci se omezíme na prezentaci populárního a účinného editoru [Notepad++](#_topic_Notepad) , který byl použit pro všechy skripty příkladů, spolu s konzolovou aplikací [Console 2.00](#_topic_Console2), ve které jsem měl otevřené interaktivní prostředí Redu pro zadávání jednoduchých kódů.

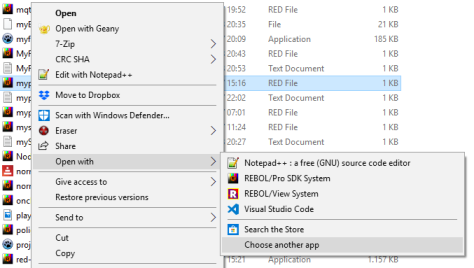
Alternativou k Notepad++ je editor a programovací prostředí [Visual Studio.](https://code.visualstudio.com/)



**Některé užitečné informace:**

Skripty Redu jsou čisté textové soubory. Mohou mít jakoukoli extenzi ale je vhodné používat pouze extenzi .red. Soubory s touto extenzí lze ve Windows asociovat s vybraným editorem.

Následující obrázek docela výstižně ilustruje vytvoření této asociace. Pravým klikem na souboru s extenzí .*red* otevřeme kontextovou nabídku, kde vybereme *Otevřít v programu (Open with)* a z této další nabídky vybereme žádanou aplikaci :



*Created with the Standard Edition of HelpNDoc:* [*Easy to use tool to create HTML Help files and Help web sites*](https://www.helpndoc.com/help-authoring-tool)

**Notepad++**

# Provizorní IDE v Notepad++

Pěkné vývojářské prostředí Redu ve Windows lze vytvořit s použitím editoru Notepad++. Můžete postupovat podle následujících instrukcí nebo si stáhnout instalační složku [Red\_IDE](http://helpin.red/Red_IDE.zip). Tato zazipovaná složka obsahuje kromě potřebných konfiguračních souborů také aplikace notepad++.exe a red-063.exe.

Pokud si ji stáhnete, máte v jeho (anglickém) Notepadu++ již přednastavený jazyk Red-lang i položku Red-run v menu Run, takže dále popisovaná nastavení nemusíte provádět. Složka Red\_IDE dokonce obsahuje připravenou pracovní složku Programy.

První spuštění red-063.exe ve složce RedIDE chvíli trvá, než otevře konzolu, protože se instaluje a provádí sestavení aplikace Red.

Následující pokyny popisují postup pro instalaci z individuálně stažených instalačních souborů:

1 - Vytvořte v počítači složku **RedIDE**/;

2 - Stáhněte si aktuální [Notepad++](https://notepad-plus-plus.org/download/v7.5.6.html) [.](https://notepad-plus-plus.org/download/v7.5.1.html) Určitě si vyberte "minimalistický paket **32-bit x86**".

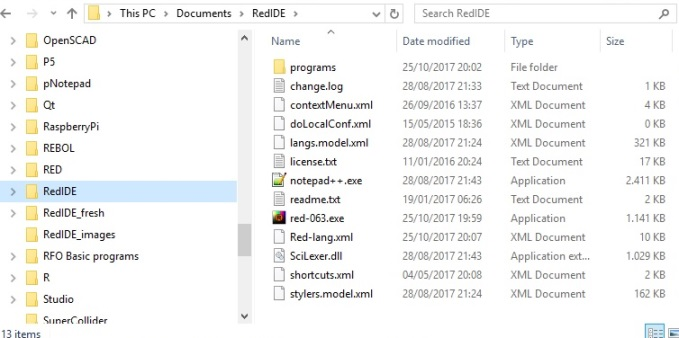


3 - Rozbalte Notepad++ ve vytvořené složce RedIDE/.

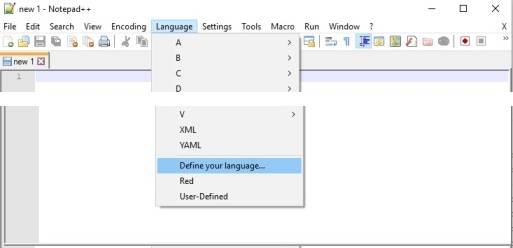
4 - Stáhněte si aktuální verzi [Redu pro Windows](http://www.red-lang.org/p/download.html) a přemístěte ji do složky RedIDE/.

5 - Stáhněte si konfigurační soubor [Red-lang.xml](https://github.com/Ungaretti/Notepad-config-file-for-Red-Language). Vložte jej rovněž do složky RedIDE/.

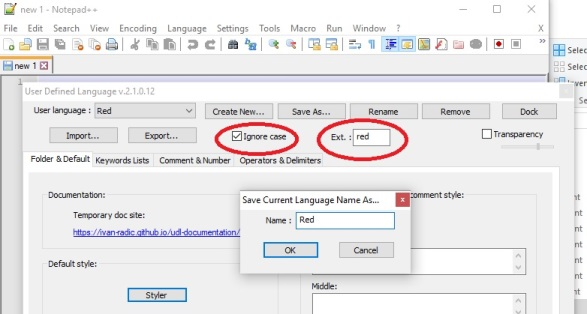
6 - Rovněž doporučuji, aby jste si uvnitř složky RedIDE vytvořili ještě složku **Programy** pro ukládání programů. Složka RedIDE nyní může vypadat takto:



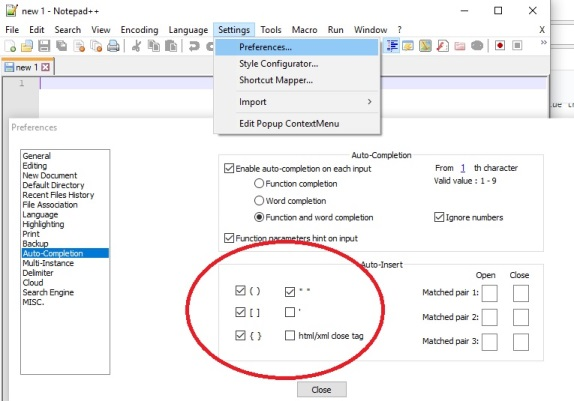
7 - V otevřemém Notepad++ aktivujte nabídku “Syntaxe/Definovat vlastní syntaxi…” (případně Language/Define your language...) . V otevřeném okně klikněte na “Import…” a vyberte stažený “Red-lang.xml”. Vysouvací návěstí vám řekne, že byl import úspěšný.



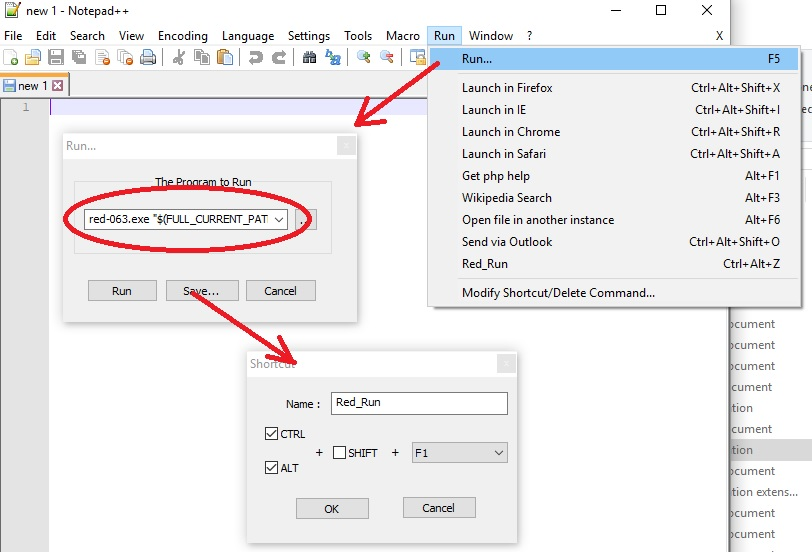
8 - Zatrhněte políčko “Nerozlišovat malá a velká písmena”. Klikněte na “Uložit jako…” a uložte jako “Red”. Potom napište “red” do políčka “Přípona”. Obrázek dole je ve fázi "Uložte jako".



9 - Otevřte okno “Nastavení/Volby/Automatické dokončování”. Zatrhněte políčka pro “()”, “” “{}” and “[]”. (If you can’t find those in the options, create a “matched pair”.)



10 - Otevřte okno “Spustit/Spustit..” , zapište název staženého binárního souboru Redu a za mezeru opište tento text: “$(FULL\_CURRENT\_PATH)”. Uložte, zadejte oblíbené klávesové zkratky a jméno - například “Red\_Run” (pamatujte, že později můžete chtít něco jako “Red\_Compile” etc.) a klikněte “OK”.

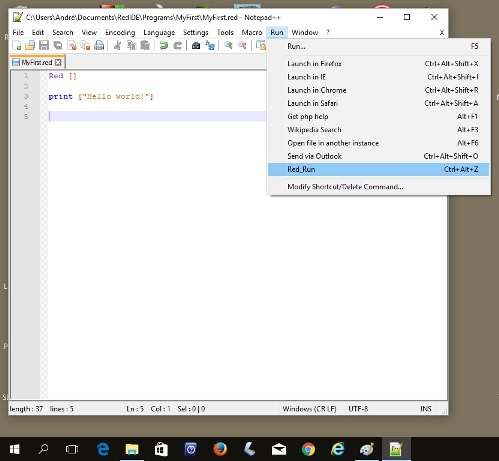


11 - Zavřete a restartujte Notepad++.

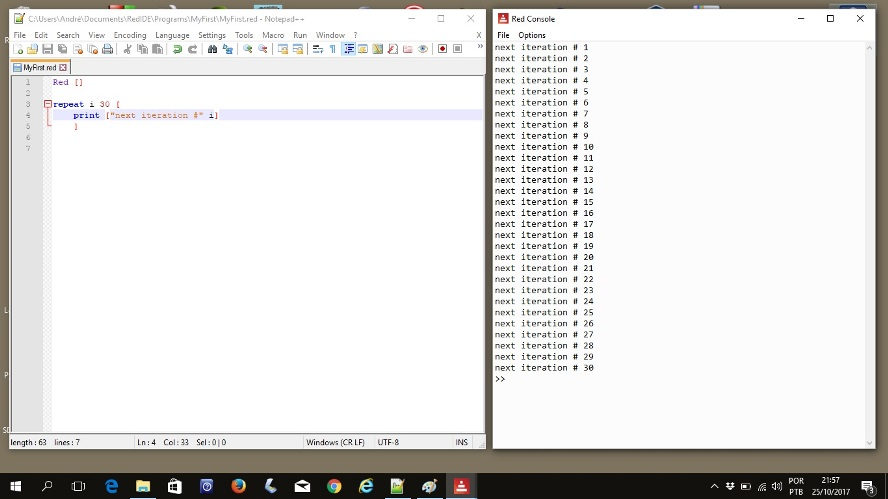
12 - Otevřte v Notepad++ nový dokument a opište dolní text. V případě tohoto "Red[]" je Red výjimečně "case sensitive".

Red[]   
print "Hello world!"

13 - Uložte program s extenzí .red (“MyFirst.red”). V zapsaném skriptu se zvýrazní vybraná slova. Doporučuji jej uložit do vytvořené složky Programy/. Klikněte “Spustit/Red\_Run”. Skript by se měl otevřít v grafické konzole Redu.



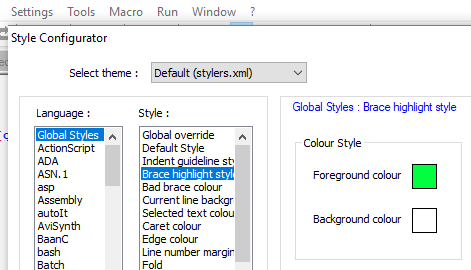
14 - Upravte rozměry obou aplikací a místěte je úhledně vedle sebe a máte hotové IDE.



15 - Silně doporučuji vytvořit si kopii složky “RedIDE/” jako například “RedIDE\_backup”.

16 - Nyní svůj skript zapíšete v Notepad++, **uložíte jej** a spustíte pokynem “Run/Red\_Run”.

Všiml jsem si, že v dané konfiguraci jsou párové hranaté závorky zvýrazněny stejnou barvou jako kulaté závorky.. Je možné to změnit v Settings>Style Configurator:



Znovu připomínám, že tato adaptace je možná pouze s Notepad++ 32-bit.

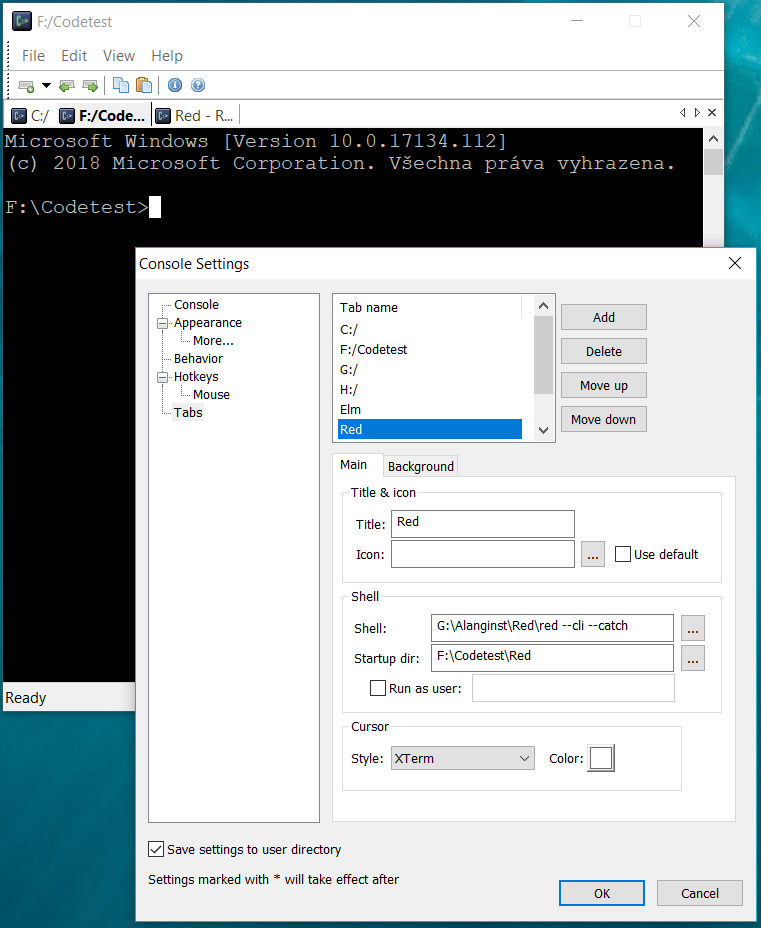
*Created with the Standard Edition of HelpNDoc:* [*Create HTML Help, DOC, PDF and print manuals from 1 single source*](https://www.helpndoc.com/help-authoring-tool)

**Console 2**

Výborné prostředí pro práci s interaktivní konzolou Redu ve Windows je aplikace **Console 2.00**, kterou si lze zdarma stáhnout ze stránky [SourceForge.](https://sourceforge.net/projects/console-devel/files/console-releases/2.0b148a/) Velikost instalace je menší než 2 MB.

V instalované konzole volíme **Edit** > **Settings**... kde zvýrazníme volbu **Tabs**, která nám rozbalí níže zobrazený formulář. V tomto formuláři zadáme název karty, případně cestu k souboru s ikonou.

V další skupině (Shell) zadáme cestu k binárnímu souboru red.exe a příslušné volby (viz obrázek). Na dalším řádku zadáme cestu ke složce, v níž se budou ukládat soubory, s nimiž bude interaktivní konzola Redu pracovat.



Výhoda této aplikace spočívá v tom, že si lze najednou otevřít více karet s různými programy či cestami, případně tentýž program či cestu otevřít vícekrát s různým obsahem.

V obrázku ilustrovaná úprava aktivace otevíraného souboru příponami --cli --catch způsobí, že se ve zvolené kartě otevře přímo interpretační konzola Redu (nikoliv Red Console). Pokud si na další kartě nastavíme Red bez přípon, bude nám otevírat volně přemistitelnou konzolu Red Console.

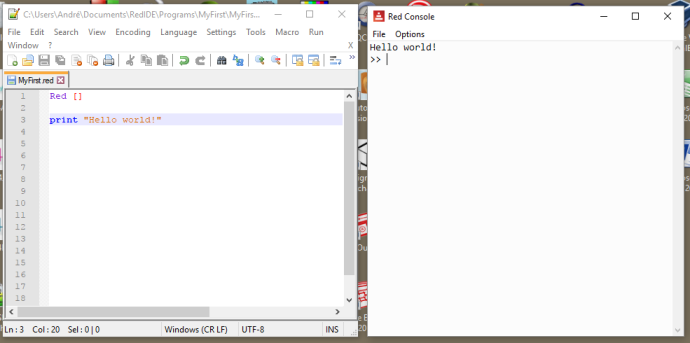
Zadaný příkaz lze po vyhodnocení opakovat (i vícekrát) šipkou nahoru. Po řádku se lze pohybovat šipkami vlevo či vpravo a klávesami *Home* či *End*. Interaktivní prostředí Redu opustíme příkazem **q** nebo **quit**.

*Created with the Standard Edition of HelpNDoc:* [*Generate Kindle eBooks with ease*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

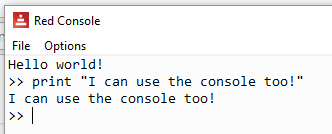
**"Hello world" - spuštění a kompilace**

**"Hello world" v konzole:**

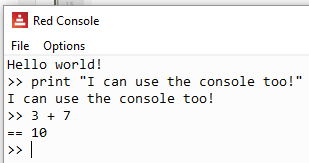
Otevřte v Notepadu++ již vytvořený skript "MyFirst.red" a spusťte jej (run/red-run). Měl byste v konzole dostat toto:



Okno vpravo je interaktivní konzola Red Console, někdy obecně označovaná jako REPL, což je zkratka sousloví Read, Evaluate, Print, Load . V její ploše zapište  print "I can use the console too!" a stiskněte Enter. Na to konzola reaguje výpisem zadaného textu na následujícím řádku:



Nyní zadejte 3 + 7 a stiskněte Enter:



Všimněte si, že musíte mít mezery mezi jednotlivými slovy. Mezery jsou povinné oddělovače a bez nich obdržíte chybu:

Hello world!   
>> **print "I can use the console too!"**   
I can use the console too!   
>> **3 + 7**   
== 10   
>> **3+7** ; no spaces!!!!!   
\*\*\* Syntax Error: invalid integer! at "3+7"   
\*\*\* Where: do   
\*\*\* Stack: load

Všimněte si, že jsem za 3+7 připsal ; no spaces!!!!! . Red ignoruje slova za středníkem.

**Zpracování skriptu:**

Interpretované programovací jazyky vykonávají kód skriptu řádek po řádku. Red není zcela interpretovaný, protože provádí jistou kompilaci před exekucí ale jeho programům se přesto říká skripty.

Na prvním řádku skriptu je povinně blok Red [ ], nikoliv RED, nikoliv red. Tento první blok je určen pro metadata programu, nicméně může být prázdný. Úplný blok by mohl vypadat takto:

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.

}

] ; cokoli před tímto blokem je ignorováno!

**print** "Hello World!"

**"Hello world" s grafickým rozhranním - GUI:**

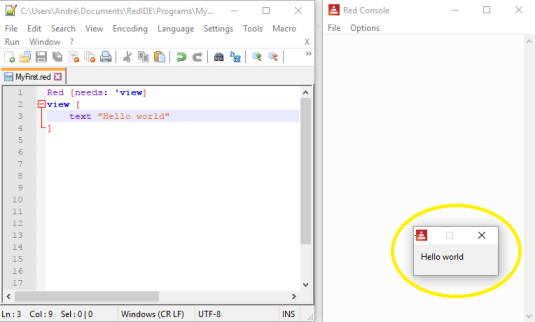
Jednou z nejnápadnějších vlastností Redu je jeho schopnost snadného vytváření programů s grafickým rozhraním. Chytře využívá vlastní API operačního systému. Jednoduchý prográmek pro Hello world v GUI vypadá takto:

Red [needs: '**view**]

**view [**

text "Hello World!"

]



Pokyn needs: 'view v bloku záhlaví říká Redu, že má načíst grafickou knihovnu "view". Toto záhlaví s pokynem není potřebné, zadáváme-li příkaz view [text "Hello World!"] z konzoly Redu, v níž je knihovna "view" již obsažena.

Není však zřejmě nic proti ničemu, když i v konzole uvedeme needs: 'view - již bez bloku Red [].

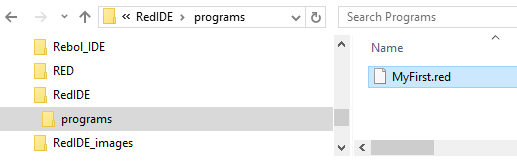
**Spuštění skriptu z konzoly:**

Spuštění kódu, zapsaného v konzole provedeme prostým příkazem Enter, spuštění kódu, zapsaného ve skriptu, provedeme v Notepad++ příkazem Run/Red Run. Spuštění skriptu z konzoly je popsáno v kapitole [Exekuce kódu](#_topic_Exekucekodu).

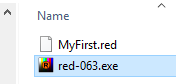
**Kompilace "Hello world" na spustitelný soubor:**

Prográmek GUI "Hello World" lze kompilovat na spustitelný soubor.

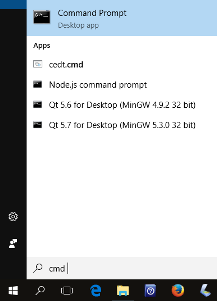
Máte-li výše uvedený program uložen jako "MyFirst.red" ve složce "programs" v adresáři "RedIDE", měl byste mít v počítači přibližně následující konstelaci:



Aby se nám produkty kompilace nerozsypaly do složky RedIDE, vložte do složky "programs" ještě kopii prováděčky Redu:



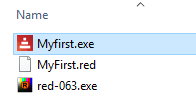
Otevřte systémovou konzolu. Nevíte-li jak, napište "cmd" do vyhledávacího pole Windows a potvrďte položku Příkazový řádek (Command Prompt):



V příkazovém řádku se nacédujte do složky s prováděčkou Red (právě jsme ji překopírovali do složky "programs") a zadejte žlutě označený text dole:



Red odpoví řadou sdělení a asi po 10 vteřinách budete mít ve složce "programy" pět dalších souborů, včetně "Myfirst.exe".



Dvojklikem jej aktivujte a hned se vám na obrazovce objeví úhledné sdělení "Hello World".

**Doplňující poznámky ke kompilování:**

Zjistil jsem, že se kompilovaná verze programu může chovat odlišně od interpretované verze. Měl jsem problémy s příkazy "print", které jsem do programu vložil kvůli ladění. Mám zato, že volání příkazů konzoly v binárním (executable) souboru není to pravé ořechové. Také jsem měl problémy s globálními proměnnými (slovy) uvnitř funkcí; zdá se že kompilátor je neoznačí jako globální. Poslední problém jsem řešil dvojím způsobem:

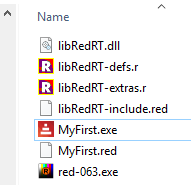
1. Deklaroval jsem proměnné (přiřadil hodnoty slovům) na počátku svého programu.
2. Při kompilaci jsem použil opci "-e" (v github není uvedena, patrně jde o experimentální volbu).

 -e znamená "encap" v Rebolu. Výstupem je binárka EXE, jejíž kód je interně interpretován, takže problémy, které jsou omezením kompilátoru, mohou být řešeny" - Gregg Irwin.)

MyFirst.red program jsme mohli kompilovat také s použitím pouze opce  -c (compile):



Po jedné z obou kompilací máte ve složce "programy" tyto soubory:



Jediné dva soubory, které musí být k disposici pro spuštění vašeho programu jsou **libRedRT.dll** a vaše prováděčka (binárka), v tomto případě **MyFirst.exe**.

Když ovšem spustíte binárku, kompilovanou pouze s opcí -c, otevře Red také obtěžující okno konzoly. Chcete-li se tomu vyhnout (kdo by nechtěl?), zadejte ještě opci -t windows, která zajistí kompilaci pro určenou platformu.



Měli byste být schopni kompilace pro níže uvedené platformy ale v současné době se Red stále vyvíjí a tak se může stát, že s některými platformami budete mít problémy (např. se zdá, že kompilace pro android nechodí).

**From Red's github:**

**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

**Způsoby kompilace:**

-c, --compile : Generuje v pracovním adresáři spustitelný kód (prováděčku) s použitím libRedRT. (**developement mode**)  
-d, --debug, --debug-stabs : Kompiluje zdrojový soubor v ladícím regimu (debug mode). STABS is supported for Linux targets.  
-dlib, --dynamic-lib : Generuje sdílenou knihovnu ze zdrojového souboru.   
-h, --help : Output this help text. -o <file>, --output <file> : Specify a non-default [path/][name] for  
 the generated binary file.  
-r, --release : Kompiluje v režimu **release mode**, spojujíc vše dohromady (implicitní je 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)

Existuje také volba -e. Viz "Doplňující poznámky ke kompilování" výše.

*Created with the Standard Edition of HelpNDoc:* [*Full-featured EPub generator*](https://www.helpndoc.com/create-epub-ebooks)

**Poznámky ke skladbě**

* Red je 'case insensitive' - s několika málo výjimkami, z nichž nejdůležitější je ta, že program musí začínat slovem  **Red** (nikoliv RED nebo red).
* Znaky new-line  jsou interpretem Redu většinou ignorovány. Relevantní výjimkou je  new-line uvnitř řetězce (stringu).
* Red je funkcionální jazyk, což znamená, že vyhodnocuje výsledky. Pořadí vyhodnocení výrazu není obvyklé a hovoříme o něm v části [Vyhodnocení](#_topic_Vyhodnoceni).
* Program Redu je dlouhý řetěz slov (words). Těmito slovy mohou být buď "data" nebo "aakce".
* Slova jsou oddělena jednou nebo více mezerami (whitespaces) .
* Red si vede slovník se systémovými slovy a s uživatelsky vytvořenými slovy.
* Slova mohou být seskupována do "bloků", vymezených hranatými závorkami. Blok je pouhá skupina slov, která může ale nemusí být vyhodnocena nějakou "akcí".
* Všechna programová data se nacházejí uvnitř samotného programu. Jsou-li potřebná externí data, jsou přidána k řetězci slov programu.
* Každé slovo musí při vyhodnocení mít nějakou hodnotu. Tato hodnota může pocházet z:

o slova samotného, je-li datem;

* + vyhodnocení, je-li slovo akcí;
  + odkazu na jiné slovo nebo blok (e.g. myRoom: 33).
* V Redu lze říci, že proměnná je přiřazena své hodnotě, nikoliv obráceně. V Redu vlastně nejsou žádné "proměnné", pouze slova přiřazená k hodnotám (datům).
* Kopírování slov (proměnných) může být komplikované. Chcete-li vytvořit opravdu nezávislou kopii, použijte slovo  copy . Viz kapitola [Kopírování](#_topic_Kopirovani).
* Podobně komplikované je vyprazdňování (clearing) řad ([series),](#_topic_Blokyarady) k nimž patří i řetězce. Pouhé přiřazení " " (empty string) nebo nuly nemusí přinést očekávaný výsledek. Takže k nulování řady je vhodné použít příkaz  clear.
* Každé slovo je nějakým datovým typem. Těch má Red pozoruhodné množství. Jsou vyjmenovány v kapitole  [Datové typy](#_topic_Datovetypy).
* Deklarované slovo datového typu word! lze použít různým způsobem:

|  |  |
| --- | --- |
| **Zápis** | **Význam** |
| word | Vzít přirozenou hodnotu slova. Je-li funkcí, nejprve ji vyhodnotit. |
| word: | Přiřadit (assign) slovo k hodnotě. |
| :word | Vzít hodnotu slova bez vyhodnocení. Užitečné pro získání definice funkce. |
| 'word | Zacházet se slovem jako s hodnotou (a word symbol) - bez vyhodnocení. |
| /word | Považovat slovo za upřesnění (refinement). Používá se hlavně u volitelných argumentů. |

**Refinements (upřesnění)**

Mnohé akce v Redu umožňují "upřesnění", které modifikuje chování příkazu. Refinement se deklaruje přidáním slova za lomítkem "/<refinement>".

*Created with the Standard Edition of HelpNDoc:* [*Create cross-platform Qt Help files*](https://www.helpndoc.com/feature-tour/create-help-files-for-the-qt-help-framework)

**Introspekce a nápověda**

Red má mimořádnou nápovědu. Pouhým zápisem několika slov v konzole lze získat obsáhlé množství informací o jazyku a o konkretní entitě.

function!**? (or help)**

Poskytne informaci o všech vyhrazených slovech Redu a rovněž o vašem vlastním kódu. Můžete použít příkaz  help nebo ?.  V konzole se vytiskne informace o použití nápovědy.

