Helpin' Red

Licence: CC BY NC SA - You can copy, change and distribute. Copies and derivative work must have the same licence, not be commercial and give original work credit.

Table of contents

Homepage 4
Úvod 4
Začínáme 7
    Notepad++ 8
    Console 2 13
    "Hello world" 16
    Poznámky ke skladbě 22
    Introspekce a nápověda 24
    Vyhodnocení 26
    Zvýraznění kódu 29
Konzola - vstup a výstup 30
Exekuce kódu 32
Datové typy 34
    Další datové typy 41
    Hash!, vector! and map! 45
    Konverze datových typů 47
Bloky a řady 48
    Navigace řadami 50
    Series "getters" 54
    Series "changers" 62
Formátování dat 73
Matematika a logika 78
Konverze bází 88
Kryptografie 90
Kopírování 92
Opakování 93
Podmínky 96
Manipulace s textem 100
Práce s časem 106
Ošetření chyb 108
Soubory 109
    Psaní do souboru 113
    Čtení ze souboru 115
Funkce 117
Objekty 122
Reaktivní programování 124
Rozhraní OS 127
I/O 129
GUI 130
    Nastavení kontejneru 136
    Layout 140
    Faces 145
    Události a aktéři 163

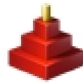
- Introspekce událostí ... 169
- Pokročilá témata ... 171

Draw ... 179

- Vlastnosti čáry ... 186
- Barva, gradienty a vzory ... 189
- Transformace v rovině ... 196
- Sub-dialekt Shape ... 202

Parse ... 207

- Matching ... 212
- Iterace ... 215
- Extrakce ... 218
- Úprava vstupu ... 220
- Control flow ... 221
- Introspekce ... 223



Helpin'Red

Pr vodce s p íklady [programovacím jazykem Red]

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

Do   eštiny p eložil Tovim

P edkládaný text je pokusem o srozumitelný výklad programovacího jazyka Red.   eský p eklad zatím nedokon ené jeho oficiální dokumentace lze   íst na https://doc.red-lang.org/cs .

Podn ty a p ipomínky k p ekladu jsou v ele vítány! M žete je zaslat na adresu tovim@seznam.cz

Tento text byl vytvo en aplikací HelpNDoc dne 25. 9. 2018 a lze jej   íst také ve formátu PDF a Word.

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

Úvod

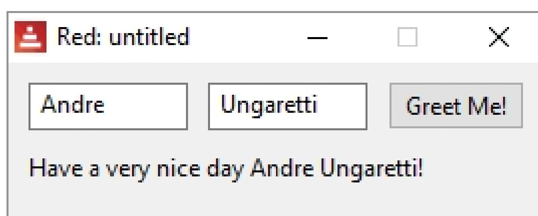
O programovacím jazyku Red

- 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 m žete sv j kód kompilovat a generovat samostatnou provád   ku.

- Red si p ed interpretací ást kódu kompiluje, takže je docela rychlý.
- Red je jednoduchý. Jeho kód je istý a stru ný.
- Red je aktuáln (duben 2018) ve fázi 'alfa' ale má ambiciózní cíle:
    - býti multi-platformní;
    - míti nativní systém GUI s UI dialektem;
    - býti multi-úrov ový (full-stack) programovací jazyk.
- Red je 'open-source' pokra ováním jazyka Rebol, vytvo eného tv rcem Amigy - Carlem Sassentarhem. Chcete-li si vyzkoušet n které procedury, které nejsou zatím v Redu zavedeny, m žete si je vyzkoušet v instalovaném Rebolu.
- Red je vyvíjen skupinou lidí pod vedením Nenada Rakocevice.
- V poslední dob posbíral Red nemalý fond z ICO a v Pa íži byla ustavena spole nost Red Foundation, takže je nad je 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, m žete nahlédnout do Short Red Examples od Nicka Antonaccia, p ípadn do eského p ekladu prozatimní oficiální dokumentace Programovací jazyk Red.

Introductory notes by Ungaretti:

This is an evolution of the Red Language Notebook.
I chose to use HelpNDoc 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.
- I try to make my work original, but some text was copied and pasted from Red's official documentation and I based some examples on what I found at:
    - red-by-example.org by Arie van Wingerden and Mike Parr
    - mycode4fun.com.uk by Alan Brack
    - redprogramming.com by Nick Antonaccio

Also, a lot of help came from the Red community at gitter.im/red/home

Nezbytná poznámka p ekladatele:

Mám zato, že programátor nebo zájemce o programování musí alespo áste n um t anglicky a mít znalost alespo nezbytné zásoby anglických slov.