>> ? 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

Pokud přesně nevíte co hledáte, pomůže vám otazník:

>> **? -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?]

Můžete nalézt všechna definovaná slova pro daný datatyp.

>> **? tuple!**

Red 255.0.0

white 255.255.255

transparent 0.0.0.255

black 0.0.0

gray 128.128.128

; ... seznam je docela dlouhý

function!**what**

Vytiskne seznam globálně definovaných funkcí. Zkuste si to!

function!**source**

Zobrazí zdrojový kód mezaninové nebo uživatelem vytvořené funkce. Vyzkoušejte si například

 >> source replace .

function!**about**

Zobrazí číslo verze a datum sestavení.

**Introspekce grafických objektů**

Introspekci grafických objektů (piškotů) si ukážeme v kapitole [GUI](#_topic_GUI).

*Created with the Standard Edition of HelpNDoc:* [*Easy to use tool to create HTML Help files and Help web sites*](https://www.helpndoc.com/help-authoring-tool)

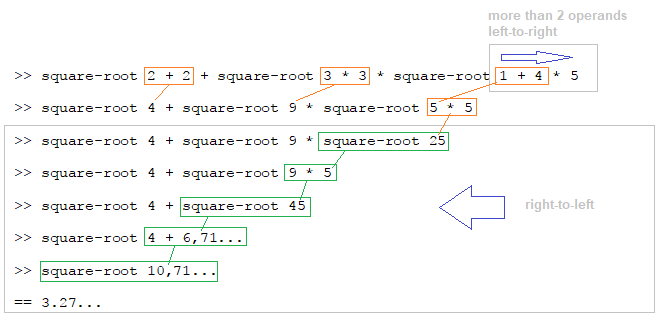
**Vyhodnocení výrazu**

Stále se rozhlížím po jednoduchém pravidle, vysvětlujícím proces vyhodnocování v Redu. Aktuálně mám dva oblíbené kandidáty. První je přímočarý a snadno použitelný. Druhý není příliš praktický ale dává nahlédnout (domnívám se) na způsob, kterým interpret Redu "myslí".

**1. První oblíbenec:**

* Všechny operace s  [infixovými operátory,](https://en.wikipedia.org/wiki/Infix_notation) které mají pouze hodnoty (nikoliv funkce) jako operandy, jsou vyhodnocovány přednostně. Mají-li tyto infixové výrazy více než dva operandy, jsou vyhodnoceny zleva doprava ( → ) bez žádných preferencí .
* Celý výraz je potom vyhodnocen zprava doleva (← ).

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



**2. Druhý oblíbenec se třemi koncepty:**

Zdá se, že to chodí ale mám pochybnosti o formální správnosti, neboť si nejsem jist, že každý infixový operátor má přesně korespondující operaci funkce.

**První koncept: Vždy zleva doprava →**

V Redu se všechno vyhodnocuje zleva do prava. Neexistuje žádný systém preferencí jako u jiných jazyků (např. se násobení neprovádí automaticky přen sčítáním). Přednost v pořadí si ovšem lze vynutit závorkami.

>> **2 + 3 \* 5**   
== 25

>> **2 + (3 \* 5)**   
== 17

Nejenom výrazy ale celý kód programu je vyhodnocován zleva doprava.

|  |
| --- |
| **Infixové operátory**  "+", "-", "\*", "/" jsou infixové operátory. Korespondují s funkcemi  add , multiply, divide a subtract, jež vyžadují dva argumenty:  3 + 2 je totéž ako add 3 2  5 \* 8 je totéž jako multiply 5 8 ...  ...a tak dále.  2 + 3 \* 5  je pouze více čitelná forma výrazu  multiply add 2 3 5  . Interpret Redu si potřebnou konverzi provede sám. |

**Druhý koncept: Vyhodnotitelné skupiny.**

V určité části kódu se vyskytují skupiny slov, které lze dílčí exekucí redukovat na základní datový typ. Například  [square-root 16 8 + 2 8 / 2 77] se ve skutečnosti skládá ze čtyř vyhodnotitelných skupin: square-root 16 ; 8 + 2 ; 8 / 2  a  77. Popisovanou redukci lze provést příkazem reduce:

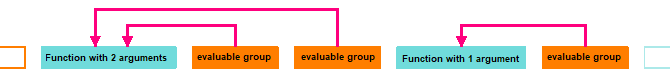
>> **a: [ square-root 16 8 + 2 8 / 2 77]**



>> **reduce a -> [4.0 10 4 77]**

**Třetí koncept: Funkce si vybírají své argumenty z vyhodnotitelných skupin**

Funkce si vybírá své argumenty z následných skupin hodnot či výrazů (počet očekávaných argumentů je potřebné znát) v pořadí zleva doprava. Výrazy jsou před použitím funkcí redukovány na hodnoty.



 Důsledkem tohoto přístupu je to, že výsledkem výrazu

square-root 16 + square-root 16

není 8, jak by mnozí očekávali, ale 4.47213595499958, protože Red vidí něco jako toto:



čímž je funkce pro jeden argument, stojící před skupinou s infixovým operátorem (majícím přednost); operandy operátoru (+) jsou číslo a funkce, která se před použitím operátorem vyhodnotí.

Abychom získali oněch intuitivních 8, musíme použít závorky:

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

Alternativní sekvenci

>> **square-root add 16 square-root 16**   
== 4.47213595499958

můžeme číst jako

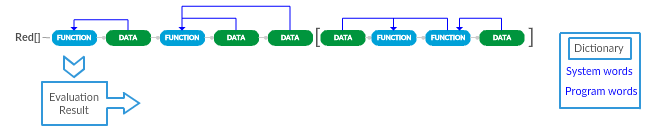
>> square-root (add 16 (square-root 16)) --> 4.47213595499958

Jiný příklad, kombinující infixový operátor a jemu odpovídající funkci:

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



**Poněkud zjednodušený pohled na tok výpočtu:**



Note: The function that picks data from before it (the third from left to right) refers to infix operators like "+", "-" , "\*" , "/" etc.

**Jiné vysvětlení:**

Nenad Rakocevic ve svém přehledu Red/System Language Specification ([red-lang.org -](https://static.red-lang.org/red-system-specs.html#section-5.2)září 2017) uvádí toto stručné vysvětlení:

Výrazy se vyhodnocují zleva doprava. Operátory (infixové funkce) mají stejné pořadí důležitosti, kromě toho, že mají přednost před prefixovými funkcemi.

*Created with the Standard Edition of HelpNDoc:* [*Easily create Web Help sites*](https://www.helpndoc.com/feature-tour)

**Zvýraznění kódu**

**Zvýraznění textu ve skriptu**

Zvýraznění syntaxe je pro začátečníky velmi prospěšné - zejména v programovacím jazyce Red, s jeho mnoha předdefinovanými slovy a stručným kódem. Kdykoli je to možné, používám zvýrazněný kód převzatý z Notepad++.

Red []

; mezera kvůli přehlednosti

a: "Hello"

b: 123

c: [33 "fox"]

**print** c

Práce v konzole je prezentována s tmavým pozadím:

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

Uživatelský vstup je prezentován tučným písmem, odezva interpreta normálním. To pomůže předejít zbytečným chybám při kopírování kódu z prezentovaných příkladů.

Pro lepší přehlednost vkládám prázdné řádky mezi jednotlivé příkazy. Růžové komentáře jsou přidány při poslední editaci textu.

>> **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 ; barevně jsou označeny pozdější komentáře a doplňky

*Created with the Standard Edition of HelpNDoc:* [*Full-featured Help generator*](https://www.helpndoc.com/feature-tour)

**Konzola - vstup a výstup**

**Poznámka**: vstup do ~ a výstup z konzoly může být při kompilování programů problematický, neboť při kompilaci je konzola mimo provoz.

native!**print**

print  je funkce, která posílá data do konzoly. Za daty ještě do konzoly posílá znak newline , který je při výstupu respektován.

Red []

**print** "hello"

**print** 33

**print** 3 + 5

hello

33

8

native!**prin**

prin  také posílá data do konzoly ale bez znaku newline, takže vytiskne všechna zadaná data na jeden řádek.

Red []

**prin** "Hello"

**prin** "World"

**prin** 42

HelloWorld42

function!**probe**

probe  vrací svůj argument bez vyhodnocení, na rozdíl od funkce print, která svůj argument vyhodnotí; probe může být použit při ladění k zobrazení nezměněného kódu.

>> **print [3 + 2]**   
5   
  
>> **probe [3 + 2]**  
[3 + 2]  
== [3 + 2]  
  
>> **print probe [3 + 2]**  
[3 + 2]  
5

Popsáno také [zde](#_topic_Formatovanidat), za příkazem  mold.

function!**input**

Vloží  **string** (řetězec) z konzoly. Notice that any number typed on console are converted to a string.newline character is removed.

Red []

**prin** "Enter a name: "

name: **input** ;-- John --> "John"

**print** [name "is" **length?** name "characters long"]

John

John is 4 characters long

routine!**ask**

Totéž jako input ale zobrazí se řetězec.

Red []

name: **ask** "What is your name: "

**prin** "Your name is "

**print** name

What is your name: John

Your name is John

*Created with the Standard Edition of HelpNDoc:* [*Free PDF documentation generator*](https://www.helpndoc.com)

**Exekuce kódu**

**Spuštění exekuce**

Svůj skript můžete uložit jako soubor s příponou ~.red nebo ~.txt a spustit jej z příkazového řádku jako argument aplikace Red, asi takto:

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

To spustí interpret Redu, otevře konzolu (REPL) a spustí váš skript.

V běžícím prostředí Redu však můžete provést exekuci kódu přímo příkazem do.

native!**do**

Vyhodnotí argumenty kódu, jinými slovy provede exekuci kódu. Argumentem může být blok, soubor, funkce či jiná hodnota.

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

Před dalším pokračováním nahlédněte do kapitoly [Soubory](#_topic_Soubory).

Pokud jste například uložil skript jako myprogram.red, můžete jej realizovat tímto příkazem:

>> **do %myprogram.red**

Nezapomeňte, že v tomto případě musí být interpret Redu a zdrojový soubor ve stejné složce; jinak musíte uvest také cestu ke skriptu.

Můžete také načíst skript do interpreta Redu:

>> **a: load %myprogram.red**

a spustit jej:

>> **do a**

Můžete také načíst a provést skript, uložený jako textový soubor :

>> **do load %myprogram.txt**

 Uvědomte si, že toto vše můžete provádět  **z vnitřku programu Red!**

**Zastavení exekuce**

function! **quit**

Zastaví vyhodnocení a opustí program.

Zapíšete-li to v konzole GUI, zavře se. Zapíšete-li to na příkazový řádek interaktivního rozhraní, pouze opustíte interpreta Redu.

**/return** Zastaví vyhodnocení a opustí program ... ;; koketuje s OS, opatrně!

quit/return 3 ;hands the value 3 to the Operating System

function! **halt**

Zastaví interpretaci skriptu.

routine! **quit-return**  ;; koketuje s OS, opatrně!

Stops evaluation and exits the program with a given status. Seems to me as exactly the same as quit/return, but it's a routine! datatype, not a function! Go figure out.

VID DLS **on-close**

Aktivita řízená událostí - spustí zadaný kód po zavření aplikace (okna GUI). Spusťte následující program. Vytvoříte tlačítko a když na něj kliknete, tlačítko zmizí a v konzole se má objevit text "bye!". ;; ale nechodí a nechodí

Red [needs: **view**]

**view** [

on-close [**print** "bye!"]

button [**print** "click"]

]

**Control C**

Dvojklik Ctrl+c zastaví exekuci a opustí interpreta Redu.

*Created with the Standard Edition of HelpNDoc:* [*Single source CHM, PDF, DOC and HTML Help creation*](https://www.helpndoc.com/help-authoring-tool)

**Datové typy**

Každá hodnota Redu má svůj specifický **datový typ**, který určuje rozsah přípustných hodnot, přípustné operace a způsob uložení hodnot v paměti. Datových typů je aktuálně 46, včetně datového typu datatype! Výpis datových typů získáme dotazem:

>> **? datatype!**

Z těchto 46 datových typů je vytvořeno 16 skupin (typesetů), majících společné vlastnosti. Některé datové typy se vyskytují ve více skupinách. Výpis těchto skupin získáme dotazem:

>> **? typeset!**

Jsou to tyto tyto skupiny: any-(type!, object!, string!, word!, function!, block!, path!, list!), series!, number!, scalar!, immediate!, internal!, external!, default!, all-word!.

**Jednotlivé datové typy:**

**word!**

Základní datový typ. Slova jsou klíčovým pojmem v jazyce Red. Vyskytují se ve čtveré typové formě:

word - typ **word!**, základní formát pro přirozenou hodnotu slova

word: - typ **set-word!**, formát pro přiřazení hodnoty ke slovu

:word - typ **get-word!**, odkaz na hodnotu slova (bez vyhodnocení)

'word - typ **lit-word!**, odkaz na slovo jako symbol

**block!**

Bloky jsou skupiny hodnot a slov. Používají se všude, od samotného skriptu až po bloky dat a bloky kódu ve skriptu.

>> **bl: [123 data "zdar"]** ; blok s daty  
== [123 data "zdar"]

>> **[]** ; prázdný blok  
== []

**paren!**

Datový typ paren! je kolektor, ohraničený kulatými závorkami. Používá se hlavně k vyjádření preference aritmetických operací.

Objekt typu paren! se vytvoří příkazem make paren! :

>> **prn: make paren! ["a" 5]**  
== ("a" 5)

**none!**

Ekvivalent "nuly" v jiných programovacích jazycích. Indikuje neexistující data.

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

**logic!**

Kromě klasických true a false, používá Red také on , off, yes a no jako hodnoty datového typu  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!**

Řada (series) znaků uvnitř uvozovek nebo složených závorek. Zabírá-li řetězec více než jeden řádek, je použití {} jedině možné.

Manipulace s řetězci se provádí s použitím příkazů, popsaných v [kapitolách o řadách](#_topic_Blokyarady).

>> **a: "my string"**   
== "my string"

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

**>> a: {my**   
{ **string}** ; první "{" je signál konzoly!  
== "my^/string"   
>> **print a**   
my   
string

>> **a: "my new** ;trying to span over more than one line   
\*\*\* Syntax Error: invalid value at {"my new}

**char!**

Uvedena znakem # a uvnitř uvozovek, představuje hodnota typu char! kódový bod Unicodu. Jsou to celá čísla v rozsahu hexadecimal 00 po hexadecimal 10FFFF (0 až 1,114,111).

#"A" je char!

"A" je string!

Umožňuje určité matematické operace.

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

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

**integer!**

32 bitová celá signovaná čísla. Od −2,147,483,648 po 2,147,483,647. Je-li číslo mimo tento rozsah, přiřadí mu Red datový typ float!.

Poznámka: Dělení 2 celých čísel dává oseknutý výsledek:

>> **7 / 2**   
== 3

**float!**

64 bitová desetinná čísla. Prezentují se s desetinnou tečkou či čárkou nebo e-notací.

>> **7.0 / 2**   
== 3.5

>> **3e2**   
== 300.0

>> **6.0 / 7**   
== 0.8571428571428571

**file!**

Název je uveden znakem %. Nepoužíváte-li aktuální cestu, máte uvést cestu v uvozovkách. Cesta používá lomítka (/); zpětná lomítka (\) jsou ve Windows automaticky konvertována na obyčejná lomítka.

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

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

**path!**

Používá se k přístupu k interním položkám větších struktur pomocí lomítka (/). Používá se v různých situacích, například:

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

Lomítka "/" se také používají při přístupu k objektům a k upřesněním.

**time!**

Čas je vyjádřen ve formátu hours:minutes:seconds.subseconds. Všimněte si, že sekundy a subsekundy jsou odděleny tečkou, nikoliv dvojtečkou. Ke každé části je možný přístup přes upřesnění (refinement). Nahlédněte do kapitoly [Práce s časem](#_topic_Pracescasem).

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

>> **a: now/time/precise**   
== 22:05:46.805   
>> **type? a**  
== time! ; datový typ objektu **a** je time!  
>> **a/hour**  
== 22  
>> **a/minute**  
== 5  
>> **a/second**  
== 46.805 ; datový typ a/second je float!

**date!**

Red akceptuje datumy v různých formátech:

>> **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** ; zkuste si totéž pro první a druhé tisíciletí   
31-Oct-2017

Red rovněž ověřuje platnost datumů, včetně s uvážením přechodných let. Na den, měsíc či rok se lze odkázat upřesněním:

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

**point!** a **pair!**

Point! a pair! se zdají být úplně stejné. Uvádějí celočíselné souřadnice v kartézském souřadnicovém systému (x y axys). K typu point! nejsou informace

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

**percent!**

Prezentují se přidáním znaku "%" za číslem.

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

**tuple!**

Tuple (entice) je výpis tří až dvanácti celých čísel o velikosti 0 až 255, oddělených tečkami. Entice jsou užitečné pro prezentaci čísel verzí, IP adres a barev (např.: 0.255.0).

S enticí lze provádět tyto operace: random, add, divide, multiply, remainder, subtract, and, or, xor, length?, pick (nikoliv 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

Úplný přehled barev ve vyjádření typu word! a tuple! získáme zadáním následujícího dotazu:

>> **? tuple!**  
 red 255.0.0 ; následuje výpis 45 barev

*Created with the Standard Edition of HelpNDoc:* [*Create HTML Help, DOC, PDF and print manuals from 1 single source*](https://www.helpndoc.com/help-authoring-tool)

**Další datové typy:**

**issue!**

Datový typ issue! (emise) je sekvence znaků, použitelná pro vyjádření telefonního nebo seriového čísla či čísla kreditní karty. Emise začínají znakem (#) a pokračují až k prvnímu oddělovači (např. k mezeře). Uvnitř emise lze použít většinu znaků s výjimkou lomítka "/".

>> **a: #333-444-555-999**   
== #333-444-555-999   
  
>> **a: #34-Ab.77-14**   
== #34-Ab.77-14

**url!**

Schematické složení výrazu tupu URL je toto: <*protocol*>**://**<*path*>. První část uvádí komunikační protokol (http, ftp, nntp, mailto, file, ...), za lomítky je cesta k odkazované entitě.

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

**email!**

Datový typ pro uvedení emailové adresy. Zápis musí obsahovat znak @.

>> **a: myname@mysite.org**   
== myname@mysite.org   
  
>> **type? a**   
== email!

**image!**

Datový typ image! je řada, která obsahuje RGB zobrazení. Podporované formáty jsou GIF, JPEG, PNG and BMP. Obrazy se obvykle načítají ze souboru.:

>> **a: make image! [30x40 #{00A2E800A2E800A2E800A** ... **} ]**    
>> **a: load %heart.bmp**   
== make image! [30x20 #{00A2E800A2E800A2E800A ... } }

; S načteným objektem typu image lze manipulovat jako s řadou:   
  
>> **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

**refinement!**

Neboli **upřesnění**; indikuje variaci (za lomítkem) v použití nebo rozšíření významu funkce, objektu, jména souboru, URL nebo cesty.

>> **block: [1 2]**   
>> **append/only block (3 4)**   
== [1 2 3 4]

**action!**

Datový typ, zahrnující cca 52 funkcí, například  add , take , append, negate, ...

>> **action? :take** ; Dvojtečka je povinná.   
== true

Seznam všech funkcí typu action! získáme příkazem

>> **? action!**

**op!**

Datový typ, zahrnující funkce, které pracují jako infixové operátory, jako  + nebo \*\*.

**routine!**

Používá se pro připojení externího kódu.

**binary!**

Binární řetězce se zapisují do složených závorek s předsazeným znakem #. V tomto formátu lze uložit libovolnou sekvenci bajtů (image, audio, spustitelný soubor, komprimovaná či zašifrovaná data).

Formát binárních dat může být číslo se základem 2 (binár), 16 (hexadecimál) nebo 64 (tetrahexagesimál). Nepřítomnost číslice avizuje implicitní hexadecimální kódování se základem 16.

Příklad:

#{3A1F5A} ; báze 16

2#{01000101101010} ; báze 2

64#{0aGvXmgUkVCu} ; báze 64

**event!**

Tento datový typ pro událost se používá v systému Red/GUI, viz  [GUI - Introspekce událostí](#_topic_Introspekceudalosti).

**function!**

**object!**

**handle!**

**unset!**

**tag!**

**lit-path!**

**set-path!**

**get-path!**

**bitset!**

**typeset!**

**error!**

**native!**

*Created with the Standard Edition of HelpNDoc:* [*News and information about help authoring tools and software*](https://www.helpauthoringsoftware.com)

**Hash! vector! a map!**

Tyto datové typy si zasluhují samostatnou kapitolu. Mohou výrazně zlepšit kvalitu a rychlost práce.

Hash! a vector! jsou výkonné řady, zejména u velkých sestav.

Doporučuji před dalším čtením nahlédnout do kapitol [Bloky a řady](#_topic_Blokyarady).

**hash!**

hash! je řada, upravená pro rychlejší vyhledávání. Protože "hašování" spotřebovává zdroje, nemá cenu vytvářet hash! pro řady, které budou prohledávány jenom několikrát. Pokud však má být vaše řada prohledávaná často, vyplatí se z ní vytvořit hash! . Webová stránka Rebolu tvrdí, že vyhledávání může být 650 krát rychlejší než u normálních řad.

>> **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

Jinak to jsou řady, jejichž hodnoty jsou interně vnímané jako skupiny dvojic.

**vector!**

Vektory jsou výkonné řady položek typu integer! ,float!, char! nebo percent!.

K vytvoření vektoru použijete příkaz make vector!

Zatímco  **hash!** umožňuje rychlejší vyhledávání, **vector!** umožňuje rychlejší provádění matematických operací, noboť jsou prováděny pro celou řadu najednou.

>> **a: make vector! [33 44 52]**   
== make vector! [33 44 52]   
  
>> **print a**   
33 44 52   
  
>> **print a \* 8**   
264 352 416

Všimněte si, že to nejde udělat s blokem, patřícím rovněž do skupiny series!:

>> **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!**

Mapy jsou vysoce výkonné slovníky, které sdružují klíče s hodnotami (key1: val1 key2: val2 ... key3: val3).

Mapy **nejsou** řadami. Nelze u nich uplatnit většinu příkazů pro řady (series).

Pro vyjmutí hodnoty z mapy použijeme příkaz select a pro zadání hodnoty použijeme specielní akci: 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

*Created with the Standard Edition of HelpNDoc:* [*Easily create Help documents*](https://www.helpndoc.com/feature-tour)

**Konverze datových typů:**

action!**to**

Konvertuje jeden datový typ na druhý, např.  integer! na string!, float! na integer! a dokonce string! na number!.

>> **to integer! 3.4**   
== 3

>> **to float! 23**   
== 23.0

>> **to string! 23.2**   
== "23.2"

>> **to integer! "34"**   
== 34

function!**to-time**

Konvertuje hodnoty na datový typ 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**

Konvertuje dvě čísla typu integer! nebo float! na pair!

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

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

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

function!**to-binary**

Konvertuje argument na hodnotu typu binary!.

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

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

*Created with the Standard Edition of HelpNDoc:* [*Easy to use tool to create HTML Help files and Help web sites*](https://www.helpndoc.com/help-authoring-tool)

**Bloky a řady**

**Bloky**

Datový typ block! patří do typových skupin (typeset!) **series**, **any-block** a **any-list**.

Bloky jsou skupiny hodnot a slov, ohraničené hranatými závorkami, napříkad: [one block], [another block [block within a block]]. Hodnoty a výrazy v bloku se implicitně nevyhodnocují. Vyhodnocení bloku se evokuje funkcemi do a reduce viz [Datový typ block!](https://doc.red-lang.org/cs/block.html).

Blok lze vytvořit výpisem závorek a hodnot, například (literálová skladba):

[one 2 "three"]  
[print 1.23]

nebo funkcemi make či to (konstruktorová skladba):

>> **make block! 10**  
== [] ; alokuje prostor pro 10 prvků

>> **to block! {one 2 4:00}**  
== [one 2 4:00:00] ; konvertuje string! na block!

**Řady (series!)**

Slovo **series**! je označení typové skupiny (typeset!), skládající se z těchto typů: block!, paren!, string!, file!, url!, path!, vector!, hash!, binary!, tag!, email!, image!.

Obecně je **řada** (serie) uspořádaná sada hodnot různých typů. Prvkem serie může být cokoli z repertoáru: data, slova, funkce, objekty, a jiné serie.

**Pole - arrays**

Pole (array) netvoří zavedený datový typ. Je to kolekce indexovaných (viz [Navigace řadami](#_topic_Navigaceradami)) hodnot stejného typu. Jednorozměrné pole se označuje jako **řádková matice** (vektor), dvojrozměrné pole definuje **matici**. Třírozměrné pole se někdy označuje jako tenzor.

Řádková matice je tvořena jednoduchým blokem, matici tvoří blok řádkových matic, více rozměrové pole lze vyjádřit jako matici hodnot typu tuple!

Zde je příklad pole o rozměrech 2 x 3:

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

Pro přístup k jeho elementům lze použít lomítko:

>> **a/1**  
== [1 2]  
>> **a/1/1**  
== 1  
>> **a/3/2**  
== 6

Nebo slovní označení indexu:

>> **third a**  
== [5 6]

*Created with the Standard Edition of HelpNDoc:* [*Free EPub producer*](https://www.helpndoc.com/create-epub-ebooks)

**Navigace řadami**

* První element řady se nazývá **head** (čelo). Jak uvidíme, nemusí to být vždy nutně první element;
* Za posledním elementem řady je něco, čemu se říká **tail** (chvost). Je to místo bez hodnoty.
* Každá řada má "**entry index**", jehož nejlepší definicí by mohlo být: "místo, kde začíná použitelná část řady". Mnohé operace s řadami mají tento  **"**entry index**"** jako startovní bod. **Entry index** lze posouvat vzad i vpřed s vlivem na výsledek operace.
* Každý element řady je označen číselným indexem, začínajícím 1 (not zero!) u prvního elementu.
* Počínaje pozicí **entry indexu**, mají elementy slovní označení : "first" pro první, "second" pro druhý a tak dál až po "fifth".

Poznámka: Pojem "**entry index**" jsem si vymyslel, neboť v dokumentaci není. Tam je "**entry index**" označován pouze jako "index", což se podle mého názoru může plést s "číslem indexu".

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

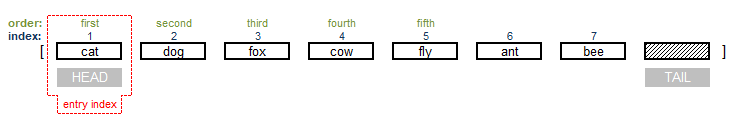
Tyto příkazy vrací informaci o pozici **entry indexu**. Je-li entry index v čele řady, dotaz  head? vrací true, jinak false. Tatáž logika platí pro tail?. Dotaz index? vrací číslo indexu místa, kde se aktuálně nachází **entry index**.

Následující příklady vysvětlení objasní.

Vytvořme řadu **s,**obsahující řetězce**"cat" "dog" "fox" "cow" "fly" "ant" "bee"**:

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

Dostaneme něco, co vypadá asi takto:



>> **head? s**   
== true

>> **index? s**   
== 1

>> **print first s**   
cat

action!**head** action! **tail**

Příkaz head přemístí **entry index** k prvnímu elementu řady, na její čelo.

Příkaz tail přemístí **entry index** na pozici za posledním elementem řady, do chvostu.

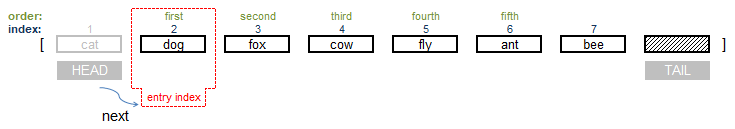
Umístění head a tail samo o sobě řadu nemění, pouze head  **vrací** celou řadu a tailnevrací nic.

action!**next**

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

Příkaz next přemístí **entry index** o jeden element směrem ke chvostu. Všimněte si, že next pouze **vrací** změněnou řadu ale nemění ji. Opakované použití příkazu next nezpůsobí další posun vstupního indexu, protože příkaz bychom aplikovali na původní, nezměněné řadě.

Nyní tedy máme:



>> **print s**   
dog fox cow fly ant bee   
  
>> **head? s**   
== false ; entry index není v čele   
  
>> **print first s** ; s pozicí entry indexu se posouvají i aliasy  
dog   
  
>> **index? s** ; indexování elementů se však nemění  
== 2

action!**back**

Příkaz back je opakem příkazu next: přemístí **entry index** o jeden element směrem k čelu. Provedete-li back v naší řadě **s,** vrátí se "cat" zpátky na scénu! Nikdy nebylo zapomenuto!

To znamená, že Red staré **s** neodhodil. Je to jedna ze zvláštností Redu: data zůstávají trvale vnořena v kódu.

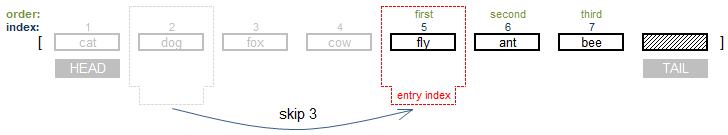
Poté, co jste přemístil index v naší řadě **s** vpřed a přiřadil ji k jinému slovu, např.  **b** (b: s), můžete provádět přesuny indexu vzad i vpřed  pro **b** a dobývat "skryté" hodnoty **s,** protože **b** ukazuje na stejná data jako **s**.

Chcete-li vytvořit nezávislou kopii řady, musíte použít příkaz  copy.

Jak jsem se již zmínil dříve, v Redu jsou (na rozdíl od jiných jazyků) proměnné (slova) přiřazeny k datům a nikoliv naopak.

action! **skip**

Přemístí **entry index** o daný počet elementů k chvostu.



>> **s: skip s 3**   
== ["fly" "ant" "bee"]   
  
>> **print s**   
fly ant bee   
  
>> **print first s**   
fly   
  
>> **print index? s**   
5

Je-li skok delší než počet elementů, zůstane **entry index** ve chvostu.

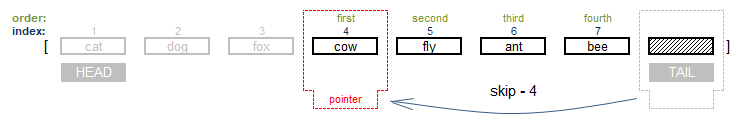
>> **s: skip s 100**   
== []



>> **tail? s**   
== true   
  
>> **index? s**   
== 8

Viditelnost elementů řady lze obnovit zpětnými skoky:

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



>> **print first s**   
cow   
  
>> **print index? s**   
4

*Created with the Standard Edition of HelpNDoc:* [*Create help files for the Qt Help Framework*](https://www.helpndoc.com/feature-tour/create-help-files-for-the-qt-help-framework)

**Series "getters"**

Existuje takové množství příkazů pro manipulaci s řadami (series)se, že jsem je rozdělil do dvou kapitol - jednu pro příkazy získávající informace z řad (getters) a druhou pro příkazy, které přímo mění řady (changers).

Příkazy "getters" pouze vracejí hodnoty, bez změny řad. Nicméně i tyto příkazy mohou měnit řady, pokud přiřadíme řadu k vratné hodnotě.

action!**length?**

Vrací velikost (délku) řady od aktuálního indexu ke konci.

>> **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** ;see the command "find"  
== 3 ;from "13" to the tail there are 3 elements

function!**empty?**

Vrací true , je-li řada prázdná, jinak vrací  false.

>> **a: [3 4 5]**   
== [3 4 5]   
  
>> **empty? a**   
== false   
  
>> **b:[]**   
== []   
  
>> **empty? b**   
== true

action!**pick**

Vybere hodnotu z řady na pozici, dané druhým argumentem.



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

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

action!**at**

Vrací řadu od zadaného indexu.

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

action!**select** aaction! **find**

Oba příkazy vyhledají zadanou hodnotu v řadě. Hledání probíhá zleva doprava, není-li použito upřesnění  /reverse  nebo  /last.

Při nalezení shody:

* select vrací následující element řady za shodou;

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

* find vrací celý zbytek řady od shody ke chvostu.

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

**/part**

Omezí délku prohledávané řady na daný počet elementů; v zobrazení dole je prohledávaná oblast zvýrazněna:

>>**select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"] 3**   
== none   
  
>> **select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["fox"] 3**    
== "cow" ; why?!

>> **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**

Hledá pouze blok uvnitř prohledávané řady.

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

**/case**

Pokyn pro rozlišování malých a velkých písmen.

**/skip**  [*řada skupin*] [*shoda*][*délka skupiny*]

Považuje řadu za sled skupin zadané délky. Požadovaná shoda se musí nalézat na prvním místě skupiny.

>> **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"]

Skupiny jsou označeny žlutě a hledaná shoda modře. Vrací zbytek bloku včetně shody.

**/last**

Vyhledá poslední shodu se zadaným elementem; vrátí zbytek řady.

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

**/reverse**

The same as /last , but from the current index that can be set, for example by the at command.

**find/tail**

Normálně vrací find výsledek včetně shodujícího se elementu. S upřesněním /tail se vrací zbytek řady za shodným elementem, podobně jako u  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**

Upřesnění **/match** porovnává klíč s počátkem řady. Případnou vracenou hodnotou je zbytek řady za shodou.

>> **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**

Vrací poslední hodnotu řady.

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

function!**extract**

Extrahue hodnoty z řady v zadaném intervalu, vracejíc novou řadu.

>> **extract [1 2 3 4 5 6 7 8 9] 3**   
== [1 4 7]   
  
>> **extract "abcdefghij" 2**   
== "acegi"

**/index**

Extrahuje hodnoty v zadaném intervalu od zadané pozice (indexem).

>> **extract/index "abcdefghij" 2 4** ; interval index   
== "dfhj"

**/into**

Připojí extrahované hodoty k zadané řadě (akumulátoru).

>> **akum: []** ;creates empty series   
== []   
  
>> **extract/into "abcdefg" 2 akum**   
== [#"a" #"c" #"e" #"g"]   
  
>> **extract/into ["cat" "dog" "fox" "cow" "fly"] 2 akum**   
== [#"a" #"c" #"e" #"g" "cat" "fox" "fly"]

action!**copy**

Viz kapitola [Kopírování](#_topic_Kopirovani).

native!**union**

Vrací spojení dvou řad. Duplikátní elementy jsou zařazeny pouze jednou.

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

**/case**

Case-sensitive porovnávání

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

**/skip**

Považuje řady za sled skupin zadané délky. Porovnává se pouze první element každé skupiny. U duplicitních vstupů se zachovává záznam první řady:

>> **union/skip** [**1 2 3** **4 5 6** **7 8**] [**1 8 5** **1 2**] 3   
== [1 2 3 4 5 6 7 8]

>> **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**

Vypouští duplicitní výskyty elementů z obou řad.

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

**/case**

Case-sensitive porovnávání

**/skip**

Považuje řady za sled skupin zadané délky.

native!**intersect**

Vrací pouze duplicitní výskyty elementů:

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

**/case**

Case-sensitive porovnávání

**/skip**

Považuje řady za sled skupin zadané délky

native!**unique**

Vypouští duplikáty:

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

**/skip**

Považuje řady za sled skupin zadané délky

native!**exclude**

Vypouští zadané elementy z řady a vrací seznam neopakovaných elementů.

>> **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] ; nemění původní hodnotu řady

>> **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**

Case-sensitive porovnávání

**/skip**

Považuje řady za sled skupin zadané délky

*Created with the Standard Edition of HelpNDoc:* [*Create help files for the Qt Help Framework*](https://www.helpndoc.com/feature-tour/create-help-files-for-the-qt-help-framework)

**Series "changers"**

**Tyto příkazy mění původní řady !**

action!**clear**

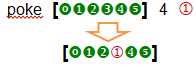
Smaže všechny elementy řady.

Pouhé přiřazení prázdného řetězce (" ") nebo nuly nemusí mít očekávaný účinek. Pamatování "věcí" realizuje Red mnohdy neočekávanými způsoby. Skutečné smazání řady zajistí příkaz  clear.

>> **a: [11 22 33 "cat"]**   
== [11 22 33 "cat"]   
  
>> **clear a**   
== []   
  
>> **a**   
== []

action!**poke**

Změní hodnotu elementu řady v pozici dané druhým argumentem na hodnotu třetího argumentu.

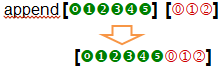


>> **x: ["cat" "dog" "mouse" "fly"]**   
== ["cat" "dog" "mouse" "fly"]   
  
>> **poke x 3 "BULL"**   
== "BULL"   
  
>> **x**   
== ["cat" "dog" "BULL" "fly"]

>> **s: "abcdefghijklmn"**   
== "abcdefghijklmn"   
  
>> **poke s 4 #"W"**   
== #"W"   
  
>> **s**   
== "abcWefghijklmn"

action!**append**

Vloží elementy druhého argumentu na konec řady.



>> **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**

Omezuje počet připojovaných elementů.

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

**/only**

Připojí připojovanou řadu jako vnořený blok.

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

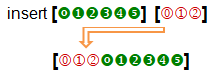
**/dup**

Připojí vícekrát (dup) druhý argument k prvnímu.

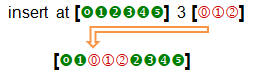
>> **append/dup ["a" "b" "c"] ["A" "B"] 3**   
== ["a" "b" "c" "A" "B" "A" "B" "A" "B"]

action!**insert**

Vloží elementy druhého argumentu v místě aktuálního entry-indexu (obvykle to je počátek řady). Zatímco append vrací řadu od čela, insert ji vrací až za místem vložení. To umožňuje řetězit vícero operací  insert, nebo počítat délku vložené části.



>> **a: "abcdefgh"**   
== "abcdefgh"   
  
>> **insert a "OOO"**   
== "abcdefgh"   
  
>> **a**   
== "OOOabcdefgh"



>> **a: "abcdefgh"**   
== "abcdefgh"   
  
>> **insert at a 3 "OOO"**   
== "cdefgh"   
  
>> **a**   
== "abOOOcdefgh"

**/part**

Vloží pouze zadaný počet vkládaných elementů.

**/only**

Umožňuje vložení elementů jako blok.

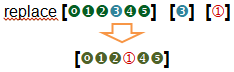
**/dup**

Umožňuje opakované vložení druhého argumentu v počtu, daném třetím argumentem.

>> **a: "abcdefg"**   
== "abcdefg"   
  
>> **insert/dup a "XYZ" 3**   
== "abcdefg"   
  
>> **a**   
== "XYZXYZXYZabcdefg"

function!**replace**

Nahradit uvedený element danou hodnotou. Provede se jen pro první výskyt.



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

**/all**

Nahradí se všechny výskyty

>> **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**

Třídí řadu podle velikosti hodnot.



>> **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**

Provede case-sensitive třídění.

**/skip**

Zachází s řadou jako s pořadím skupin zadané délky.

**/compare**

Pro definovaný komparátor (offset, blok, funkce) zadáme porovnávané elementy.

>> **names: ["Larry" "Curly" "Mo" ]**   
== ["Larry" "Curly" "Mo" ]

>> **sort/compare names function [a b] [a > b]** ; zajímavá diskrepance!

== ["Mo" "Larry" "Curly"]

**/part**

Třídí pouze části řad

**/all**

Porovná všechna vybraná pole. Používá se s upřesněním 'skip' pro určení velikosti skupin.

**/reverse**

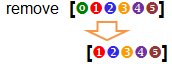
Obrátit pořadí

**/stable**

Místo někdy nestabilního třídění Quicksort se použije stabilnější ale pomalejší algoritmus Merge.

action!**remove**

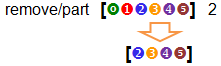
Odebere první hodnotu řady.



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

**/part**

Odebere zadaný počet elementů.



>> **s: "abcdefghij"**   
== "abcdefghij"   
  
>> **remove/part s 4**   
== "efghij"

Totéž lze provést příkazem remove at [0 1 2 3 4 5] 2

native!**remove-each**

Podobně jako foreach, postupně vyhodnotí blok pro každý element řady. Vrací-li blok hodnotu true, je element odebrán z řady:

Red []

a: ["dog" 23 3.5 "house" 45]

**remove-each** i a [**string?** i] ; odebere všechny řetězce

**print** a

23 3.5 45

Red []

a: " my house in the middle of our street"

**remove-each** i a [i = #" "] ; odebere všechny mezery

**print** a

myhouseinthemiddleofourstreet

action!**take**

Odebere **první** element z řady a vrátí jej jako vratnou hodnotu(return).

>> **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**

Odebere **poslední** element z řady a vrátí jej jako vratnou hodnotu(return).

>> **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 and append can be used to perform stack (queue) operations.

**/part**

Odebere daný počet elementů z počátku řady a vrátí je jako  **return**.

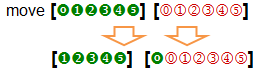
>> **s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]**   
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]   
  
>> **take/part s 3**   
== ["cat" "dog" "fox"]   
  
>> **s**   
== ["cow" "fly" "ant" "bee"]

**/deep**

Kopíruje vnořené hodnoty.

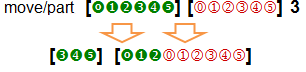
action! **move**

Přemístí jeden (implicitně) nebo více elementů řady na jinou pozici téže nebo jiné řady. Mění oba vstupní argumenty.



**/part**

Přemístí zadaný počet elementů



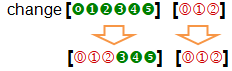
>> **a: [a b c d]**   
== [a b c d]   
  
>> **b: [1 2 3 4]**   
== [1 2 3 4]   
  
>> **move a b**   
== [b c d]  
   
>> **a**   
== [b c d]   
  
>> **b**   
== [a 1 2 3 4]   
  
>> **move/part a b 2**   
== [d]   
  
>> **a**   
== [d]   
  
>> **b**   
== [b c a 1 2 3 4]

Příkaz move může být kombinován s jinými příkazy pro přemístění uvnitř jediné řady. Na příklad:

>> **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**

Změní počáteční elementy řady a vrací řadu po změně. Mění pouze první řadu.



>> **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**

Upraví měněnou délku řady vložením hodnot z jiné řady.

>> **a: [1 2 3 4 5]**   
== [1 2 3 4 5]   
  
>> **change/part a ["a" "b"] 3**   
== [4 5]  
   
>> **a**   
== ["a" "b" 4 5]

**/only**

Nahradí první element blokem.

>> **change/only s: [1 2 3 4 5] [1 2]**   
== [2 3 4 5]  
   
>> **s**   
== [[1 2] 2 3 4 5]

**/dup**  
Provede zadaný počet duplikátů změny.

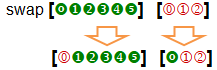
function!**alter**

Připojí element k řadě nebo jej z řady odebere. Pokud  alter  zadaný element v řadě nenalezne připojí jej a vrátí true. Pokud jej nalezne, odebere jej a vrátí 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**

Prohodí první elementy dvou řad. Vrací první řadu ale mění obě:



>> **a: [1 2 3 4] b: [a b c d]**   
== [a b c d]   
  
>> **swap a b**   
== [a 2 3 4]   
  
>> **a**   
== [a 2 3 4]   
  
>> **b**   
== [1 b c d]

S použitím slova find lze prohodit libovolný element dvou řad nebo elementy téže řady:

>> **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**

Převrátí pořadí elementů v řadě:

>> **reverse [1 2 3]**   
== [3 2 1]   
  
>> **reverse "abcde"**   
== "edcba"

**/part** omezí akci reverse na zadaný počet elementů:

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

*Created with the Standard Edition of HelpNDoc:* [*Full-featured Kindle eBooks generator*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Přístup a formátování dat**

native! **get**

Každé slovo Redu, nativní i uživatelsky vytvořené, je umístěno ve slovníku. Je-li slovo sdruženo s výrazem, je uloženo i s přidruženým výrazem, který se vyhodnotí v závislosti na typu volání slova.

Chcete-li znát slovníkový popis slova, použijete příkaz  get . Uvědomte si, že když v Redu odkazujete na slovo samotné (nikoliv jeho hodnotu), předznamenáte slovo apostrofem ('). Příkaz  get poskytuje význam i nativních slov ale vrací chybové hlášení, je-li použit pro konkretní hodnotu, např. typu  integer!, pair! či 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**

Příkaz mold přemění zadanou hodnotu (např. typu block!, integer!, series! etc.) na řetězec a **vrátí jej**:

>> **type? 8**   
== integer!   
  
>> **type? mold 8**   
== string!   
  
>> **print [4 + 2]**   
6   
  
>> **print mold [4 + 2]**   
[4 + 2]

**Refinements:**

**/only** - Vyloučí vnější hranaté závorky bloku  
**/all** - Vrací hodnotu v načtení schopném formátu  
**/flat** - Vyloučí veškerou identaci  
**/part** - Omezí délku výsledku na zadanou hodnotu

action!**form**

Příkaz form  také přemění zadanou hodnotu na řetězec ale v závislosti na typu, nemusí výsledný text obsahovat extra informace o typu (jako [ ] { } a ""), jako u příkazu mold. Užitečné pro [Manipulace s textem](#_topic_Manipulacestextem).

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

Umožňuje použití refinementu /part k omezení počtu znaků.

**Hlavní použití pro mold a form:**

mold se hlavně používá pro přeměnu řad na kód, který může být uložen a interpretován později

form se hlavně používá pro generování obyčejného textu z řad

>> **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**

Příkaz probe vytiskne svůj argument bez vyhodnocení a také jej vrátí. Lze jej s výhodou použít při ladění. Vzpomeňte si, že příkaz print svůj argument vyhodnocuje.

>> **print [3 + 2]**   
5   
  
>> **probe [3 + 2] [3 + 2]**   
== [3 + 2]  
  
>> **print probe [3 + 2]**  
[3 + 2]  
5

native!**reduce**

Vyhodnocuje výrazy uvnitř bloku a vrací nový blok s vyhodnocenými hodnotami. Nahlédněte do kapitoly  [Vyhodnocení](#_topic_Vyhodnoceni).

>> **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** Vloží výsledek do připraveného bloku.

function!**collect** a **keep**

Shromáždí v novém bloku všechny hodnoty vybrané funkci keep z těla bloku.

Jinými slovy: vytvoří nový blok, zachovávaje pouze hodnoty určené funkcí keep, obvykle na základě splnění určité podmínky.

Red []

a: [11 "house" 34.2 "dog" 22]

b: **collect** [

**foreach** element a [**if** **string?** element [keep element]]

]

**print** b ;obsah bloku b je vytvořen funkcí collect

house dog

syntaxe pro upřesnění: **collect/into**

Red []

c: ["one" "two"]

a: [11 "house" 34.2 "dog" 22]

**collect**/into [

**foreach** element a [**if** **scalar?** element [keep element]]] c

**print** c ;obsah bloku **c** je rozšířen o vybrané prvky bloku **a**

one two 11 34.2 22

native!**compose**

Vrací kopii bloku, vyhodnocujíce pouze prvky typu paren!.

Příkaz compose je velmi důležitý pro dialekt [DRAW](#_topic_Draw);

Red []

a: [**add** 3 5 (**add** 3 5) 9 + 8 (9 + 8)]

**print** a ; vyhodnotí vše, včetně zanořených závorek

**probe** **compose** a ; vrací argument s vyhodnocenými závorkami typu paren!

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

**/deep** => provést příkaz compose i pro zanořené bloky

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** - složit (compose) vnořené bloky jako bloky, obsahující své hodnoty nemtudum

je zapotřebí vložit příklad

**/into** - vložit výsledek do připraveného bloku místo vytvářní nového bloku.

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]

*Created with the Standard Edition of HelpNDoc:* [*Free help authoring tool*](https://www.helpndoc.com/help-authoring-tool)

**Matematika a logika**

Většina matematiky a logiky v Redu je běžná až na pořadí [výpočtu](#_topic_Vyhodnoceni). Níže uvádím výčet operátorů (slov), používaných pro výpočty spolu s případnými poznámkami.

**Základy:**

Následující skupinu tvoří jak **prefixové** (např. add) tak **infixové**  (např. operátor "+") funkce . Přijímají hodnoty typu number!, char!, pair!, tuple! nebo vector! jako argumenty.

Funkcionální (prefixové) operátory se uvádějí před svými operandy (např.: add 3 4).

action!**add** or op! **+**

action!**subtract** or op!  **-**

action!**multiply** or op! **\***

action!**divide** orop! **/**

action!**power** orop! **\*\***

action!**absolute**

Vyhodnotí výraz a vrátí absolutní hodnotu, to jest kladné číslo.

action!**negate**

Změní signum hodnoty, to jest: positive <=> negative

float!**pi** (konstanta typu float!)

3,141592...

action!**random**

Vrací náhodnou hodnotu téhož typu jako argument.

Je-li argumentem celé číslu, vrací integer mezi 1 (inclusive) a argumentem (inclusive).

Je-li argumentem desetinné číslo, vrací float mezi 0 (inclusive) a argumentem (inclusive).

Je-li argumentem řada, přeskupí elementy.

>> **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

**Upřesnění (refinements):**

**/seed** - Restartovat a randomizovat. Patrně lze použít v situaci, kdy je funkce random volána v programu opakovaně. V tom případě nemusí být výsledek zcela náhodný, pokud se nepoužije random/seed.

**/secure** - TBD: Vrací kryptograficky zajištěné náhodné číslo.

**/only** - Vybrat náhodnou hodnotu z řady.

>> **random/only ["fly" "bee" "ant" "owl" "dog"]**   
== "fly"   
  
>> **random/only "aeiou"**   
== #"o"

action!**round**

Vrací nejbližší celé číslo. Poloviny (např. 0,5) se zaokrouhlují směrem od nuly.

>> **round 2.3**   
== 2.0   
  
>> **round 2.5**   
== 3.0   
  
>> **round -2.3**   
== -2.0   
  
>> **round -2.5**   
== -3.0

**Refinements:**

**/to** - pro určení "přesnosti" zaokrouhlení:

>> **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** - Poloviny (např. 0.5) se zaokrouhlují směrem k sudému číslu.

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

**/down** - Odsekne desetinnou část ale číslo zachová jako float!.

>> **round/down 3.9876**   
== 3.0  
   
>> **round/down -3.876**   
== -3.0

**/half-down** - Poloviny zaokrouhlovat směrem k nule, nikoli od nuly.

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

**/floor** - Zaokrouhlovat v negativním směru.

>> **round/floor 3.8**   
== 3.0   
  
>> **round/floor -3.8**   
== -4.0

**/ceiling** - Zaokrouhlovat v pozitivním směru.

>> **round/ceiling 2.2**   
== 3.0   
  
>> **round/ceiling -2.8**   
== -2.0

**/half-ceiling** - Zaokrouhluje poloviny v pozitivním směru.

>> **round/half-ceiling 2.5**   
== 3.0   
  
>> **round/half-ceiling -2.5**   
== -2.0

native!**square-root**

Příjímá libovolnou hodnotu typu number! jako argument.

**Remainders (zbytky) etc.:**

action!**remainder**nebo op!**//**

Přijímá hodnotu typu number! char! pair! tuple! a vector! jako argument. Vrací zbytek dělení prvního čísla druhým.

>> **remainder 15 6**   
== 3   
  
>> **remainder -15 6**   
== -3   
  
>> **remainder 4.67 2**   
== 0.67  
   
>> **17 // 5**   
== 2   
  
>> **4.8 // 2.2**   
== 0.3999999999999995

op!**%**

Vrací zbytek po dělení jedné hodnoty druhou.

function!**modulo**

Vrací pozitivní zbytek (modulus) po dělení prvního argumentu druhým. Je-li některé z čísel (či obě) záporné, uvedené jednoduché pravidlo neplatí a je to mnohem složitější - viz Vikipedia: [Modulo operation](https://en.wikipedia.org/wiki/Modulo_operation).

>> **modulo 9 4**   
== 1   
  
>> **modulo -15 6** ;bylo to však složitější   
== 3   
  
>> **modulo -15 -6** ;bylo to však složitější  
== 3   
  
>> **modulo -15 7** ;neboť to je složitější  
== 6   
  
>> **modulo -15 -7** ;neboť to je složitější   
== 6

**Logaritmy :**

function!**exp**

Druhá mocnina přirozeného čísla **e**.

native!**log-10**

Pro daný argument vraci logaritmus se základem 10 .

native!**log-2**

Pro daný argument vraci logaritmus se základem 2.

native!**log-e**

Pro daný argument vraci logaritmus se základem e.

**Trigonometrie:**

Všechny trigonometrické funkce s dlouhými názvy (arccosine, cosine etc) přijímají argumenty ve stupních, umožňují ale použití argumentu v radiánech s upřesněním  /radians. Verze s krátkými názvy (acos, cos etc.) přijímají úhly v radiánech.

function!**acos** či native! **arccosine**

function!**asin** či native!**arcsine**

function!**atan** činative! **arctangent**

Vrací trigonometrický arctangent.

function!**atan2** činative! **arctangent2**

Vrací úhel spojnice bodu 0,0 a x,y v radiánech, měřený proti směru hodinových ručiček od kladné osy jednotkové kružnice. Vracené hodnoty se pohybují mezi -pi a +pi.

function!**cos** činative! **cosine**

function!**sin**činative! **sine**

function!**tan** činative! **tangent**

**Extras:**

native!**max**

Vrátí větší ze dvou argumentů typu scalar! nebo series!

>> **max 8 12**   
== 12   
  
>> **max "abd" "abcd"**   
== "abd"

Při porovnávání řetězců (což jsou series!) nebo bloků je postupně porovnáván každý element. Při výskytu první neshody porovnávání končí a jako větší je vybrán blok (string) s větší hodnotou neshodného elementu.

>> **max [1 2 3] [3 2 1]** ; porovnávání skončilo hned u prvního elementu  
== [3 2 1]   
  
>> **max [1 2 99] [3 2 1]** ; rovněž tak  
== [3 2 1]

Při porovnání hodnot typu pair!, vrací větší z každého elementu:

>> **max 12x6 7x34**   
== 12x34

native!**min**

Vrací menší ze dvou argumentů. Poznámka o max je zde rovněž revelantní.

action!**odd?**

Vrací true je-li argument typu integer! lichý a false je-li sudý.

action!**even?**

Vrací true  je-li argument typu integer! sudý a false  je-li lichý.

native!**positive?**

Vrací true je-li argument větší než nula a false je-li menší než 1.

native!**negative?**

Vrací true je-li argument menší než 1 a false je-li větší než 0.

native!**zero?**

Vrací true je-li argument roven nule.

function!**math**

Vyhodnotí hodnotu typu block! s použitím normálních precedencí, to jest s upřednostněním dělení a násobení před sčítáním a odčítáním.

>> **math [2 + 3 \* 4]**   
== 14

function!**within?**

Přijímá 3 argumenty typu pair! - pozici bodu x,y - souřadnice x,y levého horního bodu obdélníka - šířku a výšku obdélníka.

Vrací  true!, nalézá-li se bod uvnitř nebo na hranici plochy obdélníka.

>> **within? 12x11 5x10 8x2** ; souřadnice vrcholů trojúhelníka typu pair!  
== true

native!**NaN?**

Vrací  true! není-li argument typu number!

>> **NaN? square-root 9**   
== false

>> **square-root -9**   
== 1.#NaN ; Red neumí komplexní čísla?

native!**NaN**

Returns TRUE if the number is Not-a-Number.

function! **a-an** nemtudum

Returns the appropriate variant of a or an (simple, vs 100% grammatically correct).

**Logické funkce :**

action! **and~**čiop! **and (infix)**

native! **equal?**čiop! **=**

native! **greater-or-equal?**či op!**>=**

native! **greater?**či op!**>**

native! **lesser-or-equal?**či op!**<=**

native! **lesser?**či op!**<**

native! **not**

native!**not-equal?**čiop! **<>**

action! **or~**čiop! **or (infix)**

native! **same?**či op!**=?**

Vrací true! odkazují-li argumenty na tatáž data (object, string etc.), to jest, odkazují-li na stejné místo v paměti.

>> **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?** orop! **==**

Vrací true! mají-li argumenty stejnou hodnotu a jsou stejného typu či velikosti (lower-case/uppercase) u řetězců.

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

>> **a == b**   
== false

*Created with the Standard Edition of HelpNDoc:* [*Create help files for the Qt Help Framework*](https://www.helpndoc.com/feature-tour/create-help-files-for-the-qt-help-framework)

**Konverze bází**

native!**to-hex**

Konvertuje integer! na issue! (with leading # and 0's).

>> **to-hex 10**  
== #0000000A ; hexadecimální formát

>> **to-hex 16**  
== #00000010

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

**/size** - určit počet hexadecimálních číslic ve výsledku

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

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

native!**enbase** anative!**debase,**

Používají se pro kódování a dekódování binárních řetězců.   
These are not for number conversion and, honestly, I don't understand the use for them, but here is how they work:

>> **enbase "my house"**  
== "bXkgaG91c2U="  
  
>> **probe to-string debase "bXkgaG91c2U="** "my house"  
== "my house"

**/base** - báze binárního formátu; může to být 64 (default), 16 nebo 2.

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

native!**dehex**

Converts URL-style hex encoded (%xx) strings.

>> **dehex "www.mysite.com/this%20is%20my%20page"**  
== "www.mysite.com/this is my page" ; Hex 20 (%20) is space

>> **dehex "%33%44%55"**  
== "3DU"  
; %33 is hex for "3", %44 is hex for "D" and %55 is hex for "U".