Vycházeje z tohoto p edpokladu, pokusím se n které termíny nep ekládat, pokud budou rozumn použitelné v eské v t .

P ivítám pomoc p i p ekladu termínu **face** (tvá), pro který jsem použil výraz **piškot**, jenž n které Redisty (Rebolisty) pon kud irituje. Slovo face zde nemá s tvá i mnoho spole ného, nebo je to zkrácenina slova interface. Face i interface mi nep ijdou jako ozna ení pro daný objekt jednozna n p iléhavá a proto se jich nedržím. Alternativní ozna ení k face je widget, jež se mi líbí ale špatn se s ním v eské v t pracuje. Také se mi líbí slovo **grafion**. Tož uvidíme.

Použítí upraveného editoru **Notepad ++** v ele doporu uji - spolu s interaktivní konzolou Redu v aplikaci Console 2 pro jednodušší p íklady.

Created with the Standard Edition of HelpNDoc: Easily create EPub books

Za íáme

Instalace prost edí Red

Nejprve je zapot ebí stáhnout si do po íta e vhodnou instalaci Redu. Když ji (dvojklikem) spustíte, vytvo í a otev e ve vašem monitoru interaktivní konzolu Red Console.

Cestu ke složce s binárním souborem red~.exe lze uložit do systémové prom nné System PATH. To umožní otevírat interpreta ní prost edí Red Console odkudkoliv v po íta i prostým zápisem slova **red** do p íkazového ádku systémové konzoly. Tento zp sob lze doporu it pouze v cí znalým, nebo systémová prom nná rozezná pouze bazální název red.exe. Z toho d vodu je nutné bu   instala ní soubor na disku p ejmenovat nebo vytvo it soubor red.bat.

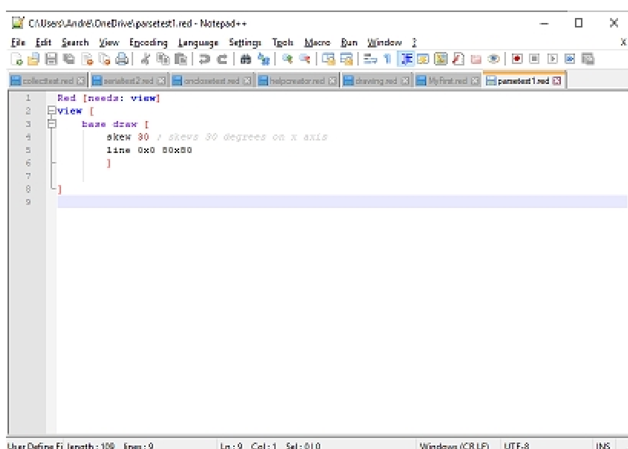
Další p ístup k interpreta nímu prost edí Redu vytvo íme nastavením doporu ovaného editoru Notepad++ a ješt   další p ístup nastavením rovn ž doporu ované konzoly Console2.

Pokyny pro spoušt ní skript   jsou popsány v kapitole Hello world, nejprve si však musíte vybrat textový editor

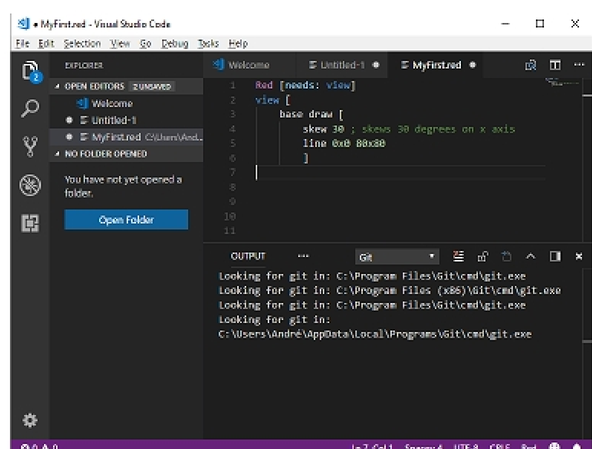
Výb r editoru

Skripty lze psát v jakémkoli textovém editoru, který produkuje   isté textové soubory. V našem Pr vodci se omezíme na prezentaci populárního a ú inného editoru Notepad++ , který byl použit pro všechy skripty p íklad   , spolu s konzolovou aplikací Console 2.00, ve které jsem m l otev ené interaktivní prost edí Redu pro zadávání jednoduchých kód .