*Created with the Standard Edition of HelpNDoc:* [*News and information about help authoring tools and software*](https://www.helpauthoringsoftware.com)

**Kryptografie**

native!**checksum**

Počítá checksum, CRC, hash nebo HMAC.

Argumenty mohou mít hodnotu string! binary! nebo 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** - Extra value for HMAC key or hash table size; not compatible with TCP/CRC32 methods.

I believe hash is not implemented in Red 0.63 and I could not figure out how HMAC works.

*Created with the Standard Edition of HelpNDoc:* [*Single source CHM, PDF, DOC and HTML Help creation*](https://www.helpndoc.com/help-authoring-tool)

**Kopírování**

**Varování pro začátečníky**: Přířazujete-li hodnotu slova jinému slovu , proveďte to s příkazem COPY.

>> **var1: var2** ; jste-li si jisti, co činíte   
  
>> **var1: copy var2** ; může vám ušetřit hodiny ladění

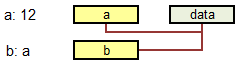
action!**copy**

Přiřadí kopii dat jinému slovu.

Lze použít ke kopírování řad a  [objektů](#_topic_Objekty).

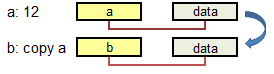
Nepoužívá se pro jednotlivé položky typu: integer! float! char! etc. Pro tyto účely můžeme jednoduše použít dvojtečku.

Pohleďme na jednoduché přiřazení (závislá kopie):



>> **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"] ; změna se projeví u obou proměnných

Nyní na přiřazení s 'copy' (nezávislá kopie):



>> **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"]

'Nezávislost' se netýká vnořených řad (bloků). Příkaz  copy nemění odkaz na tyto vnořené řady. Chcete-li pro tento případ vytvořit nezávislou kopii, musíte použít upřesnění /deep.

**copy/part**

Omezuje délku (number! či series!) kopírované části.

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

**copy/types**

Kopíruje jenom určité typy neskalárních hodnot.

**copy/deep**

Kopíruje vnořené hodnoty.

*Created with the Standard Edition of HelpNDoc:* [*Write eBooks for the Kindle*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Opakování**

native!**loop**

Provede blok opakovaně v zadaném počtu.

Red[]

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

hello!  
hello!  
hello!  
>>

native!**repeat**

repeat je totéž jako 'loop', obsahuje však počítadlo (index), jehož hodnota se s každou smyčkou zvětšuje o 1

Red[]

**repeat** i 3 [**print** i]

1  
2   
3   
>>

native!**forall**

Provádí blok pro posouvanou polohu 'vstupního indexu'.

Red[]

a: ["china" "japan" "korea" "usa"]

**forall** a [**print** a]

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

native!**foreach**

Vrací první hodnotu řady s posouvaným 'vstupním indexem'.

Red[]

a: ["china" "japan" "korea" "usa"]

**foreach** i a [**print** i]

china  
japan  
korea  
usa  
>>

native!**while**

Provádí blok, pokud je splněna zadaná podmínka.

Red[]

i: 1

**while** [i < 5] [

**print** i

i: i + 1

]

1  
2   
3   
4   
>>

native!**until**

Prvádí blok, pokud není plněna vstupní podmínka.

Red[]

i: 4

**until** [

**print** i

i: i **-** 1

i < 0 ; <= condition

]

4  
3   
2   
1   
0   
>>

native!**break**

Ukončí provádění smyčky při nesplnění zadané podmínky.

**/return**

Totéž, navíc však vrátí zadaný text pro zjištění, kdy nebyla podmínka splněna.

>> **print foreach number [1 2 4 8 16]** [  
 if number > 4 [break/return <finito>]  
 print number ]

1  
2   
4   
<finito>

native!**forever**

Vytvoří nikdy nekončící smyčku.

*Created with the Standard Edition of HelpNDoc:* [*Easily create Help documents*](https://www.helpndoc.com/feature-tour)

**Podmínky**

native!**if**

Provede blok při splnění zadané podmínky.

**if <test> [**block **]**

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

native!**unless**

Totéž jako if not. Provede blok pouze při nesplnění zadané podmínky.

**unless <test> [**block (if test false)**]**

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

native!**either**

Nový název pro klasickou podmínku if-else. Provede první blok, je-li podmínka vyhodnocena jako  true ; v opačném případě provede druhý blok.

**either <test> [**trueblock**] [**falseblock**]**

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

native!**switch**

Provede blok, odpovídající zadané hodnotě.

Red[]

**switch** 20 [

10 [**print** "ten"]

20 [**print** "twenty"]

30 [**print** "thirty"]

]

twenty   
>>

**/default**

Nenalezne-li program shodu se zadáním, provede '*defaultní'* blok.

Red[]

**switch**/default 15 [

10 [**print** "ten"]

20 [**print** "twenty"]

30 [**print** "thirty"]

][ **print** "none of them"] ;default block

none of them >>

native!**case**

Vyhodnotí řadu podmínek a provede blok, odpovídající prvnímu testu s hodnocením true.

Red[]

**case** [

10 > 20 [**print** "not ok!"]

20 > 10 [**print** "this is it!"]

30 > 10 [**print** "also ok!"]

]

this is it!   
>>

**/all**

Provede všechny testy s hodnotou 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** a **throw** lze použít pro vytvoření složitější řídící struktury. Ve sve nejjednodušší formě obdrží catch  návratovou hodnotu z více možností 'throw' :

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**

odchytí pojmenované 'throw'

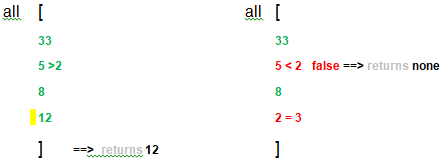
**throw/name**

throws to a named catch

**Boolean branching**

native!**all**

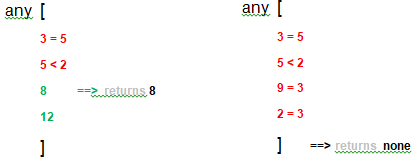
Vyhodnotí všechny výrazy v bloku a vrací poslední výslednou hodnotu, lze-li všechny výrazy označit jako true. Případně vrací  none, je-li některý výraz hodnocen jako false.



>> **john: "boy"**   
== "boy"   
  
>> **alice: "girl"**   
== "girl"   
  
>> **all [john = "boy" alice = "girl" 10 + 3]** ;all true, the last evaluation is returned.   
== 13   
  
>> **all [john = "boy" alice = "boy" 10 + 3]** ; alice = "boy" is false!   
== none

native!**any**

Hodnotí postupně výrazy v bloku a vrací první hodnotu, která není  false. Jsou-li všechny vyhodnocené hodnoty false , vrací  none.



>> **john: "boy"**   
== "boy"   
  
>> **alice: "girl"**   
== "girl"   
  
>> **any [john = "girl" alice = "girl" 10 + 3]** ;alice = "girl" is not false: return it!   
== true   
  
>> **any [john = "girl" 10 + 3 5 > 2]** ; 10 + 3 is not false: return it!   
== 13

*Created with the Standard Edition of HelpNDoc:* [*Free PDF documentation generator*](https://www.helpndoc.com)

**Manipulace s textem**

function! **split**

Vytvoří [blok](#_topic_Blokyarady), sestavený z částí řetězce, rozděleného zadaným oddělovačem. Příkaz  split  je obzvláště vhodný pro analýzu a manipulaci s textovými  [soubory](#_topic_Soubory).

>> **s: "My house is a very funny house"**  
>> **split s " "** ; oddělovačem je každá mezera   
== ["My" "house" "is" "a" "very" "funny" "house"]

>> **s: "My house ; is a very ; funny house"**   
>> **split s ";"** ; oddělovačem (delimiter) je středník  
== ["My house " " is a very " " funny house"]

**Odebrání znaků:**action!**trim**

Slovo trim (zredukovat) bez upřesnění odebírá bílá místa (taby a mezery) ze začátku a konce řetězce (také odebírá none z entity typu block! nebo object!). Hodnota argumentu se změní.

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

**trim/head** - odebírá mezery pouze z čela

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

**trim/tail** - odebírá mezery pouze z chvostu

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

**trim/auto** - Auto indents lines relative to first line. Dosud neimplementováno

**trim/lines** - odebere přerušení řádku a nadbytečné mezery  
**trim/all** - odebere všechny mezery, nikoliv ale přerušení řádku

**trim/with** - odebere zadané znaky typu: char! string! integer!

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

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

**Opak trim:**function!**pad**

Příkaz pad rozšíří řetězec na danou velikost přidáním mezer. Implicitně se mezery přidávají vpravo ale s upřesněním /left  se mezery přidávají vlevo. Pozor, mění původní string.

>> **a: "House"**   
>> **pad a 10**   
== "House "   
  
>> **pad/left a 20**   
== " House "

**Spojení textu:**function!**rejoin**

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

případně s příkazem append, což mění původní řady:

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

>> **a: "house" b: " " c: "entrance"**   
  
>> **append a c**   
== "houseentrance"   
  
>> **append a append b c**   
== "houseentrance entrance" ; "a" was changed to "houseentrance" in the last manipulation

**Přeměna řady na text:**action!**form**

Příkaz form přemění řadu na string odebráním hranatých závorek a přidáním mezer mezi elementy.

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

**form/part**

Upřesnění /part limituje počet znaků ve vytvořeném řetězci.

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

**Zjištění délky řetězce:** action! **length?**

>> **f: "my house"**  
>> **length? f**  ; see chapter "Series 'getters'"   
== 8

**Přehled manipulací**

**Levá strana řetězce:**

použitím **copy/part** :

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

**Pravá strana řetězce:**

použitím at :

>> **s: "nasty thing"**  
>> **at tail s -5**   
== "thing"

použitím remove/part - pozor, mění původní řetězec!

>> **s: "nasty thing"**  
>> **remove/part s 6**   
== "thing"

**Střední část řetězce:**

použitím copy/part a at:

>> **a: "abcdefghijkl"**  
>> **copy/part at a 4 3**   
== "def"

**Vkládání řetězců:**

**na začátek, s použitím** insert:

>> **s: "house"**  
>> **insert s "beautiful "**   
  
>> **s**   
== "beautiful house"

**na konec, s použitím** append:

>> **s: "beautiful"**  
>> **append s " day"**   
== "beautiful day"

**ve střední části, s použitím** insert at:

>> **s: "nasty thing"**  
>> **insert at s 7 "little "**   
  
>> **s**   
== "nasty little thing"

native!**lowercase**

Pozor, mění původní řetězec.

>> **a: "mY HoUse"**  
>> **lowercase a**   
== "my house"

**lovercase/part**

>> **a: "mY HoUse"**   
>> **lowercase/part a 2**   
== "my HoUse"

native!**uppercase**

>> **a: "mY HoUse"**  
>> **uppercase a**   
== "MY HOUSE"

**uppercase/part**

>> **a: "mY HoUse"**  
>> **uppercase/part a 2**   
== "MY HoUse"

*Created with the Standard Edition of HelpNDoc:* [*Generate EPub eBooks with ease*](https://www.helpndoc.com/create-epub-ebooks)

**Práce s časem**

native! **wait**

Zastaví exekuci na zadaný počet vteřin.

* Note: as of November 2017, the GUI Console does not work well with wait.

native!**now**

Vrací datum a čas:

>> **now**   
== 12-Dec-2017/19:24:41-02:00

**now/time** - Vrací pouze čas (time!)

>> **now/time**  
== 21:42:41

**now/precise** - Zvýšená preciznost údaje (date!)

>> **now/precise**  
== 2-Apr-2018/21:49:04.647-03:00

>> **a: now/time/precise**  
== 22:05:46.805 ;a is a time!  
  
>> **a/hour**  
== 22 ;hour is an integer!  
  
>> **a/minute**  
== 5 ;minute is an integer!  
  
>> **a/second**  
== 46.805 ;second is a float!

Tento skript vytvoří časový posun o 300 milisekun (0.3 sekundy):

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

**now/year** - Vrací pouze rok (integer!)

>> **now/year**  
== 2018

**now/month**- Vrací pouze měsíc

>> **now/month**  
== 4

**now/day**- Vrací pouze den měsíce

>> **now/day**  
== 2

**now/zone** - Vrací časový odstup od UCT (GMT) (time!)

>> **now/zone**  
== -3:00:00

**now/date** - Vrací pouze datum (date!)

>> **now/date**  
== 2-Apr-2018

**now/weekday** - Vrací den tydne jako integer! (pondělí je den 1).

>> **now/weekday**  
== 1

**now/yearday** - Vrací den roku (Julian).

>> **now/yearday**  
== 92

**now/utc** - Univerzální čas (no zone) (date!)

>> **now/utc**  
== 3-Apr-2018/0:53:50

VID DLS**rate**

Nastavení času lze také provést v dialektu [VID s použitím aspektu rate](#_topic_Pokrocilatemata).

*Created with the Standard Edition of HelpNDoc:* [*Free HTML Help documentation generator*](https://www.helpndoc.com)

**Ošetření chyb**

function! **attempt**

Vyhodnotí blok a vrací výsledek nebo vrací none při výskytu chyby.

>> **attempt [a: 10 b: 9]**  ;first lets try with no errors...   
== 9   
>> **a**   
== 10 ;... no problems here!   
  
  
>> **attempt [a: 10 nosyntax]** ;nosyntax has no value: ERROR!   
== none

native!**try**

Pokusí se o vyhodnocení bloku. Vrací hodnotu bloku ale při výskytu chyby vrací hodnotu error!.

Pro identifikaci závadného bloku bez tisku chybového hlášení použijeme funkci  error?.

Můžete se ptát proč nepoužít attempt místo error? & try. Domnívám se že proto, že kombinace error? & try vrací true a false místo none nebo provedeného vyhodnocení. To je užitečné při použití uvnitř jiných struktur.

>> **error? [nosyntax]**   
== false ;nosyntax has no value and it generates an error,   
 ;but only if evaluated. In itself, is not a error! datatype.   
  
>> **try [nosyntax]**   
\*\*\* Script Error: nosyntax has no value   
\*\*\* Where: try   
\*\*\* Stack: ; just "try" does not work, you get an error!!   
  
>> **error? try [nosyntax]**   
== true ; OK!

native!**catch** andnative! **throw**

Tyto příkazy se používají k ošetření chyb. Delší vysvětlení lze nalézt  [zde](http://www.red-by-example.org/index.html#catch).

*Created with the Standard Edition of HelpNDoc:* [*Easily create PDF Help documents*](https://www.helpndoc.com/feature-tour)

**Soubory**

**Názvy cest**

Cesty k souborům se píší se znakem %, následovaným výčtem adresářů, oddělených lomítkem. Ve Windows si Red sám mění zpětná lomítka na normální lomítka.

Jenom pro připomnění:

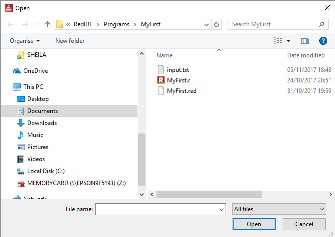
* / písmeno před prvním lomítkem označuje kořenový adresář oddílu
* ./ označuje aktuální adresář
* ../ označuje rodiče aktuálního adresáře
* cesty, které nezačínají odkazem na kořenový adresář oddílu, jsou relativní
* absolutní cesta pro často používaný oddíl "C" ve Windows: %/C/docs/file.txt
* absolutní cesty nejsou strojově nezávislé;

**Grafický selektor souboru:**

function!**request-file**

Příkaz request-file otevře grafické okno Průzkumníka a vrátí úplnou cestu vybraného souboru (file!)

>> **request-file**



== %/C/Users/André/Documents/RED/myFirstFile.txt

**Refinements**

**request-file/title** - Window title.

**request-file/file** -

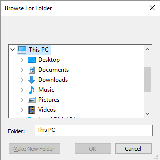
**request-file/filter** - Poskytnutý blok filtrů ovlivní konfiguraci otevřeného výběrového okna. Například, request-file/filter ["executables" "\*.exe" "text files" "\*.txt"] vytvoří roletku s nabídkou 'executables' a 'text files'.

**request-file/save** - File save mode. Example with filters: request-file/save/filter ["executables" "\*.exe" "text files" "\*.txt"]

**Grafický selektor adresáře:**

function!**request-dir**

Příkaz request-dir otevře okno se základní nabídkou adresářů (User, Tento počítač, Knihovny, Síť). Pro vybraný soubor či složku vrací úplnou cestu  (file!)



== %/C/Users/André/Documents/RED/

Refinements

request-dir**/title** => Window title.

request-dir**/dir** => Set starting directory.

request-dir**/filter** => TBD: Block of filters (filter-name filter).

request-dir**/keep** => Keep previous directory path.

request-dir**/multi**  => TBD: Allows multiple file selection, returned as a block.

**Smazání souboru:**

action!**delete**

 Smaže soubor a vrací true, pokud se zadařilo, jinak vrací false.

>> **delete %file.txt**   
== true

**Zjištění velikosti souboru:**

native!**size?**

Vrací velikost v bajtech nebo none, když soubor neexistuje.

>> **size? %myFirstFile.txt**   
== 37

**Další funkce:**

**cd** nebo **change-dir -**mění aktuální adresář.

**dir, ls** nebo **list-dir -**vypíše obsah adresáře; bez poskytnutého argumentu vypíše obsah aktuálního adresáře.

**dir? -** vrací 'true', je-li zadané jméno platnou cestou, jinak vrací 'false'.

**dirize -** přemění argument na platný adresář; argumentem může být hodnota typu file! string! url!.  
Effectively dirize only appends a trailing / if needed.

**exists? -**vrací 'true', je-li argument platnou cestou typu path!

**file? -**  vrací 'true', je-li argument souborem

**get-current-dir** - vrací aktuální adresář, používaný programem

**get-path? -** vrací 'true', je-li argument typu get-path!

**path? -**vrací 'true', je-li argument typu path!

**split-path -**rozdělí sestu typu file! nebo url! path; vrací blok, obsahující cestu a cíl

**suffix?** - vrací příponu souboru, např. exe, txt

**what-dir -** vrací cestu k aktuálnímu adresáři ve formátu file!

**to-red-file -**Converts a local file system path to Red's  
standard machine independent path format.

**to-local-file -**Converts standard, system independent Red  
file paths to the file format used by the local operating system.

**to-red-file -**maže '.' a '..' v cestě a vrací vyčištěnou cestu

**red-complete-file**

**red-complete-path**

**set-current-dir**

*Created with the Standard Edition of HelpNDoc:* [*News and information about help authoring tools and software*](https://www.helpauthoringsoftware.com)

**Psaní do souboru**

**Psaní do textového souboru:**

action!**write**

Provede zápis do souboru, který vytvoří, pokud neexistuje.

>> **write %myFirstFile.txt "Once upon a time..."**

**Připojení a manipulace s textovým souborem:**

**write/append**

Opětovný zápis do souboru způsobí vymazání původního obsahu. Chcete-li jenom připojit (append) další text, použijete:

>> **write/append %myFirstFile.txt "there was a house."**

Váš soubor nyní obsahuje "Once upon a time...there was a house".

**Psaní řad na vlastní řádky:**

Přidejme nyní další soubor se třemi řádky:

>> **write/lines %mySecondFile.txt ["First line;" "Second line;" "Third line."]**

**Přidávání dalších řádků:**

>> **write/append/lines %mySecondFile.txt ["Fourth line;" "Fifth line;" "Sixth line."]**

Soubor nyní vypadá takto:

First line;   
Second line;   
Third line.   
Fourth line;   
Fifth line;   
Sixth line.

Vězte, že jste mohli psát write/lines/append. Pořadí upřesnění nemá na nic vliv.

**Výměna znaků v souboru:**

Pro výměnu znaků, počínaje na n+1 pozici, použijeme write/seek %<file> <n> :

>> **write/seek %myFirstFile.txt "NEW TEXT" 5**

První soubor nyní obsahuje text: "Once NEW TEXTime...there was a house."

**Upřesnění**

/binary => Preserves contents exactly.

/lines => Write each value in a block as a separate line.

/info =>

/append => Write data at end of file.

/part => Partial write a given number of units.

/seek => Write at a specific position.

/allow => Specifies protection attributes.

/as => Write with the specified encoding, default is 'UTF-8.

function! **save**

Uloží hodnotu, blok, či jiná data do souboru, URL, bináru (binary) nebo řetězce.

**Rozdíl mezi write a save:**

>> **write %myFourthFile.txt [11 22 "three" "four" "five"]**

Soubor nyní obsahuje blok: [11 22 "three" "four" "five"]

>> **save %myFourthFile.txt [11 22 "three" "four" "five"]**

Soubor nyní (po přepsání) obsahuje výčet: 11 22 "three" "four" "five"

Důležité použití příkazu save je při ukládání skriptů Redu, které mohou být interpretovány s použitím akce  do :

>> **save %myProgram.r [Red[] print "hello"]**   
>> **do %myProgram.r**   
hello

Příkazy do, [load](https://ungaretti.gitbooks.io/red-language-notebook/content/files-and-i-o.html) a [save](https://ungaretti.gitbooks.io/red-language-notebook/content/files-and-i-o.html)  se lépe chápou, považujete-li konzolu Redu za monitor starého počítače z 80. let, realizující některou variaci základního jazyka. Můžete program  load (načíst),  save (uložit) nebo do (provést).

*Created with the Standard Edition of HelpNDoc:* [*Generate Kindle eBooks with ease*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Čtení ze souboru**

**Načtení celého souboru do jednoho řetězce:**

action!**read**

>> **a: read %mySecondFile.txt**   
== {First line;^/Second line;^/Third line.^/Fourth line;^/Fifth li

Nyní proměnná "a" obsahuje celý obsah souboru:

>> **print a**   
First line;   
Second line;   
Third line.   
Fourth line;   
Fifth line;   
Sixth line.

**Načtení souboru do bloku po jednotlivých řádcích:**

Chcete-li číst soubor jako řadu, v níž je každý řádek elementem, použijete read/lines:

>> **a: read/lines %mySecondFile.txt**   
== ["First line;" "Second line;" "Third line." "Fourth line;"...   
  
>> **print pick a 2**   
Second line;

**read/part** => čte pouze zadaný počet jednotek (source relative).

**read/seek**  => čte ze zadané pozice (source relative).

**read/binary** => přesně zachovává obsah

**read/lines** => přemění na blok řetězců

**read/info** =>

**read/as**  => čte se zadaným kódováním, implicitně je 'UTF-8.

function!**load**

**Načtení souboru do bloku po jednotlivých slovech, oddělených mezerou:**

Pro tento účel použijeme load místo read:

>> **a: load %mySecondFile.txt**   
== [First line Second line Third line.   
 Fourth line Fifth...   
  
>> **print pick a 2**   
line

**Čtení a psaní binárních souborů:**

Pro čtení či psaní binárního souboru jako je image nebo zvuk, se používá upřesnění  /binary. Následující kód načte bitmapové zobrazení do proměnné a zapíše jej do souboru s jiným názvem:

>> **a: read/binary %heart.bmp**   
== #{   
424D660700000000000036000000280000001E00000014000000010...   
>> **write/binary %newheart.bmp a**

**load/header**  => TBD.

**load/all**  => načte všechny hodnoty, vrací blok; TBD: nevyhodnocuje záhlaví Red.

**load/trap** => načte všechny hodnoty, vrací blok: [[values] position error].

**load/next**  => načte pouze následující hodnotu.

**load/part**  =>

**load/into** => vloží obsah do existujícího bloku

**load/as**  => zadání typu načítaných dat; použijeme NONE pro načtení jako kód

*Created with the Standard Edition of HelpNDoc:* [*Write eBooks for the Kindle*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Funkce**

Funkce musejí být před použitím deklarovány a proto se píší v přední části programu. To ale není požadováno, je-li funkce volána z jiné funkce.

native!**func**

Skladba funkce s func je tato:

**<název>: func [<arg1> <arg2> ... <arg n>] [ <actions performed on arguments>]**

Red []

mysum: **func** [a b] [a + b]

**print** mysum 3 4

7

Proměnné uvnitř funkce, vytvořené se slovem func, jsou **globální**. Jsou viditelné z celého programu.:

Red []

mysum: **func** [a b] [

mynumber: a + b

**print** mynumber

]

mynumber: 20

mysum 3 4

**print** mynumber

7

7

native!**function**

Proměnné uvnitř funkce se slovem function  jsou **lokální**, to jest, jsou přístupné pouze uvnitř definované funkce.

Red []

mysum: **function** [a b] [

mynumber: a + b

**print** mynumber

]

mynumber: 20

mysum 3 4

**print** mynumber

Různé výsledky:

7

20

**Vnucená globální přístupnost proměnných upřesněním /extern:**

Red []

myfunc: **function** [/extern a b] [

a: 22

b: 33

]

a: 7

b: 9

myfunc

**print** a

**print** b

22

33

**Definice typu argumentu:**

Datový typ (datatype) argumentu lze předem určit :

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

Lze připustit více datových typů:

Red []

mysum: **function** [a [**integer!** **float!**] b[**integer!**]] [**print** a + b]

**print** mysum 3.2 4

7.2

Případně lze zadat nadřízený datový typ:

Red []

mysum: **function** [a [**number!**] b[**number!**]] [**print** a + b]

**print** mysum 3.2 4

7.2

**Vratná hodnota funkce:** native! **return**

Vratnou (**return**) hodnotou funkce je buď poslední funkcí vyhodnocená hodnota nebo hodnota explicitně určená slovem  return:

Red []

myfunc: **function** [] [

8 + 9

3 + 3

**print** "got here" ; this executes

10 + 5 ; this is returned

]

**print** myfunc

got here

15

Příklad s 'return'; slovo return je poslední aktivní příkaz funkce:

Red []

myfunc: **function** [] [

8 + 9

**return** 3 + 3 ; this is returned

**print** "never got here" ; NOT executed

10 + 5

]

**print** myfunc

6

**Vytváření vlastních upřesnění:**

Je možné vytvářet vlastní upřesnění funkcí, podobná nativním upřesněním Redu: Jednotlivá upřesnění jsou bulíny (boolean values), ověřované funkcí:

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

Pro upřesnění nejsou argumenty povinné.

**Přiřazení funkcí ke slovům (variables)**

To assign a function to a variable (a word) you must precede the function with a colon: <word>: :<function>

>> **mysum: func [a b] [a + b]**

== func [a b][a + b]

>> **a: :mysum**

== func [a b][a + b]

>> **a 3 9**

== 12

native!**does**

Pokud naše rutina jenom něco provádí - bez **argumentů** a **lokálních proměnných**, použijeme slovo does :

Red []

greeting: **does** [

**print** "Hello"

**print** "Stranger"

]

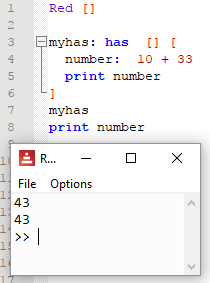
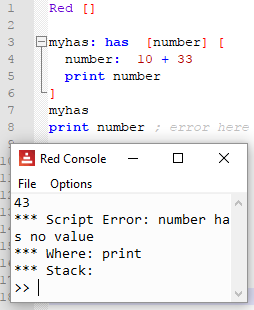
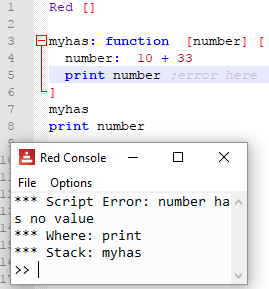
greeting

Hello

Stranger

native!**has**

Pokud naše rutina nepoužívá žádné externí argumenty ale má lokální proměnné, použijeme slovo  has. To přemění případný argument na lokální proměnnou. Porovnejte níže uvedené tři programy. První používá has bez argumentu, slovo number je tedy globální proměnná. Druhá obsahuje number jako argument, jenž je však uvnitř funkce redefinovaný na lokální proměnnou. Pokud změníme název tohoto argumentu, je funkcí ignorován, neboť argument není u tohoto typu funkce povinný. V třetí ukázce vidíme, že v definici funkce musí být argument reálně použit, nikoliv redefinován a že funkce musí být volána s hodnotou argumentu.

native!**exit**

Opustí funkci bez vracení jakékoli hodnoty.

**Různé typy vestavěných funkcí**

* **mezaninové** funkce jsou vytvořeny v jazyce Red a jejich zdrojový kód lze číst s použitím příkazu  source <název fce>.

Výpisy funkcí následujících typů lze získat dotazem (help či ?) na název typu:

* funkce typu **function**
* funkce typu **op!**
* funkce typu **action!**
* funkce typu **routine!**
* funkce typu **native**! jsou součástí interpreta a aktivují se na velmi nízké úrovni;

*Created with the Standard Edition of HelpNDoc:* [*Free EBook and documentation generator*](https://www.helpndoc.com)

**Objekty**

Objekt je kontejnér, který sdružuje data či funkce a je (téměř vždy) přiřazený ke slovu (proměnné). Pro přístup k atributu objektu používáme lomítko (neboli formát podobný cestě). Tímto se Red liší od většiny jazyků, které v podobných situacích používají tečku.

**Vytvoření objektu:**

action!**make object! ,** function! **context** afunction! **object**

K vytvoření objektu použijeme příkaz  make object! nebo jeho kratší varianty object případně  context.

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

Vyhodnocení výrazu se provede pouze při deklaraci objektu (constructor code). Zde například příkaz 'print' je posléze nepřístupný.:

>> **myobject: object [print "hello" a: 1 b: 2]**   
hello   
== make object! [   
 a: 1   
 b: 2   
]   
>> **myobject/a**   
== 1

**Reference 'self':**

Má-li objekt odkazovat sám na sebe, použijeme specielní klíčové slovo 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]

**Klonování objektu a dědění:**

Pouhé přiřazení objektu k jinému jménu vytváří pouhou kopii **ukazovátka** (pointer) na tento objekt. Změní-li se originál, změní se i jeho prezentace v jiném objektu:

>> **a: object [x: 10]** ; odezva konzoly zde není uvedena   
>> **b: a**  ; odezva konzoly zde není uvedena   
>> **a/x: 20**   
== 20   
  
>> **b/x**   
== 20 ; rovněž změněno! b je závislá kopie

K vytvoření nezávislé kopie objektu použijeme slovo  copy:

>> **a: object [x: 10]** ; odezva konzoly zde neuvedena  
>> **b: copy a** ; odezva konzoly zde neuvedena  
>> **a/x: 20**   
== 20   
>> **b/x**   
== 10 ; žádná změna! b je nezávislá kopie

Chceme-li vytvořit nový objekt, který dědí z jiného objektu, použijeme sekvenci: makezdroj [nová specifikace]:

Red []

a: **object** [x: 3] ; deklarace zdrojového objektu

b: **make** a [y: 12] ; nový objekt

**print** b

x: 3

y: 12

**find** a **select -** pro objekty

find zkontroluje, zda zadané pole existuje a vrací true nebo none

select provádí totéž ale pokud zadané pole existuje, vrací jeho hodnotu

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

Přípustný formát zadávaného pole je lit-word! ('word). Hodnotu proměnné lze evokovat příkazem  obj/a.

*Created with the Standard Edition of HelpNDoc:* [*Easily create Web Help sites*](https://www.helpndoc.com/feature-tour)

**Reaktivní programování**

**Reaktivní programování je pěkně popsané v oficiální**[**dokumentaci**](https://doc.red-lang.org/cs/reactivity.html)**.**

Reaktivní programování vytváří interní mechanizmus, který automaticky aktualizuje stav objektu A při změně objektu B - bez volání funkcí či subrutin.

Red používá objektově orientovaný [reaktivní model push.](https://en.wikipedia.org/wiki/Reactive_programming#Object-oriented)

**Reaktor** je reaktivní objekt, jehož změna spustí změny v jiných objektech. Vytváří se příkazem  make reactor!.

**Reaktivní vztah** (statický nebo dynamický) je deklarace vztahu mezi příslušnými poli reaktivně propojených objektů. Vytváří se funkcemi is nebo react .

Příkaz **make reactor!** afunkce**is**

Velmi bazální příklad použití reaktivního programování:

Red[]

a: **make** **reactor!** [x: ""] ;reaktor

b: **is** [a/x] ;reaktivní vztah

**forever** [

a/x: **ask** "? " ;here we input a value for x field of 'a'

**print** b ;here we print b and... surprise! it changed!

] ; Pozor, vytváří nekonečnou smyčku!

?house

house

?fly

fly

?bee

bee

Reaktor může aktualizovat sám sebe:

Red[]

a: **make** **reactor!** [x: 1 y: 2 total: **is** [x + y]]

**forever** [

a/x: **to** **integer!** **ask** "?"

**print** a/total

] ; Pozor, vytváří nekonečnou smyčku!

?33

35

?45

47

**deep-reactor!**

Stejně jako má copy upřesnění /deep pro přístup k zanořeným hodnotám (bloky uvnitř bloku), totéž má i reactor!.

Tento program má opakovat vstup z konzoly ale **jaksi nechodí**:

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

Pokud však použijeme 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**

Toto je příkaz pro vytváření reaktivních grafických aplikací GUIs. Nahlédněte prosím do [GUI-Pokročilá témata](#_topic_Pokrocilatemata).

function!**clear-reactions**

Odstraní bezpodmínečně všechny definované reakce.

function!**react?**

Zjistí, zda je pole objektu reaktivním zdrojem . Pokud ano, vrátí se první reakce, nalezená jako zdroj v poli objektu; pokud ne, vrací se hodnota note. Upřesnění /target kontroluje, zda je pole cílem místo zdrojem a vrátí první reakci, zacílenou na toto pole nebo vrátí none při absenci shody.

**/target** => kontroluje, jde-li o cíl (target) a ne zdroj (source).

function!**dump-reactions**

Vytvoří seznam registrovaných reakcí pro ladící účely.

*Created with the Standard Edition of HelpNDoc:* [*News and information about help authoring tools and software*](https://www.helpauthoringsoftware.com)

**Rozhraní OS**

native!**call**

Provádí příkazy z shellu. Většinou je to totéž jako z příkazového řádku systémové konzoly (CLI) ale existuje několik úchylek.

Následující kód otevře Windows Explorer:

>> **call "explorer.exe"**   
== 11272 ; this is the number of the process opened.

Toto rovněž chodí:

>> **str: "explorer.exe"**   
== "explorer.exe"   
  
>> **call str**   
== 11916

Ovšem, následující kód vytvoří proces ale neotevře Notepad na obrazovce:

>> **call "notepad.exe"**   
== 4180

Chcete-li chování více podobné zápisu příkazu v shellu, musíte použít upřesnění  /shell:

>> **call/shell "notepad.exe"** ; otevře Notepad na obrazovce   
== 6524

Další upřesnění:

**call/wait**

Spustí příkaz a čeká na jeho ukončení. Buďte opatrní, použijete-li upřesnění /wait pro proceduru, kterou nebudete umět ukončit (jako call "notepad.exe" nahoře), Red bude čekat... a čekat.. až do nekonečna.

**call/input** - zadáváme string!, file! nebo binary!, jež bude přesměrováno do **stdin**.

I don't understand this one. Seems as the same as simply call , as we provide string or a file anyway.

**call/output**

Zadáváme string!, file! nebo binary!, jež přijme přesměrovaný stdout z příkazu. Všimněte si, že výstup je připojen.

Následující kód vytvoří textový soubor s výstupem shellu pro "dir" (seznam souborů a složek z aktuální cesty):

>> **call/output "dir" %mycall.txt**   
== 0

Toto vytvoří (dlouhý) řetězec s výstupem z "dir":

>> **a: ""**   
== ""   
  
>> **call/output "dir" a**   
== 0   
  
>> **a**   
== { Volume in drive C has no label.^/ Volume Serial Number is BC5 ;...

**call/shell/show**

Force the display of system's shell window (Windows only). Your program will run with windows command prompt open.

>> **call/shell/show "notepad.exe"**   
== 12372

**call/console**

Spustí příkaz s I/O přesměrovaný do konzoly (CLI console only at present, does not work with Red's normal GUI console).

native!**write-clipboard &  read-clipboard**

Zapisuje do a čte ze schránky OS:

>> **write-clipboard "You could paste this somewhere you find useful"**   
== true   
  
>> **print read-clipboard**   
You could paste this somewhere you find useful

*Created with the Standard Edition of HelpNDoc:* [*Easy EPub and documentation editor*](https://www.helpndoc.com)

**I/O**

Not available yet on Red 0.63. Planned for Red 0.7

*Created with the Standard Edition of HelpNDoc:* [*Full-featured Kindle eBooks generator*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**GUI - Přehled**

V následujících kapitolách budou jednotlivé elementy a parametry grafického rozhraní(**GUI**) - kontejnery (**containers**), dispozice (**layouts**), piškoty (**faces**) a aspekty (**facets**) popsány zevrubně. Pro orientaci a vědomí vzájemné souvislosti je vhodné, uvést několik základních informací předem.

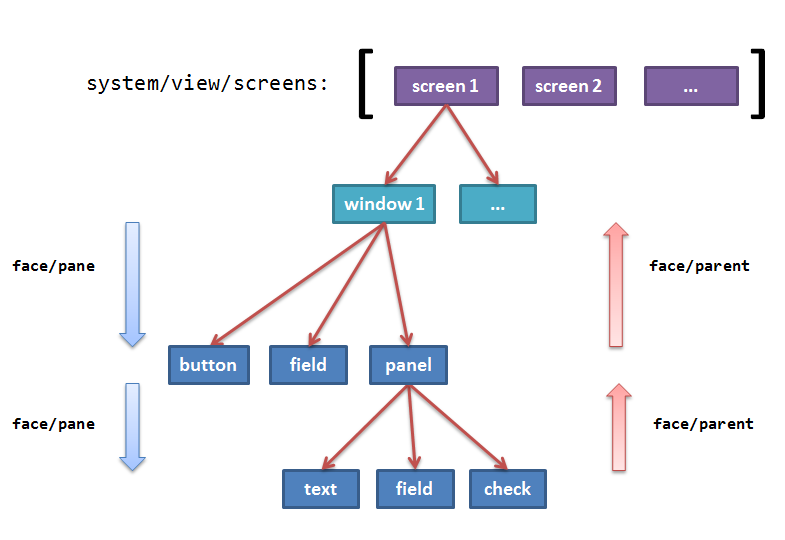
V následujícím schematu uvedené elementy screen, window, button, ..., jsou všechno **piškoty** (faces), kterých je aktuálně 18 (area, base, button,  camera, check, drop-down, drop-list,  field, group-box,  panel, tab-panel,  progress, radio, slider, screen,  text, text-list, window).

Piškot **screen** zastupuje obrazovku počítače a může obsahovat jeden či více piškotů window. Piškot **window** (okno) je kontejnerem pro další i zanořené piškoty. Tyto dva elementy mají dominantní postavení; od ostatních piškotů se liší zejména tím, že si je interpret Redu vytvoří automaticky při aplikaci některého z dalších deklarovaných piškotů.

Uvnitř piškotu **windows** lze jako kontejnery použít také piškoty **panel**, **group-box** a **tab-panel**.

Aplikací se míní realizace příkazu **view** na základě interně vytvořeného [stromu piškotů](https://doc.red-lang.org/cs/view.html#_strom_piskot_).

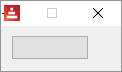
Vzájemnou hierarchii piškotů vyjadřují aspekty **pane** (zeznam dětí, na které piškot ukazuje) a **parent** (odkaz na rodičovský piškotq). Aspektů je aktuálně 23, vyjadřují vlastnosti jednotlivých piškotů a jsou nedílnou součástí každého piškotu.

Pěknou prezentaci vztahů mezi piškoty screen, window a button získáme realizací tohoto kódu:

>> **view** [b: button [**print** b]]

případně >> **view** [button [**print** face]] ; face zde má roli žolíka

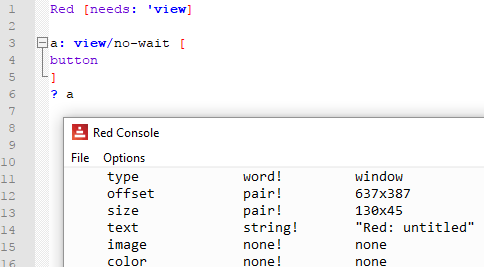
případně >> **view** [button [**probe** face]]



Stiskem levého tlačítka myši s kurzorem v tmavošedém poli spustíme provedení příkazu (**print** **b**, případně **probe** face). V okně interaktivní konzoly Redu se vytiskne výpis všech aspektů s přiřazenými hodnotami u všech tří použitých piškotů (screen, window a button). Zanoření piškotů je deklarováno v aspektu parent a pane. **Vřele doporučuji si tento postup vyzkoušet.**

Výpis aspektů v konzole je dvakrát zanořený. Piškot typu button odkazuje svým aspektem parent na piškot windows a ten zase na screen. Takto jsou jednotlivé komponenty grafického objektu vzájemně propojeny.

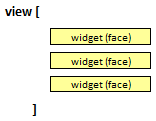
Výpis aspektů pro jednotlivé piškoty (kromě screen a window) lze také získat realizací tohoho kódu:



Zadáte-li příkaz ? a samostatně v následně otevřené konzole, kterou si předtím roztáhnete na celou šířku obrazovky, bude váš výpis přehlednější.

**Jednoduchý začátek**

Red vytváří grafické prostředí (GUI) tak, že jej popíše v bloku **view**. Tento popis je velmi přímočarý a ve své nejjednodušší formě by mohl vypadat takto:



Má-li být skript kompilován, musí mít v záhlaví Red [needs: **view**] . Je-li spouštěn z interakční konzoly Redu, není tohoto záhlaví zapotřebí, neboť konzola modul View již obsahuje.

Příklad skriptu:

Red [needs: **view**]

**view**[

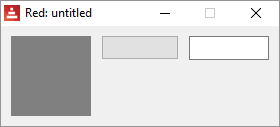
base

button

field

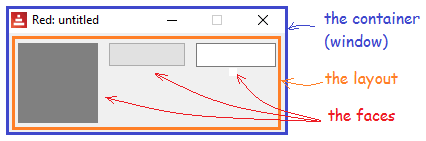
]

A jeho výsledné GUI:

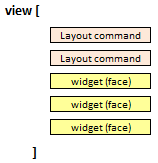


Dokumentace Redu označuje viditelné objekty grafického rozhraní názvem **face** (neboli widget, neboli **piškot**). Tyto piškoty jsou vloženy do kontejneru **windows** podle zadaného uspořádání (**layout**).

K následujícímu obrázku nutno poznamenat, že **layout** je pouze uspořádání následných piškotů base, button a field v piškotu (kontejeneru) window, nikoliv kontejner jak naznačeno :



Příkazy pro uspořádání piškotů (layout commands) se píši před vlastní definice piškotů:



V následující ukázce deklaruje příkaz below (layout command) řazení piškotů pod sebe, místo implicitního  across  (vedle sebe) první ukázky:

Red [needs: **view**]

**view**[

below ; layout command

base ; face (widget)

button ; face (widget)

field ; face (widget)

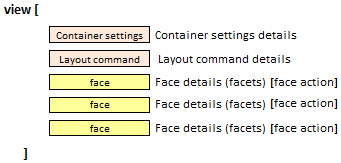
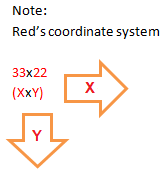
]

Výsledné GUI:



Nastavení kontejneru (size, title, backdrop)popisuje jak má vypadat samotné okno. Toto nastavení i příkazy dispozice (layout commands) umožňují podrobnější specifikaci, jako je velikost, barva, atp. Aspekty (**facets**) piškotů  určují jejich vlastnosti a vzájemné vztahy.

Součástí piškotu může být také blok příkazů (actor neboli aktér), definující akci piškotu. Tento blok je posléze obsažen v aspektu **actors**.

Příklad skriptu:

Red [needs: **view**]

**view**[

backdrop blue ; container setting

below ; layout command

base 20x20 ; face and facet

button 50x20 "press me" [**quit**] ; face, facets, actor

field red "field" ; face and facets

]

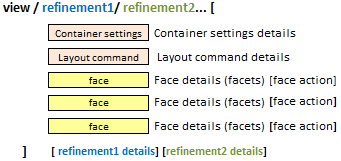
Výsledné GUI:



Red ví, co si počít s jednotlivými zadanými hodnotami již na základě jejich typu. Vidí-li pair! - ví, že jde o aspekt size; vidí-li string! - ví, že jde o aspekt text; vidí-li block! - ví, že jde o popis akce (face action), kterou má piškot zprostředkovat, uložený v aspektu actors. Zvláštním dobrodiním je skutečnost, že na pořadí aspektů nezáleží. Následující kódy rezultují v tomtéž GUI.

button 50x20 "press me" [quit] ; pair! string! block!   
button "press me" [quit] 50x20   
button [quit] 50x20 "press me"

Funkce  view umožňuje připojovat upřesnění, která mění okno samotné (nikoliv dispozici uvnitř okna). Tato upřesnění (refinements) jsou deklarována hned za slovem view a precizována ve stejně řazených samostatných blocích za hlavním blokem view :



Zavedeným upřesněním funkce view jsou **flags** (slova, měnící zobrazení nebo chování okna) a **options** (další vlastnosti okna ve formátu 'name: value'). Tato slova mají dva významy - jednak to jsou názvy aspektů a za druhé to jsou hromadná označení skupiny hodnot.

V následujícím skriptu se upřesněním /flags říká, že zobrazované okno je modálního typu (modal) a lze měnit jeho velikost (resize), zatímco upřesnění /options zajistí zobrazení okna v levém horním rohu obrazovky s posunem (offset) 50 pixelů dolů a 50 pixelů vpravo:

Red [needs: **view**]

**view**/flags/options[

size 300x100 ; container setting

below ; layout command

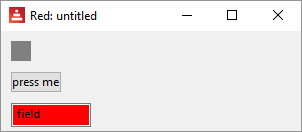
base 20x20 ; face and facet

button 50x20 "press me" [**quit**] ; face, 3facets and actor

field red "field" ; face and 2 facets

]['modal 'resize] [offset: 50x50] ; 2 flags and 1 option

Výsledné GUI:



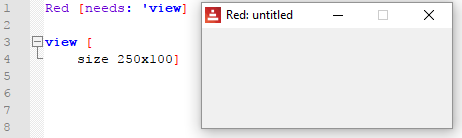
Podrobnější popis **upřesnění** pro window je uveden v následujcící kapitole.

*Created with the Standard Edition of HelpNDoc:* [*Free EPub and documentation generator*](https://www.helpndoc.com)

**GUI - Nastavení kontejneru**

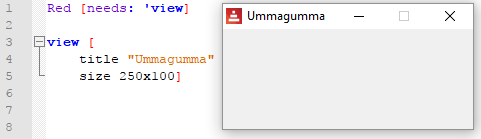
Tato nastavení definují úpravy okna (windows), které bude obsahovat další prvky GUI. Na rozdíl od hodnot aspektů je nutné hodnoty pro parametry okna uvést jejich příslušnými názvy: **size**, **title** a **backdrop**:

VID DLS**size** -Nastavuje velikost okna v pixelech.

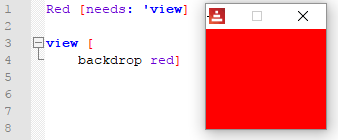


Nezadáme-li velikost okna, provede to Red za nás. Není-li okno veliké tak aby mohlo zobrazit část titulku (min. 212 x ...), nelze jej přemisťovat. Implicitní velikost okna je 110x110 px.

VID DLS**title** -Nastaví titulek v záhlaví okna.



VID DLS**backdrop** -Nastaví barvu pozadí okna.



**Upřesnění - refinements**

Další možností modifikace okna (windows) jsou upřesnění: **options, flags** a **no-wait.**Options a flags jsou aspekty piškotů a definují se v blocích za hlavním blokem  view .

**Options** je aspekt, který určuje vlastnosti piškotu window ve formátu [name: value]. Hodnotou v samostatném bloku za blokem view lze vyjádřit dva parametry okna:

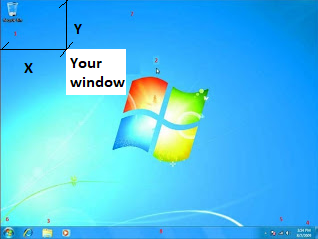
* **offset** - odsazení okna od levého horního rohu obrazovky, uváděné v pixelech ve formátu pair!
* **size** - velikost okna v pixelech ve formátu pair! (lze tedy velikost okna zadat dvojím způsobem

Red [needs: **view**]

**view/options** [ ; /deklarace upřesnění

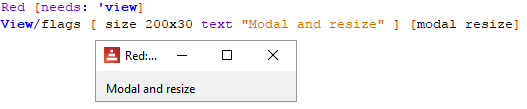
button 100x40 "click me" [**quit**]]

[offset: 100x70] ; definice upřesnění

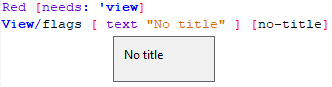


**Flags -** je aspekt, který rovněž mění zobrazení nebo chování piškotu window. Disponuje řadou parametrů (**flagů**):

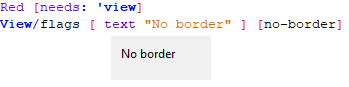
* **modal** - vyžaduje pozornost (focus) tím, že zmrazí všechna ostatní okna, dokud modální okno nezavřete. Vytvoříte-li okno s flagy modal, no-title a no-border, je velmi obtížné se jej zbavit.
* **resize** - umožní změnu velikosti okna (implicitní je stálá velikost).



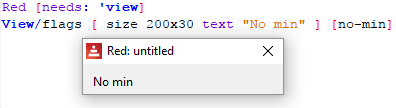
* **no-title** - nezobrazuje název okna



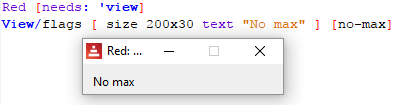
* **no-border** - odebere rámeček kolem okna



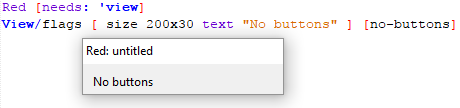
* **no-min** - eliminuje tlačítka 'min' a 'max' v záhlaví okna



* **no-max** - tlačítko 'max' se zobrazí jako neaktivní



* **no-buttons**- nezobrazí žádné tlačítko (maximize, minimize, close) v záhlaví okna



* **popup**- pouze pro Windows - činí z okna výsuvné okno (popup). Umožňuje ostatním oknům zůstat aktivní. Zavře se při přesunu fokusu na jiné okno.

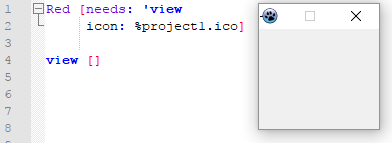
**No-wait** - je upřesnění pro funkci view, přikazující aby nebyla aplikována implicitní smyčka událostí ( jaká že? ...)

**Actors**

Viz dedikovanou[kapitolu](#_topic_Udalostiaakteri).

**Zadání ikony -** Chodí jenom při kompilaci kódu, nikoliv při jeho interpretaci!

Nejde o nastavení okna ale snad se to sem hodí. Chcete-li zadat svoji ikonu do okna, musíte ji přidat v deklaraci záhlaví skriptu :: icon: <path-to-icon>:



*Created with the Standard Edition of HelpNDoc:* [*Free Qt Help documentation generator*](https://www.helpndoc.com)

**GUI - Layout**

Rozložení (layout) piškotů uvnitř piškotu **window** je řízeno těmito pokyny: across, below, return, space, origin, at, pad. Implicitní nastavení pro window je:

* + počátek (origin) : *10x10*
  + mezera (space) : *10x10*
  + směr (direction) : *acros*
  + přiřazení (alignment): *top*

Přiřazení může pro pokyny across, below a return nabývat hodnot *top, middle, bottom, left, center, right nechodí, neznám syntaxi*.

VID DLS **across** (top | middle| bottom)

Red [needs: **view**] ; "needs: view" is needed if the script is going to be compiled

**view** [

across ; nepovinné, je to implicitní hodnota

area 20x20 red

area 20x20 blue

area 20x20 green

]



VID DLS **below** (left |center | right)

Red [needs: **view**]

**view** [

below

area 20x20 red

area 20x20 blue

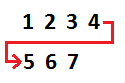
area 20x20 green

]



VID DLS**return**

Mění zadaný mód **across**:



Red [needs: **view**]

**view** [

across

area 20x20 red

area 20x20 blue

**return**

area 20x20 green

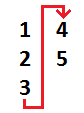
area 20x20 gray

area 20x20 yellow

]



Mění zadaný mód **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**

Zadá odsazení (offset), platné pro následné piškoty.

Red [needs: **view**]

**view** [

across

**space** 50x10

area 20x20 red

area 20x20 blue

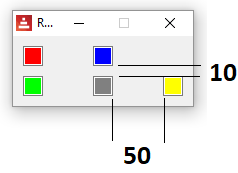
**return**

area 20x20 green

area 20x20 gray

area 20x20 yellow

]



VID DLS**origin**

Nastaví offset prvního piškotu od levého horního rohu okna.

Red [needs: **view**]

**view** [

across

origin 70x20

area 20x20 red

area 20x20 blue

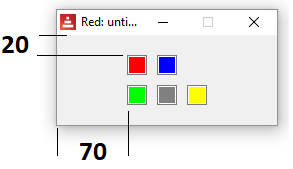
**return**

area 20x20 green

area 20x20 gray

area 20x20 yellow

]



VID DLS**at**

Vloží následující jeden piškot do absolutně zadané pozice. Neovlivňuje umístění ostatních piškotů.

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**

Mění dispozici okna relativním odsazením následujících piškotů od původně zadané polohy.

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**

Jde o exekuční příkaz, popsaný v kapitole [Running code](#_topic_Exekucekodu). V tomto případě je použit pro spuštění kódu uvnitř kontejneru view.

Lze bez problémů provést toto:

Red [needs: 'view]   
a: 33 + 12  
print a ; vytiskne v konzole   
view [   
 text "hello" ; zobrazí okno s textem   
]

Toto však vyvolá chybové hlášení:

Red [needs: 'view]   
view [   
 text "hello"   
 a: 33 + 12 ;ERROR!!!   
 print a   
]

Uvnitř okna musí být blok podřízen příkazu "do":

Red [needs: 'view]   
view [   
 text "hello"   
 do [a: 33 + 12 print a] ;OK!   
]

*Created with the Standard Edition of HelpNDoc:* [*Free PDF documentation generator*](https://www.helpndoc.com)

**GUI - Piškoty (faces)**

Piškoty neboli **faces** (neboli widgets)**,** jsou základní stavební kameny grafického rozhraní. Interní součástí každého piškotu je sada 23 **aspektů** (facets), jimiž jsou určovány jednotlivé parametry piškotu. Těmito parametry je určován nejen vzhed piškotů, ale i jejich odezva na různé události a souvislost s jinými piškoty.

**Seznam aspektů** (facets)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Facet** | **Datatype** | **Povinné?** | **Použití** | **Popis** |
| type | word! | yes | all | Typ grafické komponenty |
| offset | pair! | yes | all | Ofset reflektovaného kurzoru od počátku vlevo nahoře |
| size | pair! | yes | all | Velikost piškotu |
| text | string! | no | all | Popisek, zobrazený v piškotu |
| image | image! | no | some | Obraz na pozadí piškotu |
| color | tuple! | no | some | Barva pozadí ve formátu R.G.B nebo R.G.B.A. |
| menu | block! | no | all | Lišta menu nebo kontextuální nabídky. |
| data | any-type! | no | all | Data o piškotu. |
| enable? | logic! | yes | all | Povolit nebo zakázat události v piškotu |
| visible? | logic! | yes | all | Zobrazit či skrýt piškot. |
| selected | integer! | no | some | Index aktuálně vybraného elementu seznamu. |
| flags | block!, word! | no | some | Klíčová slova, měnící zobrazení nebo chování piškotu. |
| options | block! | no | some | Další vlastnost piškotu ve formátu [name: value] |
| parent | object! | no | all | Odkaz na rodičovský piškot, pokud existuje. |
| pane | block! | no | some | Seznam dětí, zobrazených uvnitř piškotu. |
| state | block | no | all | Info o interním stavu piškotu *(pouze u View engine)*. |
| rate | integer!, time! | no | all | Časovač piškotu. Integer udává frequenci, time udá- vá trvání, none zastavuje. |
| edge | object! | no | all | *(reserved for future use)* |
| para | object! | no | all | Pokyny pro umístění textu. |
| font | object! | no | all | Nastavení atributů fontu |
| actors | object! | no | all | Uživatelsky vytvořené ovladače událostí. |
| extra | any-type! | no | all | Volitelná uživatelská data připojená k piškotu. |
| draw | block! | no | all | Seznam příkazů Draw, jež mají být provedeny. |

**Datové typy hodnot aspektů:**

Hodnoty aspektů se zadávají přímo, bez uvedení jejich určení, které interně provádí interpret na základě jejich datového typu.

|  |  |  |
| --- | --- | --- |
| **Datový typ** | **Aspekt** | **Účel použití** |
| integer! | rate, selected | Určuje šířku piškotu, výška zůstává implicitní. U panelů udává počet řad či sloupců. |
| pair! | offset, size | Určuje šířku a výšku piškotu. |
| tuple! | color | Určuje barvu pozadí piškotu ve formátu rgb, např. 211.9.84. |
| issue! | color | Určuje barvu pozadí piškotu použitím hexadecimálního zápisu, například #d30954. |
| word! | color | Určuje barvu pozadí piškotu slovem, např. aqua |
| string! | text | Určuje text, zobrazovaný piškotem. |
| percent! |  | Nastavuje aspekt data (užitečné pro piškoty progress a slider). |
| logic! |  | Nastavuje aspekt data (užitečné pro piškoty check a radio). |
| image! | image | Nastavuje obrázek, zobrazený jako pozadí piškotu. |
| url! |  | Načte zdroj, na nějž URL ukazuje. |
| block! |  | Udává akci pro implicitní či deklarovanou událost piškotu. U panelů určuje obsah. |
| get-word! |  | Jako aktéra používá existující funkci. |
| time! |  | Určuje časový úsek delší než 1 sec v aspektu rate |

Příklad aplikace aspektů:

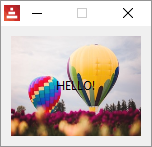
Red [needs: **view**]

**view** [

base "HELLO!" 130x100 %balloon.jpeg ; base je piškot, následují hodnoty aspektů text, size, image

]

(soubor %baloon.jpegmusí být uložen ve stejném adresáři jako prováděčka red.exe)



**Atributy aspektu font**

Aspekt font má 9 předdefinovaných formátovacích atributů: name, size, style, angle, color, anti-alias?, shadow, state, parent. Prostřednictvím atributu parent mohou na daný aspekt odkazovat i jiné piškoty.