Alternativou k Notepad++ je editor a programovací prost edí Visual Studio.



Notepad++

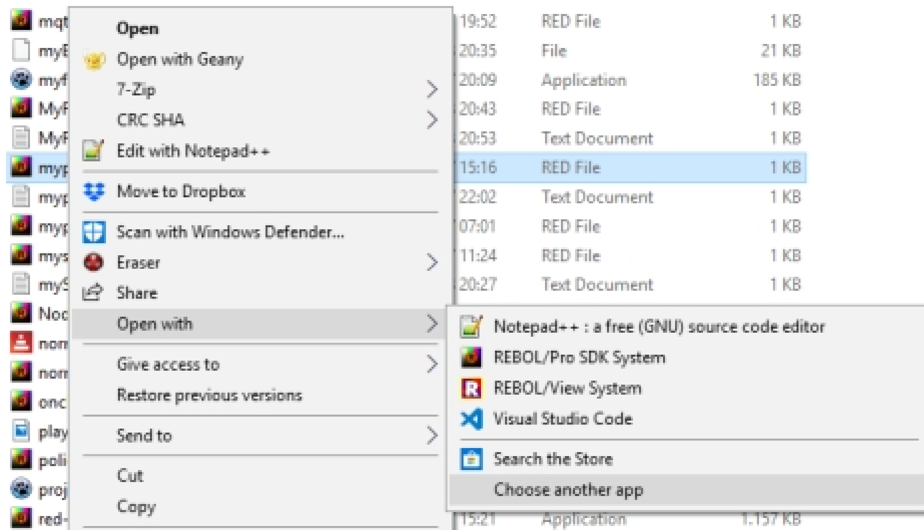


Visual Studio

N které užite né informace:

Skripty Redu jsou isté 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 výstižn ilustruje vytvo ení této asociace. Pravým klikem na souboru s extenzí *.red* otev eme kontextovou nabídku, kde vybereme *Otev ít 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

Notepad++

Provizorní IDE v Notepad++

P kné vývojá ské prost edí Redu ve Windows lze vytvo it s použitím editoru Notepad++. M žete postupovat podle následujících instrukcí nebo si stáhnout instala ní složku Red_IDE. Tato zazipovaná složka obsahuje krom   pot ebných konfigura ních soubor   také aplikace notepad++.exe a red-063.exe.

Pokud si ji stáhnete, máte v jeho (anglickém) Notepadu++ již p ednastavený jazyk Red-lang i položku Red-run v menu Run, takže dále popisovaná nastavení nemusíte provád t. Složka Red_IDE dokonce obsahuje p ipravenou pracovní složku Programy.

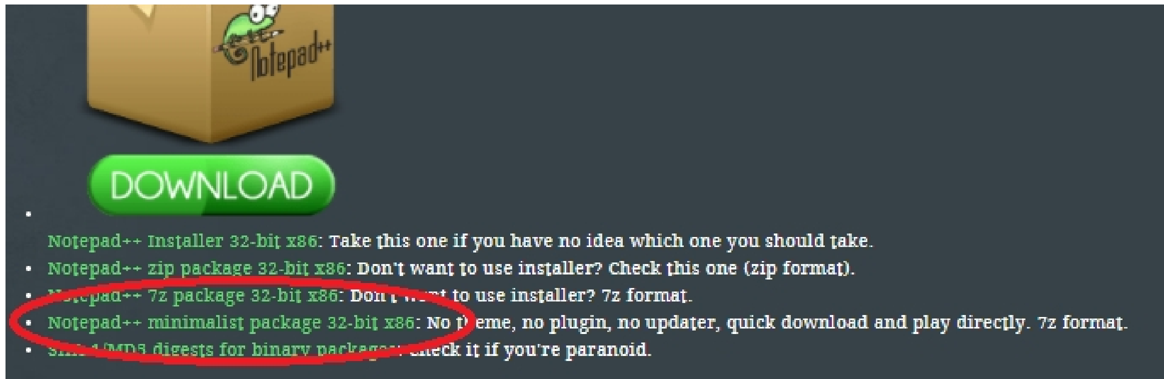
První spušt ní red-063.exe ve složce RedIDE chvíli trvá, než otev e konzolu, protože se instaluje a provádí sestavení aplikace Red.