Při deklaraci se název atributu uvádí za pomlčkou, například **font-name**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pole** | **Datový typ** | **Povinné?** | **Popis** |
| - **name** | string! | ne | Název fontu, instalovaného v OS |
| - **size** | string! | ne | Velikost fontu v bodech (points) |
| - **style** | word!, block! | ne | Styly fontů: bold, italic, underline a strike |
| - **angle** | integer! | ano | Sklon textu ve stupních |
| - **color** | tuple! | ano | Barva fontu ve formátu RGB nebo RGBA |
| - **anti-alias?** | logic!, word! | ne | Režim vyhlazení |
| - **shadow** | *(reserved)* | ne | *(reserved for future use)* |
| - **state** | block! | ne | Interní stav piškotu *(pouze ve View)* |
| - **parent** | block! | ne | Interní odkaz na rodičovský piškot |

Příklad použití:

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

] ; piškot, aspekt, atributy fontu, příslovce



**Klíčová slova (keywords):**

Klíčovými slovy (typu word!) se přiřazují slovní hodnoty různým aspektům (text, extra, data, draw, font, para, select) Jsou to tato slova:

left, center, right, top, middle, bottom, bold, italic, underline, strike, extra, data, draw, font, para, wrap, no-wrap, font-size, font-color, font-name, react, loose, all-over, hidden, disabled, select, focus, hint, rate, default.

Případně slouží jako hodnoty upřesnění při formování piškotu (window):

* /flags :: modal, resize, no-title, no-border, no-min, no-max, no-buttons, popup
* /options:: on-(enter, change, select, key)

**Aktéři - actors**

Slovo **actors** (aktéři) je název jednoho z aspektů a rovněž společné označení funkcí pro ošetření událostí. Aktér (actor) je funkce, která popisuje aktivitu piškotu při výskytu jisté události - viz [další kapitola](#_topic_Udalostiaakteri).

**Piškoty - faces**

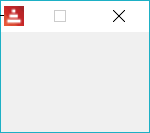
Knihovna **view** disponuje těmito **piškoty**: area, base, button, camera, check, drop-down, drop-list, field, group-box, panel, tab-panel, progress, radio, slider, screen, text, text-list, window, . Jednotlivé piškoty jsou v následujcím textu podrobněji popsány.

Rozšířením piškotu **text** jsou piškoty **h1** až **h5**, rozšířením piškotu **base** jsou piškoty **box** a **image**.

Nejvýše postaveným objekty jsou diskrétní piškoty **screen** a **window**. Tyto piškoty se nedají samostatně deklarovat. Jsou však automaticky přítomny při každé vizualizaci bloku funkcí **view**, například i prázdného:

Red [needs: **view**] ; "needs: view" je potřebné pouze při kompilaci skriptu

**view** []



**Aktivita piškotů**

Většina piškotů (vyjímaje screen a window) si ví rady s výskytem nějaké události. Tou událostí může být kliknutí myší klávesy, stisknutí klávesy na klávesnici nebo provedení výběru. Výpis těchto událostí je uveden na počátku další kapitoly. Implicitně nastavené události (ovladače) jsou dále uvedeny u každého piškotu.

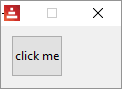
Pro tlačítka (buttons) je touto implicitně ošetřenou událostí stisk levé klávesy myši (down) a v následující ukázce spouští příkaz  quit, který končí program. Místo funkce [quit] lze použít například [print "Tornado Loo"].

Red [needs: **view**]

**view** [

button 50x40 "click me" [**quit**] ; piškot, aspekt, aspekt, aktér

]



Citlivost piškotu na událost se projeví jen v tom případě, je-li deklarace piškotu doplněna blokem s názvem definované funkce nebo s definicí uživatelské funkce - neboli aktérem.

**Přehled piškotů**

**base** (on-down)

Piškot **base** bez parametru zobrazí uvnitř piškotu **window** (jehož rodičem je piškot **screen**) tmavošedý (128.128.128) čtverec (80x80 px) bez ohraničení.

Red [needs: **view**]

**view** [

base

]



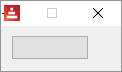
**button** (on-click)

Red [needs: **view**]

**view** [

button

]



**check** (on-change)

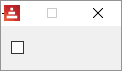
Piškot reaguje na změnu stavu zatržení uživatelem.

Red [needs: **view**]

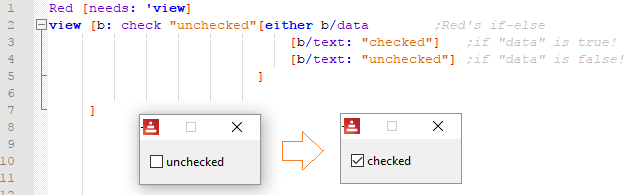
**view** [

check

]



Aktuální stav zatržítka je uložen v aspektu***data*** *(true nebo false)*



**radio** (on-change)

Událost, registrující u piškotu vstup textu nebo výběr ze seznamu, se nazývá *change.*

Při každé volbě (zde on*,* off*,* uh?) umožní realizaci příslušného aktéra - příslušného bloku.

Piškoty radio tvoří obsah kontejneru **window**. V kontejneru může být zatrženo pouze jedno výběrové tlačítko **radio**.

Red [needs: **view**]

**view** [

r1: radio "on" [t/text: "on"]

t: text "none" ; jen před první volbou

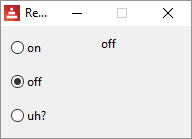
**return**

below

r2: radio "off" [t/text: "off"]

r3: radio "uh?" [t/text: "uh?"]

] ; proměnná, piškot, aspekt, aktér



**field** (on-enter)

Piškot field slouží k vložení textu.

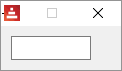
Text v poli piškotu field je uložen v aspektu***data****.* Chodí to však obojím směrem. Změníte-li obsah aspektu ***data*** (*.../data value*), změní se zobrazený text v poli.

Red [needs: **view**]

**view** [

field

]



V této ukázce se vytiskne zápis v poli do konzoly při (každém) stisku Enter:

Red [needs: **view**]

**view** [

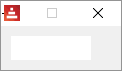
f: field [**print** f/text]

] ; proměnná, piškot, aktér

Piškot field podporuje flag no-border :

Red [needs: **view**]

**view** [field no-border]



**area** (on-change)

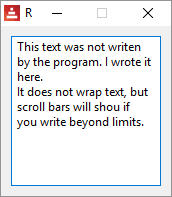
Text uvnitř piškotu area  je uložen v aspektu ***text****.*Do zobrazeného okna lze psát víceřádkový text:

Red [needs: **view**]

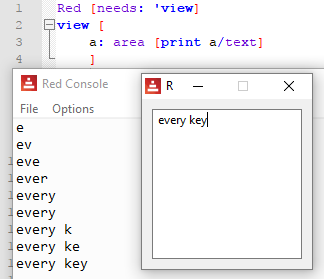
**view** [

area

]



Živější aplikaci vytvoříme připojením aktéra *[print a/text],* který tiskne aktuální obsah aspektu ***text*** do konzoly. Aktér je aktivován stiskem každé klávesy na klávesnici, včetně příkazu Enter:



**text-list** (on-change)

Procházené řetězce jsou uloženy v aspektu ***data***. Index vybraného řetězce je v aspektu ***selected***:

Red [needs: **view**]

**view** [

tl: text-list 100x100 data[

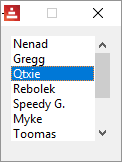
"Nenad" "Gregg" "Qtxie" "Rebolek"

"Speedy G." "Myke" "Toomas"

"Alan" "Nick" "Peter" "Carl"]

[**print** tl/selected ]

]



3  *; index vybrané položky se vytiskne do konzoly*

===

To use the string selected, the code snippet could be: nevím, jak to připojit

pick face/data face/selected

[ t/text: **pick** face/data face/selected ] viz drop-down

This would be the same as : pick ["Nenad" "Greg" "Qtxie" "Rebolek" (...)] 3

===

**progress** (on-change)

Aktuální stav posuvníku je uložen v aspektu data, jako hodnota typu percent! nebo float! v rozsahu mezi 0 a 1.

Red [needs: **view**]

**view** [

below

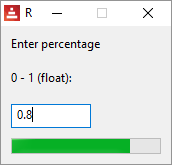
text "Enter float"

text "0 - 1 (float):"

field [p/data: **to** **percent!** face/data]

p: progress

]



**slider** (on-change)

Aktuální procentní hodnota je uložena v aspektudatajako datový typ percent! nebo float!.

Red [needs: **view**]

**view** [

title "slider"

t: text "Percentage"

slider 100x20 data 10% [t/text: **to** **string!** face/data]

] ; označení aspektu 'data' je nadbytečné ale užitečné pro porozumnění kódu

Pohybem jazýčku měníte hodnotu procenta. Implicitně nastavená hodnota je 10% :



Všimneme si zde, jak je zadána hodnota aspektu data. V tomto případě by šlo slovo 'data' vynechat - na rozdíl od deklarace piškotu text-list, kde přiřazovanou hodnotou k aspektu data byl blok.

**panel** (on-down)

Vytváří společný piškot (uvnitř piškotu window a screen) pro umístění dalších piškotů.

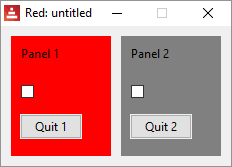
Red [needs: **view**]

**view** [

panel red [size 100x120 below text "Panel 1" check button "Quit 1" [**quit**]]

panel gray [size 100x120 below text "Panel 2" check button "Quit 2" [**quit**]]

]



**group-box** (on-down)

Group-box je kontejner podobný kontejneru panel. Rovněž slouží pro umístění dalších piškotů. Uvažuje se, že se zruší po zavedení aspektu **edge**.

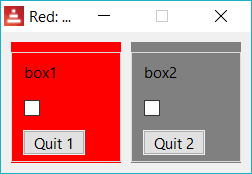
Red [needs: **view**]

**view** [

group-box [size 110x120 below text "box1" check button "Quit 1" [**quit**]]

group-box gray [size 110x120 below text "box2" check button "Quit 2" [**quit**]]

]



**tab-panel** (on-select)

Vytvoří sadu karet s oušky, formou překrývajících se panelů v piškotu window. Každý panel může obsahovat více piškotů. Informace o sestavě jsou uloženy v těchto aspektech:

<tab-panel>/data - blok s názvy oušek (string!).  
<tab-panel>/pane - odkaz na dětské piškoty (block!).  
<tab-panel>/selected - index vybraného panelu nebo hodnota none .

Na rozdíl od piškotu group-box je piškot tab-panel expilicitně deklarován jen jednou:

Red [needs: **view**]

**view** [

Title "Tab-panels"

tab-panel 200x100 [

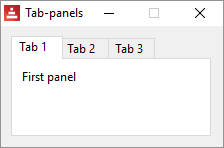
"Tab 1 " [text "First panel"]

"Tab 2 " [text "Second panel"]

"Tab 3 " [text "Third panel"]

]

]



Ukázka karty s více piškoty:

Red [needs: **view**]

**view** [

Title "Tab-panels"

tab-panel 110x140 [

"Tab 1 " [

below

text font-color blue "First panel"

button "quit" [**quit**]

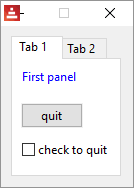
check "check to quit" [**quit**]

]

"Tab 2 " [text "Second panel"]

]

]



**drop-down** (on-select, on-change, ~~on-enter~~)

Roletka s vertikálním seznamem řetězců. Aspekt data příjímá jakékoli hodnoty ale pouze řetězce jsou přidávány do seznamu a zobrazeny. Neřetězcové hodnoty lze použít pro vytvoření asociativních řad (arrays) s použitím řetězců jako klíčů.

Aspekt selected obsahuje celočíselný index vybrané položky. Aspekt text obsahuje aktuálně vybranou položku. Zobrazí se při stisku Enter.

Red [needs: **view**]

**view** [

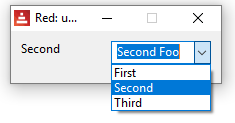
t: text "-->"

drop-down "Choose one" data [ ; piškot, aspekt text, aspekt data

"First" "Second" "Third" ] [

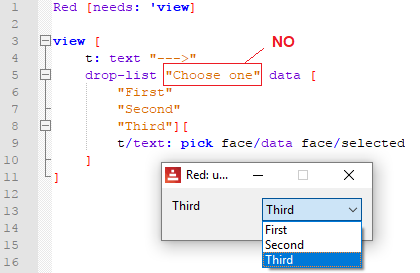
t/text: **pick** face/data face/selected ] ; aktér

]



**drop-list** (on-change)

Stejné jako drop-down, akorát že se nezobrazí řetězec aspektu text a název položky se projeví hned po jejím výběru v roletce.



**text** (on-down)

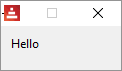
Piškot **text**  je autonomní entita s rodiči **window** a **screen** (případně panel, případně jiný piškot).

Red [needs: **view**]

**view** [

text "Hello" ; **piškot** text plus hodnota **aspektu** text

]



**camera** (on-down) Nutno doplnit komentář

**box** (on-down)  **image** (on-down)

Piškoty **box** a **image** jsou variantami piškotu **base** (type: ´base).

**box** bez parametru vymezí v nadřazeném piškotu window transparentní čtverec (80x80 px) bez ohraničení

**image** bez parametru vymezí v nadřazeném piškotu window bílý čtverec (100x100 px) bez ohraničení. Image s odkazem na soubor vrátí zobrazení tohoto souboru.

Red [needs: **view**]

**view** [

base

box

image

image %smallballoon.jpeg

]



**h1 až h5** (on-down)

Piškoty**h1, h2, h3, h4** a **h5** jsou variantami piškotu **text** (type: ´text).

Red [needs **view**]

**view** [

below

h1 "Hello"

h2 "Hello"

h3 "Hello"

h4 "Hello"

h5 "Hello" [print "Halali"] ; jediný aktivní piškot

]



Introspekcí kódu lze zjistit, že i tyto piškoty jsou deklarovány stejnými sadami 23 aspektů, stejně jako základní piškoty.

**Aspekt text**

Aspekt **text** je pole piškotu pro uložení textu.

Red [needs: **view**]

**view** [

button "hello" ; piškot button plus hodnota aspektu text

button "bold" bold ; dtto plus klíčové slovo

button "underline" underline

button "strike" strike

**return**

button "top" 70x70 top ; hodnoty aspektů text, size plus keyword

button "middle" 70x70 middle ; dtto

button "bottom" 70x70 bottom

**return**

button "left" 70x70 left

button "center" 70x70 center

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 ; right nechodí!

]



**Aspekt color**

K zadání barvy pozadí slouží aspekt **color**. Jeho hodnotu lze zadat jako literál ve formátu word! (např. aqua) nebo tuple! (0.255.255) či issue! (#00ffff) - bez uvedení názvu aspektu. V následující ukázce vidíme deklaraci 32 piškotů base uvnitř automaticky generovaného piškotu window:

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"

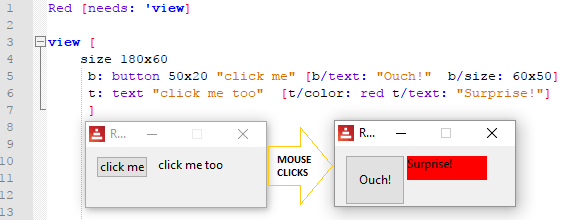
]

Dostaneme tuto pěknou paletu. Kdybychom popis barev v uvozovkách použili bez anotace text, šlo by o aplikaci **aspektu text** a popisky by byly uvnitř barevných polí. Takto jde o deklaraci **piškotu text** se zadanou hodnotou aspektu text.



**Dynamická změna aspektů**

Při běhu programu lze měnit hodnoty jejich aspektů (facets):



Všimněte si, že při deklaraci jednotlivých aspektů piškotu zadáváme přímo jejich příslušné hodnoty (50x20 "click me"), zatímco při změně hodnot (zde v hranatých závorkách) uvádíme rovněž název příslušného aspektu (b/text: "Ouch!"), přičemž můžeme uplatnit i aspekt původně nepoužitý (t/color: red), případně můžeme odkázat na aspekt jiného piškotu, například:

b: button 50x20 "click me" [b/text: "Au!" t/size: 60x50] ;vyzkoušejte si to

 Na objekt piškotu se lze také odkázat slovem face/<attribute>:



*Created with the Standard Edition of HelpNDoc:* [*Generate Kindle eBooks with ease*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

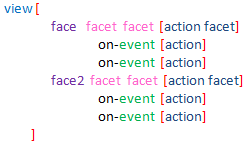
**GUI - Události a aktéři**

**Události - events:**

Stisk klávesy na klávesnici či myši, přejíždění kurzorem přes zaměřené pole na obrazovce, výběr položky - to všechno jsou události, které lze podchytit vhodně napsanou definicí piškotu a spojit je s nějakou aktivitou - implicitně definovanou nebo uživatelsky určenou. V následující tabulce je výpis aktuálně použitelných názvů událostí:

|  |  |  |
| --- | --- | --- |
| **Jméno** | **Prostředek** | **Akce** |
| **down** | mouse | Stisknuté levé tlačítko myši. |
| **up** | mouse | Uvolněné levé tlačítko myši. |
| **mid‑down** | mouse | Stisknuté prostřední tlačítko myši. |
| **mid‑up** | mouse | Uvolněné prostřední tlačítko myši. |
| **alt‑down** | mouse | Stisknuté pravé tlačítko myši. |
| **alt‑up** | mouse | Uvolněné pravé tlačítko myši. |
| **aux‑down** | mouse | Stisknuté pomocné tlačítko myši. |
| **aux‑up** | mouse | Uvolněné pomocné tlačítko myši. |
| **drag‑start** | mouse | Piškot začíná být tažen. |
| **drag** | mouse | Piškot je tažen. |
| **drop** | mouse | Od tažení piškotu bylo upuštěno. |
| **click** | mouse | Klik levým tlačítkem myši (pouze pro button). |
| **dbl‑click** | mouse | Dvojklik levým tlačítkem myši. |
| **over** | mouse | Kurzor myši přechází přes piškot. K této události dochází nejprve když kurzor vstupuje na piškot a poté když jej opouští. Pokud aspekt flags obsahuje flag **all‑over**, potom jsou rovněž produkovány všechny mezilehlé události. |
| **move** | mouse | Došlo k pusunu okna. |
| **resize** | mouse | Byla změněna velikost okna. |
| **moving** | mouse | Okno je posouváno. |
| **resizing** | mouse | Mění se velikost okna. |
| **wheel** | mouse | Otáčí se kolečko myši. |
| **zoom** | touch | A zooming gesture (pinching) has been recognized. |
| **pan** | touch | A panning gesture (sweeping) has been recognized. |
| **rotate** | touch | A rotating gesture has been recognized. |
| **two‑tap** | touch | A double tapping gesture has been recognized. |
| **press‑tap** | touch | A press-and-tap gesture has been recognized. |
| **key‑down** | keyboard | Stisknutá klávesa klávesnice. |
| **key** | keyboard | Byl zadán znak nebo stisknuta speciální klávesa (kromě kláves Ctrl, Shift a Menu). |
| **key‑up** | keyboard | Uvolněná klávesa klávesnice. |
| **enter** | keyboard | Stisknutá klávesa Enter. |
| **focus** | any | Piškot právě přešel do režimu *focus*. |
| **unfocus** | any | Piškot právě ztratil *focus*. |
| **select** | any | V piškotu je prováděn výběr z více možností |
| **change** | any | V piškotu došlo ke změně v důsledku vstupu uživatele (vložení textu nebo výběr ze seznamu). |
| **menu** | any | Výběr položky z menu. |
| **close** | any | Zavření okna. |
| **time** | timer | Proběhla prodleva, nastavená aspektem rate piškotu. |

Některé události jsou definičně spojeny s daným piškotem, případně aspektem (například aspekt rate má časovač on-time). Každý piškot je schopen akceptovat jenom vymezenou sadu událostí. Uživatelsky definované připojení události k piškotu se provádí za výčtem aspektů:



**Příklady některých událostí**

Všechny příklady si vyzkoušejte v konzole. Termín např. on-down je spojení obecné předpony on- a názvu události z tabulky.

**down, over** :

Red [needs: **view**]

**view** [

t: area 40x40 blue

on-down [**quit**]

on-over [**either** t/color = red [t/color: blue][t/color: red]]

]

**wheel** :

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** :

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 [

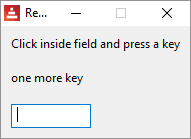
t/text: **first** list

list: **next** list

**if** **tail?** list [list: **head** list]

]

]



**time** :

V tomto příkladě bliká zadaný text zadanou frekvencí (zde jednou za vteřinu - viz kapitola [GUI- Pokročilá témata](#_topic_Pokrocilatemata)):

Red [needs: **view**]

**view** [

t: text "Now you see..." rate 1

on-time [**either** t/text = "" [t/text: "Now you see..."][t/text: ""]]

]

**close** :

- událost okna (kontejneru) - při zavření okna se má v konzole vytisknout "bye!" - (ale nevytiskne)

Red [needs: **view**]

**view** [

button [**print** "click"]

on-close [**print** "bye!"]

]

**Aktéři - actors**

Slovo **actors** (aktéři) je název jednoho z aspektů a rovněž společné označení funkcí pro ošetření událostí. Tyto funkce jsou definovány v bloku za návěstím  on-<event> . Výpis případných událostí je uveden v tabulce na začátku této kapitoly.

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]]

]

Spuštění tohoto kódu v konzole je doprovázeno obsáhlým tiskovým výstupem. Ke konci tohoto výstupu vidíme deklaraci hodnoty aspektu **actors:**

(...)

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

(...)

Tiskový výstup do konzoly reprezentuje vnitřní náhled interpreta na skladbu realizovaného kontejneru. Jde o výpis všech aspektů (type, offset, size, text, image, color, menu, data, enabled?. visible?, selected, flags, options, parent, pane, state, rate, edge, para, font, actors, extra a draw) i s příslušejícími hodotami.

Vidíme, že první aspekt parent má ve svém definičním bloku rovněž kompletní sestavu aspektů, stejně jako jeho zanořený další aspekt parent.

Poznámka:

* Dotekové (touch) události nejsou realizovatelné ve Windows XP.
* Jedna či více pohybových událostí vždy předchází událost move.
* Jedna či více událostí měnících rozměr vždy předchází událost resize.

*Created with the Standard Edition of HelpNDoc:* [*Easily create EPub books*](https://www.helpndoc.com/feature-tour)

**GUI - Introspekce událostí**

**Datový typ event!**

Kromě názvů jednotlivých událostí (viz předchozí kapitola) existuje entita, zvaná datový typ **event!**, který obsahuje 13 funkcí, použitelných k introspekci prováděného kódu.

**Instance datového typu event!**

|  |  |  |
| --- | --- | --- |
| **Název** | **Typ hodnoty** | **Vracená hodnota** |
| **type** | word! | Název události |
| **face** | object! | Piškot, v němž k události došlo |
| **window** | object! | Okno, v němž k události došlo |
| **offset** | pair! | Souřadnice kurzoru myši, relativně k piškotu při výskytu události. Pro události gest vrací souřadnice středního bodu. |
| **key** | char! word! | Stlačení klávesnice |
| **picked** | integer! percent!  word! | Nová položka, vybraná v piškotu (integer! percent!). Při události myši down v piškotu text-list vrací index položky pod kurzorem nebo none. Při události  wheel  vrací počet (+/-) otočení kolečka. Při události menu vrací odpovídající ID položky (word!). Při gestu zoom vrací procento relativního zvětšení/zmenšení. U jiných gest je jeho hodnota závislá na OS (Windows: pole allArguments ze struktury [GESTUREINFO](https://msdn.microsoft.com/en-us/library/windows/desktop/dd353232(v=vs.85).aspx)). |
| **flags** | block! | Vrací výpis s jedním či více flagy (away, down, mid-down, alt-down, aux-down, control, shift) |
| **away?** | logic! | Vrací true, opouští-li kurzor hranice piškotu . Platí pouze při aktivní události over. |
| **down?** | logic! | Vrací true, byla-li stisknuta levá klávesa myši |
| **mid-down?** | logic! | Vrací true, byla-li stisknuta střední klávesa myši |
| **alt-down?** | logic! | Vrací true , byla-li stisknuta pravá klávesa myši |
| **ctrl?** | logic! | Vrací true , byla-li stisknuta klávesa CTRL |
| **shift?** | logic! | Vrací true , byla-li stisknuta klávesa SHIFT |

**Seznam možných flagů** v poli event/flags: away, down, mid-down, alt-down, control, shift

Pokaždé, když v piškotu dojde k události typu event!, lze o ní získat informaci příkazem

>> event/ <název typu>

**Událost myši:**

V níže uvedeném zjednodušeném příkladu vytiskneme typ události a souřadnice kurzoru při stisku myší klávesy (událost down)

Red [needs: **view**]

**view** [

base 100x100

on-down [

**print** event/type

**print** event/offset

]

]

down  
39x57  
down  
86x43

**Událost klávesnice:**

Podobné informace získáme při vhodné volbě události pro klávesnici - zde například událost key:

Red [needs: **view**]

**view** [

area 100x100

on-key [

**print** event/type

**print** event/offset ; chodí podívně

**print** event/key

]

]

key  
-59x84  
r  
key  
-36x59  
s  
key  
-116x79  
o

Zdá se, že některé piškoty negenerují události klávesnice. Když například ve výše uvedeném příkladu zaměníme piškot area za base, nedostaneme v konzole žádný výsledek.

**Velikost obrazovky:**

>> print system/view/screens/1/size  
1366x768

I don't have more information about system, except what you can get typing help system.

*Created with the Standard Edition of HelpNDoc:* [*Create cross-platform Qt Help files*](https://www.helpndoc.com/feature-tour/create-help-files-for-the-qt-help-framework)

**GUI - Pokročilá témata**

VID DLS **style**

Někdy je zapotřebí nadefinovat si vlastní piškot. Pro tento účel použijeme funkci **style**. Pěknou ukázku tohoto postupu lze nalézt (anglicky) [zde](http://red.qyz.cz/writing-style.html) .

Deklarace nového piškotu má formát: style <nový název> <stávající piškot> <parametry>

Red [Needs: **view**]

**view** [

style myface: base 70x40 cyan [**quit**]

myface "Click to quit" ; hodnota pro aspekt text

myface "Here too"

panel red 90x110 [

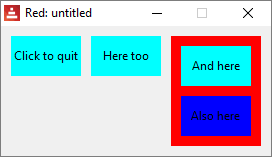
below

myface "And here"

myface "Also here" blue ; změna hodnoty aspektu color

]

]



function! **view** a **unview**

**Vícero oken v obrazovce**

Funkci view lze také použít k zobrazení oken, jež byla definována v jiné části kódu. Ovšemže, funkce unview zobrazení (view) ruší. Následující kód vytvoří dvě identická, ale nezávislá okna (rozdílné face trees) v různých místech obrazovky:

Red [needs: **view**]

my-view: [button {click to "unview"} [**unview**]]

; face string actor

**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 s upřesněním /only se zaměří pouze na zadané okno:

Red [needs: **view**]

v1: **view**/options/no-wait [

backdrop blue

button "unview blue"[**unview**/only v1]

button "unview yellow" [**unview**/only v2] ]

[offset: 30x100] ; option

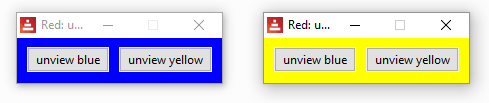
v2: **view**/options/no-wait [

backdrop yellow

button "unview blue"[**unview**/only v1]

button "unview yellow" [**unview**/only v2] ]

[offset: 400x100] ; option



Upřesnění pro funkci view:

**/tight** => nulový odstup od výchozího bodu

**/options** =>

**/flags** =>

**/no-wait** => žádné čekání

Upřesnění pro funkci unview:

**/all**  => Zavře všechna zobrazení

**/only** => Zavře zadané zobrazení

VID DLS**loose**

Klíčové slovo (příslovce) loose  umožňuje vlečení piškotu myší po okně:

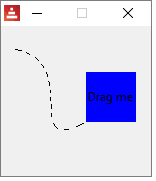
Red [needs: **view**]

**view** [

size 150x150

base blue 50x50 "Drag me" loose

] ; piškot; aspekty color, size, text; příslovce



VID DLS**all-over**

K události on-over normálně dochází v okamžiku, když kurzor "vstupuje" nebo "opouští" plochu piškoutu. Zadáte-li flag (příslovce) all-over , potom všechny události, k nimž dojde v situaci, kdy je kurzor v ploše piškotu (např. pohyby nebo kliknutí), generují událost on-over.

V následující ukázce mění levý čtverec barvu pouze při "vstupu" či "opuštění" piškotu kurzorem, avšak pravý čtverec mění barvu také při kliknutí nebo pohybu kurzoru v ploše piškotu:

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

; proměnná, piškot, aspekt, aspekt, příslovce

on-over [**either** b/color = red [b/color: blue][b/color: red]]

] ; událost, aktér s variantními reakcemi



VID DLS**hidden**

Příslovce hidden činí piškot neviditelným. Možné použití je pro vytvoření skrytého piškotu s využitým aspektem rate (zde není použito), takže lze mít časování bez zobrazení piškotu.

Red [needs: **view**]

**view** [

button "I'm here"

button "I'm not" hidden

button "Here too"

]



VID DLS**disabled**

Příslovce disabled deaktivuje piškot (nedojde k žádné události, dokud není enabled).

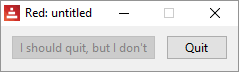
Red [needs: **view**]

**view** [

button "I should quit, but I don't" disabled [**quit**]

button "Quit" [**quit**]

]



VID DLS**select**

Příslovce  select  označí přednastavený výběr v piškotu text-list.

Red [needs: **view**]

**view** [

tl: text-list 100x100 data [

"Nenad" "Gregg" "Qtxie" "Rebolek"

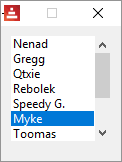
"Speedy G." "Myke" "Toomas"

"Alan" "Nick" "Peter" "Carl"

] **select** 6

[**print** tl/selected]

]



VID DLS**focus**

Příslovce **focus** dodá implicitní zaměření označenému piškotu při prvním zobrazení okna. Zaměření může mít pouze jeden piškot. Je-li deklarováno více focusů, má implicitní zaměření ten poslední.

Red [needs: **view**]

**view** [

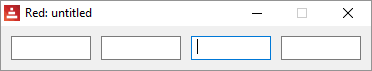
field

field focus

field focus

field

]



VID DLS**hint**

Příslovce **hint** zprostředkuje textové sdělení uvnitř piškotu field. Při zadání nového obsahu (editací pole nebo aspektu face/text) stávající text zmizí.

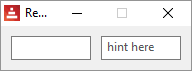
Red [needs: **view**]

**view** [

field

field hint "hint here"

]



VID DLS**default**

příslovce **default** definuje počáteční hodnotu pro aspekt data, vrací-li konverze hodnoty aspektu text hodnotu none. Aktuálně chodí pouze pro piškoty text a field. Nepochopil.

Red [needs: **view**]

**view** [

a: field 100 default "My default"

b: field 100 "My text default" ; piškot, délka, hodnota aspektu text

**do** [

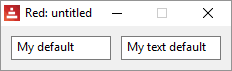
**print** a/text

**print** a/data ; přiřazeno příslovcem "default"

**print** b/text

**print** b/data ; error: hodnota aspektu data nebyla určena

] ]



My default  
My default  
My text default  
\*\*\* Script Error: My has no value  
\*\*\* Where: print  
\*\*\* Stack: view layout do-safe

VID DLS**with**

Řekněme, že chcete vytvořit piškot s hodnotami aspektů, které se vyhodnocují při realizaci piškotu. Vyhodnocení nemůžete uskutečnit coby argumenty piškotu, tudíž je zadáte pomocí slova with .

Toto nechodí:

Red [needs: view]

a: 2

b: 3

view [

base a \* 30x40 b \* 8.20.33

]

Toto chodí:

Red [needs: **view**]

a: 2

b: 3

**view** [

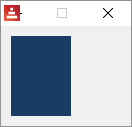
base with [

size: a \* 30x40

color: b \* 8.20.33

]

]



VID DLS**rate**

rate je aspekt s časovačem. Když časovač (timer) "tikne", generuje se událost  on-time. Pamatujte, že argumentem rate je "kolikrát za vteřinu", takže rate 20 je rychlejší než rate 5.

Tento kód způsobí blikání textu:

Red [needs: **view**]

**view** [

t: text "" rate 2

on-time [**either** t/text = "" [t/text: "Blink"] [t/text: ""]]

]

Tento kód vytvoří jednoduchou animaci, kde modrý piškot base cestuje napříč oknem:

Red [Needs: '**View**]

**view**[

size 150x150

b: base 40x40 blue "I move" rate 20

on-time [b/offset: b/offset + 1x1]

]

**Pomalejší frekvence:**

Pro časové úseky delší než 1 vteřinu, použijte pro rate argument typu time!:

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 je klíčové slovo (funkce), které spojuje chování jednoho piškotu s daty jiného piškotu.

Klasický příklad:

Red [Needs: **view**]

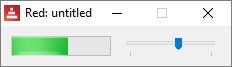
**view**[

progress 100x20 20% react [face/data: s/data]

s: slider 100x20 20%

]

Piškot progress reaguje na pohyb posuvníku piškotu slider:



**Upřesnění** funkce **react**

**/link** => Spojí objekty reaktivním vztahem.

**/unlink** => Zruší existující reaktivní vztah.

**/later** => Spustí reakci při příští změně.

**/with** => Určí vybraný pišktot (interní použití).

function!**layout**

Funkce layout se používá k vytvoření grafických útvarů bez jejich bezprostředního použití. Funkce definuje blok s popisem útvaru a s aktérem 'unview' nebo 'view'. Je aktivována pozdějším odkazem na její jméno.

Následující kód například zobrazí jedno okno a po jeho zavření zobrazí druhé:

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: 200x100]

**Zobrazení přes celou obrazovku:**

Následující skript vytvoří zobrazení přes celou obrazovku :

Red [needs: **view**]

**view** [size system/**view**/screens/1/size]

*Created with the Standard Edition of HelpNDoc:* [*Produce Kindle eBooks easily*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Draw**

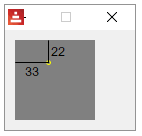
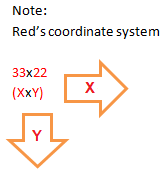
Obšírnější informaci o DSL Draw lze nalézt v [dokumentaci Redu](https://doc.red-lang.org/cs/draw.html), odkud jsou popisy příkazů převzaty. Součástí DSL Draw je sub-dialekt [Shape.](#_topic_Sub_dialektShape)

Funkce **draw** se používá ke kreslení rovinných útvarů. Tyto útvary jsou vyjádřeny jako seznamy (block!) direktiv pro funkci draw.

Celkem 30 příkazů dialektu draw lze rozdělit do čtyř skupin:

* **Kreslící instrukce**: line, triangle, box, polygon, circle, ellipse, arc, curve, spline, image, text, font
* **Vlastnosti čáry**: line-width, line-join, line-cap, anti-alias, pen, fill-pen,
* **Manipulace s objekty**: push, rotate, scale, translate, skew, transform, clip
* **Matice**: matrix, matrix-order, reset-matrix, invert-matrix - [viz](https://doc.red-lang.org/cs/draw.html#_matrix)

**Souřadnicový systém**



**Kreslící instrukce**

DRAW **line**

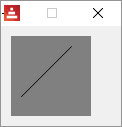
Nakreslí úsečku mezi dvěma body. Je-li zadáno více bodů, kreslí se další úsečky, napojované na předchozí.

Red [needs: **view**]

**view** [

base draw [line 60x10 10x60]

]

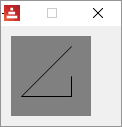


Red [needs: **view**]

**view** [

base draw [line 60x10 10x60 60x60 60x40]

]



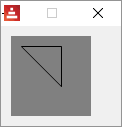
DRAW **triangle**

Red [needs: **view**]

**view** [

base draw [triangle 10x10 50x50 50x10]

]



DRAW **box**

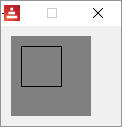
Red [needs: **view**]

**view** [

base draw [box 10x10 50x50]

; top-left bottom-right

]



**se zaobleným rohem:**

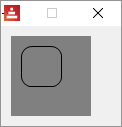
Red [needs: **view**]

**view** [

base draw [box 10x10 50x50 10]

; top-left bottom-right corner-radius

]



DRAW **polygon**

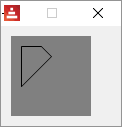
Red [needs: **view**]

**view** [

base draw [polygon 10x10 30x10 40x20 30x30 10x50]

; polygon se uzavře automaticky

]



DRAW **circle**

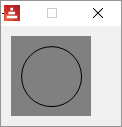
Red [needs: **view**]

**view** [

base draw [circle 40x40 30]

; center radius

]



**elipsa příkazem circle:**

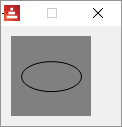
Red [needs: **view**]

**view** [

base draw [circle 40x40 30 15 ]

; center radius-x radius-y

]



DRAW **ellipse**

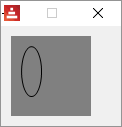
Elipsa se kreslí do pomyslného obdélníku. Zadávají se jeho body *top-left* a *bottom-right*.

Red [needs: **view**]

**view** [

base draw [ellipse 10x10 20x50]

]



DRAW **arc**

Kreslí kruhový oblouk ze zadaného středu (pair!) a se zadanými poloměry (rovněž pair!). Oblouk je vymezen dvěma úhly ve stupních. První úhel se měří od nuly, druhý úhel od prvního. Lze použít klíčové slovo  closed pro oblouk, uzavřený spojnicemi konců oblouků s jeho středem.

Red [needs: **view**]

**view** [

base draw [arc 40x40 20x20 45 180]

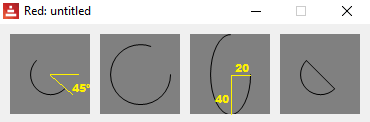
; center radius-x/radius-y start angle finish angle

base draw [arc 40x40 30x30 0 290]

base draw [arc 40x40 20x40 0 270]

base draw [arc 40x40 20x20 45 180 closed]

]



DRAW **curve**

Kreslí Bezierovu křivku ze 3 nebo 4 bodů:

* 3 body: 2 koncové, 1 kontrolní - kvadratická Bezierova křivka
* 4 body: 2 koncové, 2 kontrolní - kubická Bezierova. křivka

Red [needs: **view**]

**view** [

;nejprve si ukážeme samotné 4 body:

base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle 70x60 1]

;poté křivku ze 4 bodů: počáteční 1. kontrolní 2. kontrolní koncový:

base draw [curve 10x60 25x15 40x15 70x60]

;poté křivku ze 3 bodů: počáteční kontrolní koncový:

base draw [curve 10x60 25x15 70x60]

]



DRAW **spline**

Red [needs: **view**]

**view** [

;first we just show 4 points:

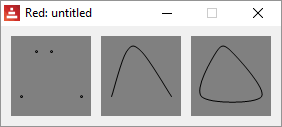
base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle 70x60 1]

;then the splines:

base draw [spline 10x60 25x15 40x15 70x60]

base draw [spline 10x60 25x15 40x15 70x60 closed]

]



DRAW **image**

Red [needs: **view**]

; image command expects a image! not a file!

; so you must first load the file

picture: **load** %smallballoon.jpeg

**view** [

base draw [image picture]

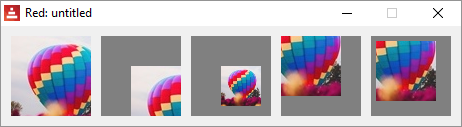
base draw [image picture 30x30]

base draw [image picture 30x30 70x70]

base draw [image picture crop 30x30 60x60]

base draw [image picture 5x5 crop 30x30 60x60]

]



There is also a color command (key color to be made transparent) and a border command, but I couldn't make those work yet.

;base draw [image picture 30x30 70x30 30x70 70x70]

;base draw [image picture 30x30 70x70 red]

;base draw [image picture 30x30 70x70 blue border]

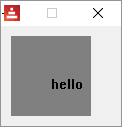
DRAW **text**

Red [needs: **view**]

**view** [

base draw [text 40x40 "hello"]

]



DRAW **font**

Vyběr fontu pro zobrazení textu

native!**Funkce compose**

Draw neprovádí vyhodnocení výrazů. Následující skript evokuje chybové hlášení:

Red [needs: view]

view [

base draw [line 60x10 (2 \* 10x30)]

]

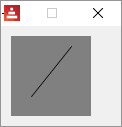
Vyhodnotit výraz nám pomůže funkce compose, nejlépe s upřesněním /deep.

Red [needs: **view**]

**view** **compose**/deep [

base draw [line 60x10 (2 \* 10x30)]

]



*Created with the Standard Edition of HelpNDoc:* [*Easily create PDF Help documents*](https://www.helpndoc.com/feature-tour)

# Vlastnosti čáry

Kromě geometrické konfigurace lze pro čáru (line) určit její tlouštku (line-width), způsob spojení (line-join), způsob jejího ukončení (line-cap) a její vyhlazení (anti-alias).

Její barevné a jiné vlastnosti se zadávají prostřednictvím nástroje pen a pen-fill - viz další [kapitola](#_topic_Barvagradientyavzory).

DRAW **line-width**

Nastaví novou tlouštku (width) čar.

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**

Nastaví způsob spojování čar při kreslení. Přípustné jsou hodnoty: miter (implicitní), round, bevel nebo miter-bevel.

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

] ]



miter-bevel usekne špičku tvaru miter na tvar bevel, přesáhne-li její [délka](https://msdn.microsoft.com/en-us/library/windows/desktop/ms534148(v=vs.85).aspx) stanovenou mez - obvykle desetinásobek tlouštky čar.

DRAW **line-cap**

Definuje tvar ukončení čáry. Lze použít flat (default), square nebo 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 **anti-alias**

Vyhlazení zubatych obrysů (anti-alias) dává pohlednější vykreslení ale degraduje výkon. Lze jej nastavit na on (default) nebo 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** ; toto je počáteční nastavení

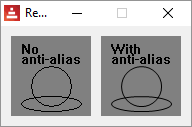
text 10x5 "With"

text 10x15 "anti-alias"

circle 40x50 20

ellipse 10x60 60x15

] ]



DRAW **pen** < color, (linear, radial, diamond) gradient, pattern, bitmap **>**

Příkaz určuje barevné provedení čáry.

DRAW **fill-pen** < color, (linear, radial, diamond) gradient, pattern, bitmap **>**

Příkaz definuje výplňový režim pro uzavřené tvary.

*Created with the Standard Edition of HelpNDoc:* [*Full-featured Kindle eBooks generator*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

# Barva, gradienty a vzory

Barevné a jiné vlastnosti čáry se zadávají prostřednictvím příkazů pen a pen-fill.

DRAW **pen** < color, (linear, radial, diamond) gradient, pattern, bitmap **>**

Příkaz určuje barevné provedení čáry.

DRAW **fill-pen** < color, (linear, radial, diamond) gradient, pattern, bitmap **>**

Příkaz definuje výplňový režim pro uzavřené tvary.

DRAW **pen <**color>

Red [needs: **view**]

**view** [

base draw [

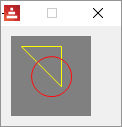
pen yellow ; color as word!

triangle 10x10 50x50 50x10

pen 255.10.10 ; color as tuple!

circle 40x40 20

] ]



DRAW **fill-pen <**color>

Red [needs: **view**]

**view** [

base draw [

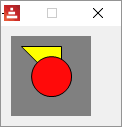
fill-pen yellow ; color as word!

triangle 10x10 50x50 50x10

fill-pen 255.10.10 ; color as tuple!

circle 40x40 20

] ]



Vypnutí režimu pen a fill-pen:

Red [needs: **view**]

**view** [

base draw [

pen **off**

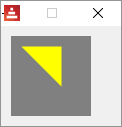
fill-pen yellow ; color as word!

triangle 10x10 50x50 50x10

fill-pen **off**

circle 40x40 20

] ]



DRAW **linear** - lineární barevný gradient

Převzato z [dokumentace Redu](-lang.org/cs/draw.html#_linear_gradient_pen)

<pen/fill-pen> linear <color1><offset> ... <colorN><offset><start><end><spread>   
  
<color1/N> : seznam barev gradientu (tuple! word!).   
<offset> : (optional) odsazení barvy gradientu (float!).   
<start> : (optional) počáteční bod (pair!).   
<end> : (optional unless <start>) koncový bod (pair!).   
<spread> : (optional) způsob šíření (word!).

Nastaví barevný gradient pro kreslení. Pro metodu spread jsou akceptované následující hodnoty:  pad, repeat, reflect (aktuálně je pad totéž jako repeat pro Windows).

Je-li použita metoda spread, definují koncové body směr gradientu. Nejsou-li body zadány, rozvíjí se gradient v horizontálním směru uvnitř kresleného obrazce.

**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** ; totéž jako repeat

line-width 5

line 0x0 80x80

]

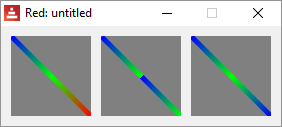
base draw [

pen linear blue green 0x0 40x40 **reflect** ; zrcadlit

line-width 5

line 0x0 80x80

] ]



**Fill-pen**

Red [needs: **view**]

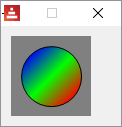
**view** [

base draw [

fill-pen linear blue green red 18x18 62x62

circle 40x40 30

] ]



DRAW **radial** - radiální barevný gradient

Převzato z [dokumentace Redu](-lang.org/cs/draw.html#_radial_gradient_pen)

<pen/fill-pen> radial <color1> <offset> ... <colorN> <offset> <center> <radius> <focal> <spread>   
  
<color1/N> : seznam barev gradient (tuple! word!).   
<offset> : (optional) odsazení barvy gradientu (float!).   
<center> : (optional) střed kružnice gradientu (pair!).   
<radius> : (optional unless <center>) poloměr kružnice (integer! float!).   
<focal> : (optional) ohnisko (pair!).   
<spread> : (optional) způsob šířdní (word!).

Nastaví radiální gradient pro kreslící operace. Pro metodu "spread" jsou akceptovány následující hodnoty: pad, repeat, reflect (aktuálně je pad totéž jako repeat ve Windows).

Radiální gradient bude vykreslován od ohniska k okraji kružnice, definované středem a poloměrem. Počáteční barva se vykreslí v ohnisku, koncová na okraji kruhu.

**Pen**

Red [needs: **view**]

**view** [

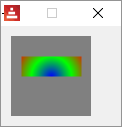
base draw [

pen radial blue green red 40x40 40 ; colors center radius

line-width 20

line 10x30 70x30

] ]



**Fill-pen**

Red [needs: **view**]

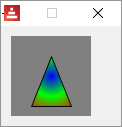
**view** [

base draw [

fill-pen radial blue green red 40x40 40 ; colors center radius

triangle 20x70 60x70 40x20

] ]



DRAW **diamond** - kárový barevný gradient

Převzato z [dokumentace Redu](-lang.org/cs/draw.html#_radial_gradient_pen)

<pen/fill-pen> diamond <color1> <offset> ... <colorN> <offset> <upper> <lower> <focal> <spread>   
  
<color1/N> : seznam barev gradientu (tuple! word!).   
<offset> : (optional) odsazení barvy gradientu (float!).   
<upper> : (optional) horní roh čtyřúhelníku. (pair!).   
<lower> : (optional unless <upper>) dolní roh čtyčúhelníku (pair!).   
<focal> : (optional) ohnisko (pair!).   
<spread> : (optional) způsob šíření (word!).

Nastaví gradient ve tvaru kár. Pro metodu "spread" jsou akceptovány následující hodnoty: pad, repeat, reflect (aktuálně je pad totéž jako repeat ve Windows).

Kárový gradient se vykreslí ve směru od ohniska k okraji kára, definovaného horním a dolním rohem. Počáteční barva se vykreslí v ohnisku, koncová barva na okraji kára.

Red [needs: **view**]

**view** [

base draw [

fill-pen diamond blue green red ; gradient s ohniskem ve středu

circle 40x40 35

]

base draw [

fill-pen diamond blue green red 10x10 50x50 ;souřadnice gradientu "box"

circle 40x40 35

]

base draw [

fill-pen diamond blue green red 10x10 50x50 30x48 ; přidané ohnisko

circle 40x40 35

]

base draw [

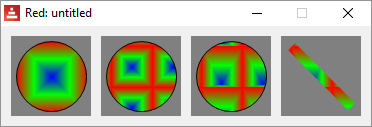
pen diamond blue green red 10x10 50x50 30x48

; a line over the last gradient:

line-width 10

line 10x10 70x70

] ]



DRAW **pattern** - výplň vzorkem

Převzato z [dokumentace Redu](-lang.org/cs/draw.html#_radial_gradient_pen)

<pen/fill-pen> pattern <size> <start> <end> <mode> [<commands>]   
  
<size> : velikost vnitřního zobrazení (pair!)   
<start> : (optional) horní roh vnitřního zobrazení (pair!).   
<end> : (optional) dolní roh vnitřního zobrazení (pair!).   
<mode> : (optional) uspořádání (word!).   
<commands> : blok příkazů Draw pro určení vzorku.

Nastaví tvar uživatele jako vzorek pro plnící operace. Pro režim "mode" jsou akceptovány tyto hodnoty:  tile (default), flip-x, flip-y, flip-xy, clamp.

Výchozím bodem je 0x0 a koncovým bodem je <size> (velikost).

Red [needs: **view**]

**view** [

; nejprve kreslíme vyplněný obdélník:

base draw [

fill-pen pattern 10x10 [

circle 5x5 4

line 3x3 7x7

] box 0x0 79x79

]

; potom vyplněnou diagonálu:

base draw [

pen pattern 10x10 [

circle 5x5 4

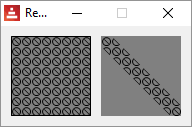
line 3x3 7x7

]

line-width 15

line 0x0 79x79

] ]



DRAW **bitmap** - výplň obrázkem

Převzato z [dokumentace Redu](-lang.org/cs/draw.html#_radial_gradient_pen)

<pen/fill-pen> bitmap <image> <start> <end> <mode>   
  
<image> : obrázek pro dlaždice (image!).   
<start> : (optional) horní roh vyplňovaného obdélníku (pair!).   
<end> : (optional) dolní roh vyplňovaného obdélníku (pair!).   
<mode> : (optional) uspořádání (word!).

Nastaví obrázek jako vzorek pro výplňové operace. V režimu mode jsou akceptovány následující hodnoty: tile (default), flip-x, flip-y, flip-xy, clamp (připnout).

Výchozím bodem vyplňovaného obdélníku (box) je 0x0 a koncovým bodem je velikost obrázku - oboje typu pair!.

Výplňovým obrázkem pro následující příklad je:  formátu bmp, png (size 35x35).

Red [needs: **view**]

myimage: **load** %asprite.bmp ; musí být v dosahu funkce load

**view** [

base draw [

fill-pen bitmap myimage tile ; default

box 0x0 79x79

]

base draw [

fill-pen bitmap myimage flip-x ; zrcadlově měnit podél osy x

box 0x0 79x79

]

base draw [

fill-pen bitmap myimage flip-y ; zrcadlově měnit podél osy y

box 0x0 79x79

]

base draw [

fill-pen bitmap myimage flip-xy ; zrcadlově měnit podél os **x** i **y**

box 0x0 79x79

]

base draw [

fill-pen bitmap myimage clamp ; připnout jako na nástěnku

box 0x0 79x79

]

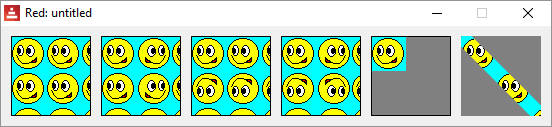
base draw [

pen bitmap myimage ; vložit do diagonály

line-width 15

line 0x0 80x80

] ]



*Created with the Standard Edition of HelpNDoc:* [*Easily create HTML Help documents*](https://www.helpndoc.com/feature-tour)

# Transformace v rovině

Popisy příkazů push, rotate, scale, translate, skew a transform jsou převzaty z  [dokumentace Redu.](https://doc.red-lang.org/cs/draw.html#_push)

DRAW **push** nutno doplnit

DRAW **rotate**

rotate <angle> <center> [<commands>]   
rotate 'pen <angle> rotate 'fill-pen <angle>   
  
<angle> : úhel ve stupních (integer! float!).   
<center> : (optional) střed otáčení (pair!).   
<commands> : (optional) příkazy dialektu Draw.

Nastaví pravotočivou rotaci ve stupních kolem daného bodu. Není-li zadán nepovinný parametr center, provede se rotace kolem počátku aktuálního souřadného systému. Negativní hodnoty lze použít pro levotočivou (counter-clockwise) rotaci. Je-li jako poslední argument zadán blok, aplikuje se rotace pouze na příkazy v bloku.

Při použití literálových slov (lit-words) 'pen nebo 'fill-pen  se rotace aplikuje na aktuální pen nebo fill-pen.

Red [needs: **view**]

**view** [

base draw [

pen red

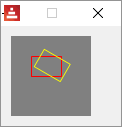
box 20x20 50x40 ; obdélník neotáčený

rotate 30 40x40 ; úhel otočení, střed otáčení

pen yellow

box 20x20 50x40 ; obdélník otáčený

] ]



DRAW **scale**

scale <scale-x> <scale-y> [<commands>]   
scale 'pen <scale-x> <scale-y>   
scale 'fill-pen <scale-x> <scale-y>   
  
<scale-x> : měřítko pro osu X (number!).   
<scale-y> : měřítko pro osu Y (number!).   
<commands> : (optional) příkazy dialektu Draw.

Nastaví velikost zvětšení. Zadané hodnoty jsou násobitelé; hodnoty větší než jedna zvětšují, hodnoty menší než jedna zmenšují. Je-li jako poslední argument zadán blok, aplikuje se změna velikosti pouze na příkazy v bloku.

Při použití literálových slov (lit-words) 'pen nebo 'fill-pen  se změna velikosti aplikuje na aktuální pen nebo fill-pen.

Red [needs: **view**]

**view** [

base draw [

pen red

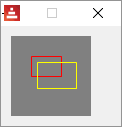
box 20x20 50x40 ; obdélník nezvětšovaný

scale 1.3 1.3 ; zvětšení o30% v obou směrech

pen yellow

box 20x20 50x40 ; obdélník zvětšovaný

] ]



DRAW **translate**

translate <offset> [<commands>]   
translate 'pen <offset>   
translate 'fill-pen <offset>   
  
<offset> : velikost posunu (pair!).

Nastaví počátek pro následné kreslicí příkazy. Víceré příkazy translate mají kumulativní účinek. Je-li blok zadán jako poslední argument, použijí se posuny pouze u příkazů z tohoto bloku.

Při použití literálových slov (lit-words) 'pen nebo 'fill-pen se posunutí aplikuje na aktuální pen nebo fill-pen.

Red [needs: **view**]

**view** [

base draw [

pen red

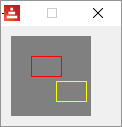
box 20x20 50x40 ; neposouvaný obdélník

translate 25x25

pen yellow

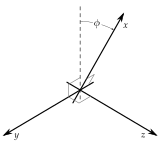
box 20x20 50x40 ; posouvaný obdélník

] ]



DRAW **skew - zkosení**

Zkosený souřadnicový systém je tehdy, když jeho osy nejsou ortogonální.



Příkaz skew natočí osu x nebo osu y o zadaný počet stupňů.

skew <skew-x> <skew-y> [<commands>]   
skew 'pen <skew-x> <skew-y>   
skew 'fill-pen <skew-x> <skew-y>   
  
<skew-x> : zkosení ve stupních ve směru x (integer! float!).   
<skew-y> : (optional) totéž ve směru y (integer! float!)   
<commands> : (optional) příkazy dialektu Draw

Nastaví zkosení souřadných os, zadané úhlem pootočení ve stupních. Není-li zadáno <skew-y>, předpokládá se, že je nulové. Je-li blok zadán jako poslední argument, použije se zkosení pouze u příkazů z tohoto bloku.

Při použití literálových slov (lit-words) 'pen nebo 'fill-pen se zkosení aplikuje na aktuální pen nebo fill-pen

Red [needs: **view**]

**view** [

base draw [

pen red

box 20x20 50x40 ; nezkosený obdélník

skew 30 ; zkosení 30° ve směru osy x

pen yellow

box 20x20 50x40 ; zkosený obdélník

]

base draw [

pen red

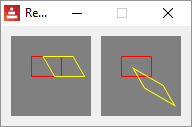
box 20x20 50x40 ; nezkosený obdélník

skew 30 30 ; zkosení 30° ve směru obou os

pen yellow

box 20x20 50x40 ; zkosený obdélník

] ]



DRAW **transform**

transform <center> <angle> <scale-x> <scale-y> <translation> [<commands>]   
transform 'pen <center> <angle> <scale-x> <scale-y> <translation>   
transform 'fill-pen <center> <angle> <scale-x> <scale-y> <translation>   
  
<center> : (optional) střed rotace (pair!).   
<angle> : úhel rotace ve stupních (integer! float!).   
<scale-x> : měřítko pro osu X (number!).   
<scale-y> : měřítko pro osu Y (number!).   
<translation> : velikosti posunu (pair!).   
<commands> : (optional) příkazy dialektu Draw.

Zadá transformaci jako je pootočení, změna velikosti (scaling) a posun jedním příkazem. Je-li blok zadán jako poslední argument, použije se transformace pouze u příkazů z tohoto bloku.

Při použití literálových slov (lit-words) 'pen nebo 'fill-pen se transformace aplikuje na aktuální pen nebo fill-pen.

Red [needs: **view**]

**view** [base draw [

pen red

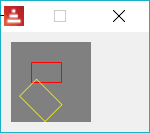
box 20x20 50x40 ; netransformovaný obdélník

transform 25x25 45 1.2 1.2 0x20 ;střed rotace, úhel, měřítka, posuny

pen yellow

box 20x20 50x40 ; transformovaný obdélník

] ]



DRAW **clip**

Určuje pravoúhlý výřez definovaný dvěma body (start, end) nebo libovolně tvarovaný výřez, definovaný blokem příkazů sub-dialektu Shape. Takto určený výřez je platný pro všechny následné příkazy dialektu Draw. Je-li blok zadán jako poslední argument, použije se výřez pouze u příkazů z tohoto boku.

clip <start> <end> <mode> [<commands>]   
clip [<shape>] <mode> [<commands>]   
  
<start> : horní levý roh výřezu (pair!)   
<end> : spodní pravý roh výřezu (pair!)   
<mode> : (optional) způsob sloučení výřezů (word!)   
<commands> : (optional) příkazy dialektu Draw   
<shape> : příkazy dialektu Shape

Red [needs: **view**]

**view** [base draw [

clip 10x40 60x70

pen black

fill-pen red circle 15x40 30

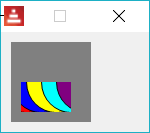
fill-pen blue circle 30x40 30

fill-pen yellow circle 45x40 30

fill-pen cyan circle 60x40 30

fill-pen purple circle 75x40 30

]



Spojení nového výřezu se stávajícím může být zadán jako jeden z následujících způsobů:

* + - replace (default)
    - intersect
    - union
    - xor
    - exclude

*Created with the Standard Edition of HelpNDoc:* [*Full-featured EBook editor*](https://www.helpndoc.com/create-epub-ebooks)

# Sub-dialekt Shape

Sub-dialekt Shape umožňuje vytvářet tvary (kresby) jako bloky. Použití dialektu shape má tuto skladbu:

**shape**  ( příkazy dialektu Shape )

Specifické rysy tohoto dialektu jsou:

* každý kreslící příkaz začíná na aktuální pozici pera
* pozici pera lze měnit bez kreslení zadáním cílové polohy příkazem **move**
* tvary jsou automaticky uzavírány - spojením posledního bodu s prvním
* generované tvary lze zadat do příkazu **fill-pen** pro vytváření individuálních výplní
* kreslící příkazy používají implicitně absolutní (relativní k piškotu) souřadnice; relativní (relativní k poslední pozici) souřadnice jsou aktivní ve verzi 'lit-word příkazu.

V argumentu funkce **shape** můžete použít tyto příkazy a vlastnosti:

move, arc, curve, curv, qcurve, qcurv, hline, vline, line, line-width, line-join, line-cap, pen, fill-pen.

Modře označené entity se deklarují stejně jako v dialektu Draw.

**move**

Přemístí pero na novou pozici, nic se nekreslí:

Red [needs: **view**]

myshape: [

line 10x10 70x70 ; line from 10x10 to 70x70

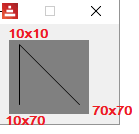
**move** 10x70 ; přemístí pero na pozici 10x70

line 10x10 ; kreslí čáru z aktuální (10x70) pozice

]

**view** **compose**/deep/only [base draw [shape (myshape)

] ]



**Relativní pozice**

Souřadnice se stanou relativní, přidá-li se apostrof před příkaz:

Red [needs: **view**]

myshape: [

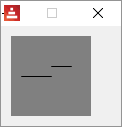
line 10x40 40x40 ; horizontální čára do středu

'**move** 0x-10 ; posun aktuální pozice vzhůru

'line 20x0 ; konec čáry relativně k aktuální pozici

]

**view** **compose**/deep/only [base draw [shape (myshape)]]



**line**

Nakreslí úsečku mezi dvěma body. Je-li zadáno více bodů, kreslí se další úsečky, napojované na předchozí.

Red [needs: **view**]

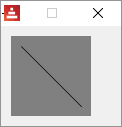
myshape: [line 10x10 70x70]

**view** **compose**/deep/only [

base draw [

shape (myshape)

] ]



Všimněte si upřesnění compose/deep/only a závorek kolem myshape**.** Tyto náležitosti jsou zřejmě při práci s DSL Shape povinné.

Upřesnění /deep přikazuje vyhodnotit zanořenou závorku.

Upřesnění /only přikazuje ... nutno doplnit.

**Automatické uzavírání tvaru**

V následujícím příkladu jsou explicitně zadány pouze dvě přímky. Příkaz fill-pen byl přidán jenom pro zvýraznění uzavřené plochy:

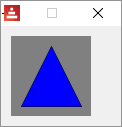
Red [needs: **view**]

myshape: [

line 10x70 40x10 70x70

]

**view** **compose**/deep/only [base draw [ fill-pen blue shape (myshape)]]



**hline** a **vline**

Kreslí horizontální nebo vertikální přímku z aktuální pozice pera.

Red [needs: **view**]

myshape: [

**move** 10x10 ; nastaví pero na 10x10

hline 30 ; horizontále X = 30 (délka 20)

vline 30 ; vertikála Y = 30 (výška 20)

'hline 30 ; horizontála délky 30

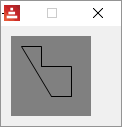
'vline 30 ; vertikála délky 30

'hline -20 ; horizontála záporné délky 20

; lomená čára se posléze uzavře

]

**view** **compose**/deep/only [base draw [shape (myshape)]]

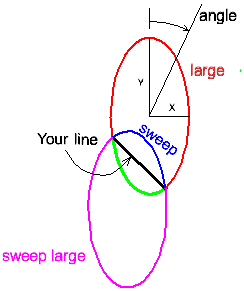


**arc** (liší se od Draw-arc)

arc evokuje absolutní souřadnice   
'arc evokuje relativní souřadnice

Kreslí oblouk mezi aktuální polohou pera a koncovým bodem oblouku. Oblouk je definován úsekem fiktivní elipsy s poloměry r-x, r-y a úhlem odklonu vodorovné osy elipsy od souřadnicové osy x.

**Syntaxe: arc**  <end> <r-x> <r-y> <angle> (plus nepovinné: sweep, large)

 large : kreslí červený oblouk   
 sweep : kreslí modrý oblouk  
 sweep large : kreslí fialový oblouk

Bez doplňku sweep, large kreslí zelený oblouk. Oklon elipsy v obrázku je 0 stupňů.

Všimněte si, že zadáváte pouze koncovou pozici oblouku. Počáteční pozice je daná aktuální polohou pera. Je-li tedy **arc**prvním příkazem argumentu pro **shape,** musíte se nejprve přesunout (move) do počáteční pozice.

Red [needs: **view**]

myshape\_1: [

**move** 10x40

arc 70x40 10 5 0 ]

myshape\_2: [

**move** 10x40

arc 70x40 30 10 0 sweep ]

myshape\_3: [

**move** 30x30

arc 60x60 10 10 315 ] ; u kružnice je hodnota úhlu irelevantní

myshape\_4: [

**move** 10x40

arc 70x40 10 5 45 ]

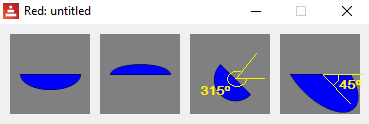
**view** **compose**/deep/only [

base draw [ fill-pen blue shape (myshape\_1)]

base draw [ fill-pen blue shape (myshape\_2)]

base draw [ fill-pen blue shape (myshape\_3)]

base draw [ fill-pen blue shape (myshape\_4)] ]



Odlišné parametry radius-x a radius-x vytvoří eliptický oblouk.

**curve**

Kreslí kubickou Bezierovu křivku, danou třemi až čtyřmi body. Připouští absolutní nebo relativní ('curve) souřadnice.

Red [needs: **view**]

myshape\_1: [

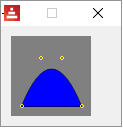
**move** 10x70

curve 30x20 50x20 70x70 ; 2 kontrolní a koncový bod

]

**view** **compose**/deep/only [base draw [ fill-pen blue shape (myshape\_1)

] ]



**curv , qcurve** a **qcurv**

Všechny tři příkazy generují Bezierovy křivky s absolutními či relativními (') souřadnicemi.

* **qcurv**  kreslí hladkou kvadratickou Bezierovu křivku z 1+1 bodů.
* **qcurve**  kreslí kvadratickou Bezierovu křivku z alespoň 1+2 bodů;
* **curv** kreslí hladkou kubickou Bezierovu křivku z alespoň 1+1 bodů;

Tyto křivky používají počáteční, koncový a kontrolní bod či body. Pěkná animace je na Wikipedii, viz [https://en.wikipedia.org./wiki/Bézier\_curve](https://en.wikipedia.org/wiki/Bézier_curve)

Red [needs: **view**]

myshape\_1: [

**move** 10x40 ;starting point

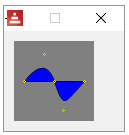
curv 30x10 40x40 50x60 70x40

; control point control point control point endpoint

]

**view** **compose**/deep/only [base draw [ fill-pen blue shape (myshape\_1)

] ]



*Created with the Standard Edition of HelpNDoc:* [*Easily create Help documents*](https://www.helpndoc.com/feature-tour)

**Parse**

Parse je vnořený jazyk (DSL - Domain Specific Language) pro parsování vstupních řad (series) s použitím gramatických pravidel tohoto jazyka. Používá se pro ověření shody, vyjímání a úpravu vstupních dat. Ve všech případech se používá funkce **parse** v této skladbě:

**parse** < input > < rules >

<input> : jakákoli hodnota řady (any-string!, any-block!, binary!)  
<rules> : blok s pravidly

Parsování začíná na začátku vstupní řady. Při úspěšné konfrontaci s pravidly (success) postupuje k dalšímu členu řady. Parsování končí výskytem neúspěchu a operace vrací FALSE nebo úspěšným průchodem celou vstupní řadou a operace vrací TRUE.

Významnou úlohu při sestavování gramatických pravidel mají **klíčová slova**, která lze rozdělit do několika kategorií:

**Ověření shody**:

ahead, end, none, not, quote, skip, thru to

**Průběh parsování:**

break, if, into, fail, then, reject

**Iterace:**

some, any, opt, while

**Extrakce a uložení:**

set, copy, keep, collect, collect-set, collect-into

**Modifikace:**

insert, remove, change

native! **parse**

Funkce parse podrobí každý element vstupu porovnání s odpovídajícím blokem pravidel. Podmínkou přechodu k ověřování dalšího vstupního elementu je úspěšné hodnocení předchozího. Funkce vrací true, konvenují-li všechny elementy vstupní řady se zadanými pravidly, případně vrací false, pokud tomu tak není.

Nejjednodušším příkladem je prostá kontrola, zda každý element vstupního bloku je roven svému protějšku v bloku pravidel:

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"] ; vstupní blok

**print** **parse** a ["fox" "dog" "owl" "rat" "elk" "cat"] ; kontrolní blok

true

Pro větší přehlednost popisu parse, přepišme výše uvedený příklad do jiného formátu. Kontrolní blok se formálně skládá ze šesti pravidel:

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"] ; input block

**print** **parse** a [ ; here the rules begin:

"fox" ; rule 1 matches element 1 => success

"dog" ; rule 2 matches element 2 => success

"owl" ; rule 3 matches element 3 => success

"rat" ; rule 4 matches element 4 => success

"elk" ; rule 5 matches element 5 => success

"cat" ; rule 6 matches element 6 => success

]

; protože všechny konfrontace byly úspěšné, a došli jsme na konec řady

; je výsledkem "true"

true

**Vstupní blok lze porovnávat i s datovými typy:**

Red[]

a: [33 18.2 #"c" "rat"] ; input block

**print** **parse** a [

**integer!**

**float!**

**char!**

**string!**

]

true

# Do kontrolního bloku (pravidel) lze vložit běžný kód v závorce:

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

**V pravidlech lze použít operátor logického "OR"**  ( **"|"** ) **:**

Red[]

a: ["fox" "rat" "elk"]

b: ["fox" "owl" "elk"]

**print** **parse** a [

"fox"

["rat" | "owl"] ; alternativně použitelné hodnoty

"elk"

]

**print** **parse** b [

"fox"

["rat" | "owl"| "cat" | "whatever"]

"elk"

]

true

true

**Obsah vstupního bloku lze upravit zadaným vypuštěním elementů:**

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"]

**print** **parse** a [

4 **skip** ; přikázaný skok příkazem "4 skip"

"elk"

"cat"

]

true

Red[]

a: ["rat" "rat" "rat" "rat" "elk" "cat"]

**print** **parse** a [

4 "rat" ; počet stejných prvků

"elk"

"cat"

]

true

Red[]

a: ["rat" "rat" "elk" "cat"]

**print** **parse** a [

0 4 "rat" ; počet stejných prvků zadán přípustným rozsahem

"elk"

"cat"

]

true

**Introspekce prováděného průběhu parse:**

**parse-trace** < input > < rules >

**Upřesnění funkce parse:**

**/case** =>

**/part** =>

**/trace** =>

*Created with the Standard Edition of HelpNDoc:* [*Easily create Help documents*](https://www.helpndoc.com/feature-tour)

# Matching

# Klíčová slova: skip, thru, to, ahead, end, none, not, quote

PARSE **skip**

Bez argumentu přeskočí pouze jeden element:

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"] ; input block

**print** **parse** a [ ; here the rules begin:

"fox" ; rule 1 matches element 1 => true

**skip** ; rule 2 skips this element

"owl" ; rule 3 matches element 3 => true

"rat" ; rule 4 matches element 4 => true

"elk" ; rule 5 matches element 5 => true

"cat" ; rule 6 matches element 6 => true

]

true

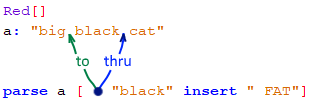
S argumentem přeskočí zadaný počet elementů:

**print** **parse** "axxb" ["a" 2 skip "b"]

true

PARSE **to** a **thru**

Přeskočí elementy až k označenému místu vstupního bloku. Slovo **to** nastaví počátek parsování před zadaný element, slovo thru nastaví počátek parsování za zadaný element.



PARSE  **to**

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

**print** **parse** a [

**to** "elk" ; přeskočí všechny elementy před "elk"

"elk" "cat" "bat" ; zbytek je konformní s pravidlem

]

true

PARSE **thru**

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

**print** **parse** a [

**thru** "elk" ; přeskočí všechny elementy včetně "elk"

"cat" "bat" ; zbytek je konformí s pravidlem

]

true

Klíčové slovo thru je prospěšné, i když procedura skončí hodnocením false:

Red[]

s: "dnes je nádherný den"

**parse** s [**thru** "nádherný" change "" " a slunný"]

false

s

== "dnes je nádherný a slunný den"

K čemu zde došlo:

Vstupní řetězec byl konfrontován s příkazem ke skoku za slovo "nádherný", jehož následná mezera byla nahrazena doplňkem " a slunný".

PARSE **ahead**

Zkontroluje, zda se následující (ahead) element shoduje s pravidlem.

Red[]

a: ["fox" "dog" "owl" "rat"] ; input block

**print** **parse** a [

"fox"

"dog"

**ahead** "owl" ;checks if the next item matches the rule

"owl"

"rat" ]

true

Ve skutečnosti je to mnohem složitější - viz [Ungaretti](http://helpin.red/RepetitionandMatchingLoops.html) # while - ahead

PARSE **end**

Vrací true, jestliže byly pozitivně zkontrolovány všechny položky - což se vykoná i bez slova end.

Red[]

a: [33 18.2 #"c" "rat"] ; input block

**print** **parse** a [ **integer! float! char!**  **string!** end ]

true

# Poznámka: Slovo end označuje pozici na konci každé sekvence. Interpret si jej dosadí sám, pokud není uveden.

PARSE **none**

Vždycky vrací true. Je to pravidlo pro všechny případy

Red[]

a: ["fox" "dog" "owl" "rat"]

**print** **parse** a ["fox" "dog"

**none (print** "Servus"**)** ; příkaz **print** lze vložit i bez **none**!

"owl" "rat" ]

true

PARSE **not**

Neguje platnost (existenci) následného "pravidla":

Red[]

a: ["fox" "dog" "owl" "rat"]

**print** **parse** a ["fox" "dog" **not** "owl" "owl" "rat"] ; museli jsme jedno "owl" přidat

true

**print** **parse** a ["fox" "dog" **not** "owl" **skip** "rat"] ; museli jsme "owl" přeskočit

true

PARSE **quote** nebo apostrof **'**

Match next value literaly (for dialect escaping needs).

Red[]

**parse** [x] ['x]

true

**parse** [[x]] [ **quote** [x]]

true

*Created with the Standard Edition of HelpNDoc:* [*Easy EPub and documentation editor*](https://www.helpndoc.com)

# Iterace

# Klíčová slova: some, any, opt, while.

Aplikace pravidla (rule) může být nepovinná nebo různým způsobem opakovaná.

|  |  |
| --- | --- |
| **Klíčové slovo či hodnota** | **Popis** |
| 3 <rule> | opakovat pravidlo 3 krát |
| 1 3 <rule> | opakovat pravidlo 1 až 3 krát |
| 0 3 <rule> | opakovat pravidlo 0 až 3 krát |
| **some** | opakovat pravidlo jednou či vícekrát |
| **any** | opakovat pravidlo nula či vícekrát |
| **opt** | shoda s pravidlem žádná nebo jedna |

**Určitý počet opakování - příklady**

>> parse "fogfogfog" [3 "fog"] ; určeno přesně

== true

>> parse "fogfogfog" [0 5 "fog"] ; určeno rozsahem

== true

**Neurčitý počet opakování - příklady**

PARSE **some**, PARSE **any**, PARSE **opt**

Aplikuje pravidlo následného vnořeného bloku (jednou či vícekrát, nikdy nebo alespoň jednou, nikdy nebo jednou), dokud neselže nebo dokud lze ve vstupním bloku postupovat:

Red[]

a: [ "elk" "cat" "owl"]

**parse** a [ **some** ["elk"] ; **jednou** nebo vícekrát

"cat" "owl" ]

true

**parse** a [ **some** ["elk" "cat"] ; jednou nebo **vícekrát**

"owl" ]

true

**parse** a [ **some** ["fig"] ; nikdy

"elk" "cat" "owl" ]

false

**parse** a [ **any** ["elk"] ; nikdy nebo **alespoň jednou**

"cat" "owl" ]

true

**parse** a [ **any** ["elk" "cat"] ; nikdy nebo alespoň jednou

"owl" ]

true

**parse** a [ **any** ["fig"] ; nikdy nebo alespoň jednou

"elk" "cat" "owl" ]

true

**parse** a [ **any** ["fig"] ; nikdy nebo alespoň jednou

"elk" "owl" ]

false ; nikdy se vztahuje jen na subpravidlo **any**

**parse** a[ **opt** ["fig"]; **nikdy** nebo alespoň jednou

"elk" "cat" "owl"]

true

**parse** a [ **opt** ["elk" "cat"] ; nikdy nebo **alespoň jednou**

"owl" ]

true

**parse** a [ **opt** ["elk" "owl"] ; nikdy nebo alespoň jednou

"cat" ]

false \*

\* Pokud se vstup neshoduje s pravidlem **opt**, přeskočí parse toto pravidlo a kontroluje tentýž vstup následným pravidlem.

Jiný příklad pro **opt**:

hd: "horskádráha" ; řetězec s 11 elementy

**parse** hd [**opt** "horská" "dráha"] ; == true

**parse** hd [**opt** "horská" "práva"] ; == false

PARSE **while**

Podobně jako **any**, aplikuje pravidlo následného vnořeného bloku nikdy nebo alespoň jednou, dokud neselže; nestará se o případnou změnu vstupu:

a: ["elk" "cat"]

**print** **parse** a [ **any** ["fox" "dog"] "elk" "cat" ]

true

**print** **parse** a [ **while** ["fox" "dog"] "elk" "cat" ]

true

Ve skutečnosti je to mnohem složitější - viz [Ungaretti](http://helpin.red/RepetitionandMatchingLoops.html) # while.

*Created with the Standard Edition of HelpNDoc:* [*Easy EBook and documentation generator*](https://www.helpndoc.com)

# Extrakce

# Klíčová slova: set, copy, keep, collect, collect set, collect into.

Připomínka: Příkaz "print" provádí úpravu výstupu. Sběrný blok ["fox" "rat"] například, upraví na fox rat.

PARSE **set** , PARSE **copy**

V obou případech přiřadí ke **jménu** následnou první hodnotu uspokojené regule:

**parse** [7 9 3] [**set** **val** integer! string! integer!] ; --> false

**print** val ; --> 7

**parse** [7 9 3] [**copy** **val** integer! integer! integer!] ; --> true **print** val ; --> 7

**parse** [7 9 3] [integer! **set** **val** string! integer!] ; --> false

**print** val ; --> val has no value!

Ve spojení s klíčovým slovem **any** se výstupy liší. Příkaz **set** vrací pouze první platnou hodnotu, příkaz **copy** vrací všechny zbyvající platné hodnoty:

**parse** [7 9 3] [integer! **set** **val** **any** integer!] ; --> true

**print** val ; --> 9

**parse** [7 9 3] [integer! **copy** **val** **any** integer!] ; --> true **print** val ; --> 9 3

|  |  |
| --- | --- |
| PARSE **keep**  *rule*  Používá se s klíčovým slovem **collect**, **collect set** a **collect into**. Připojí do sběrného bloku kopii první následné hodnoty uspokojené regule. |  |

PARSE **collect**

Vytvoří sběrný blok, zaplněný příkazem **keep.** Příkaz parse zde nevrací *true* či *false.*

a: ["fox" "dog" "owl" "rat" "elk" "cat"] ; vstupní blok

**print parse** a [

**collect**[

**keep** "fox" ; success, patří do sběrného bloku

"dog" "owl" ; success, ale nepatří do sběrného bloku

**keep** "rat" ; success, patří do sběrného bloku

**keep** "cow" ; fail, proto nepatří do sběrného bloku

"cat" ] ] ; success, ale nepatří do sběrného bloku

fox rat

PARSE **collect set**

Na rozdíl od **collect** vrací true či false. Vytvořený sběrný blok přiřadí zadané proměnné (zde b).

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"] ; input block

**print** **parse** a [ ; bez "print" nevrátí "false"

**collect** **set** b [ ; vytvoří sběrný blok **b**

keep "fox" "dog" "owl" ; "fox" is OK, will be kept

keep "rat" ; "rat" is OK, will be kept

keep "cow" "cat" ] ; "cow" will be not kept

]

**print** b

false

fox rat

PARSE **collect into**

Na rozdíl od **collect** vrací true či false. Vytvořený sběrný blok přiřadí předem připravené proměnné (zde b).

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"]

b: "" ; must create block first

**print** **parse** a [

**collect** **into** b [

keep "fox" ; success, WILL be kept

"dog"

"owl"

keep "rat" ; success, WILL be kept

keep "cow" ; FAIL! will NOT be kept

"cat"

]

]

**print** b

false

foxrat

**Selektivní výběr syntaxí set-word**

Při parsování lze ke zbytku vstupu přiřadit proměnnou, byť je výsledek negativní:

Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"]

**print** **parse** a [ "fox" "dog"  b: ] ; přiřadí zbytek slovu **b**

**probe** b

false

["owl" "rat" "elk" "cat"]

*Created with the Standard Edition of HelpNDoc:* [*Full-featured EBook editor*](https://www.helpndoc.com/create-epub-ebooks)

# Úprava vstupu

# Klíčová slova: insert, remove, change.

PARSE **insert**

Vloží zadanou hodnotu do vstupního bloku.

a: ["fox" "dog" "owl" "rat"]

**parse** a ["fox" "dog" **insert** 33 "owl" "rat"]

true

**print** a

fox dog 33 owl rat

Jiný příklad:

b: "Tvé velké oči"

**parse** b [**thru** "velké" **insert** " hnědé"] ; mezera!

false

**print** b

Tvé velké hnědé oči

PARSE **remove**

Odebere shodující se označené elementy ze vstupu.

a: ["fox" "dog" "owl" "rat"]

**parse** a [ "fox" **remove** "dog" **remove** "owl" "rat"]

true

**print** a

fox rat

Jiný příklad:

a: "Tvé velké oči"

**parse** a [ **to** "velké" **remove** "velké"] ; == false

**print** a

Tvé oči

PARSE **change**

Mění označenou položku konformního vstupního bloku:

a: ["fox" "dog" "owl" "rat"]

**parse** a ["fox" "dog" **change** "owl" "COW" "owl" "rat"] ; == false

**print** a

fox dog COW rat

*Created with the Standard Edition of HelpNDoc:* [*Full-featured Documentation generator*](https://www.helpndoc.com)

**Control flow**

# Control flow:

# Klíčová slova: break, if, into, fail, then, reject.

PARSE **break** TODO

Break out of a matching loop, always returning success.

PARSE **if** (expr)

Vyhodnotí logický výraz v závorce; nemá-li tento hodnotu "true", zastaví parsování a vrací "false":

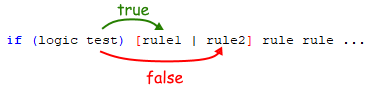
>> **parse [x] [if (1 = 1) skip]**

== true

>> **parse [x] [if (1 = 0) skip]**

== false

Vyhodnocení logického výrazu v závorce lze použít k výběru alternativnímu pravidla:



>> **parse [6 3 7] [integer! integer! if (1 = 1) [integer! | string!]]**

== true

>> **parse [6 3 7] [integer! integer! if (1 = 0) [integer! | string!]]**

== false

PARSE **into** TODO

Switch input to matched series (string or block) and parse it with rule..

PARSE **fail** TODO

Force current rule to fail and backtrack.

PARSE **then** TODO

Regardless of failure or success of what follows, skip the next alternate rule.

PARSE **reject** TODO

Break out of a matching loop (such as any, some, while) and indicate failure.

*Created with the Standard Edition of HelpNDoc:* [*Free Kindle producer*](https://www.helpndoc.com/feature-tour/create-ebooks-for-amazon-kindle)

**Introspekce**

**Introspekce**:

# Průběh funkce parse lze zobrazit pomocí funkce parse-trace.

PARSE **parse-trace**

Provede parsování vstupu a rovněž vytiskne průběh procesu krok za krokem.

a:  ["fox" "owl" "rat"]

print parse-trace a [[ "box" | "fox" ] "owl" "rat"]

-->

match: [ ["cow" | "fox"] "owl" "rat"]

input: ["fox" "owl" "rat"]

-->

match: ["cow" | "fox"]

input: ["fox" "owl" "rat"]

==> not matched

match: ["fox"]

input: ["fox" "owl" "rat"]

==> matched

<--

match: ["owl" "rat"]

input: ["owl" "rat"]

==> matched

match: ["rat"]

input: ["rat"]

==> matched

return: true

true

Průběh procesu parsování lze také zjistit vhodným vložením příkazů **print** :

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

*Created with the Standard Edition of HelpNDoc:* [*News and information about help authoring tools and software*](https://www.helpauthoringsoftware.com)