Následující pokyny popisují postup pro instalaci z individuáln   stažených instala ních

soubor :

1 - Vytvo te v po íta i složku **RedIDE**/;

2 - Stáhn te si aktuální [Notepad++](#) . Ur it si vyberte "minimalistický paket **32-bit x86**".

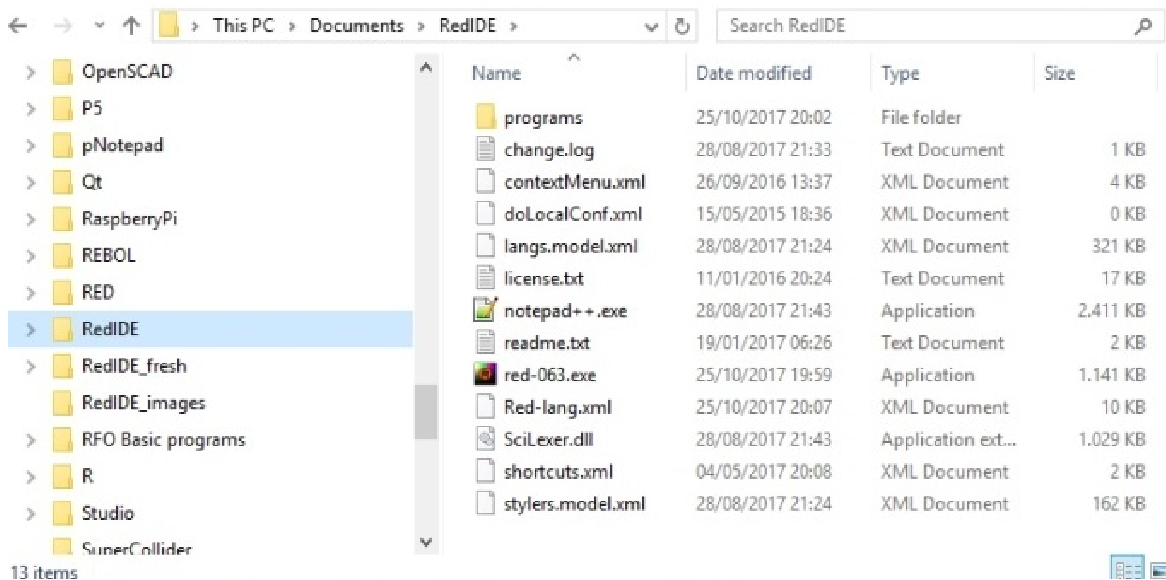


3 - Rozbalte Notepad++ ve vytvo ené složce RedIDE/.

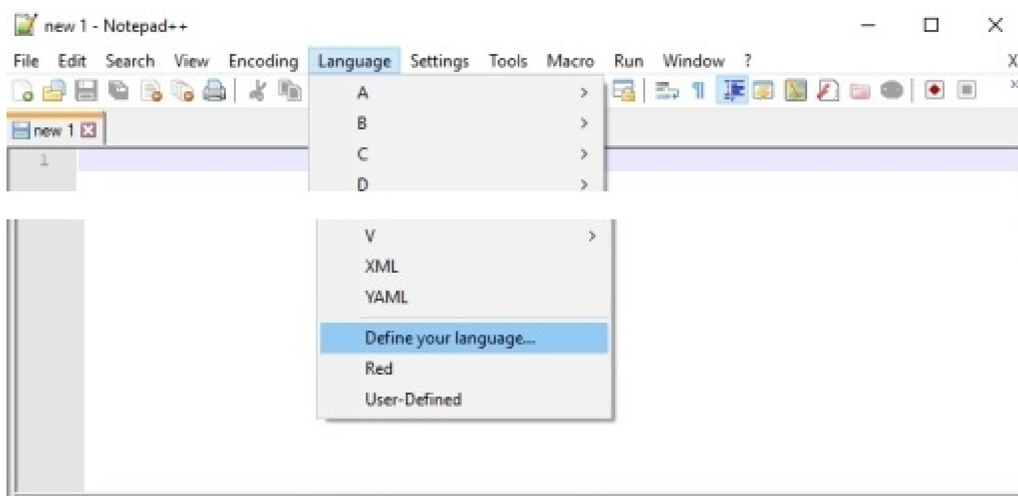
4 - Stáhn te si aktuální verzi [Redu pro Windows](#) a p emíst te ji do složky RedIDE/.

5 - Stáhn te si konfigura ní soubor [Red-lang.xml](#). Vložte jej rovn ž do složky RedIDE/.

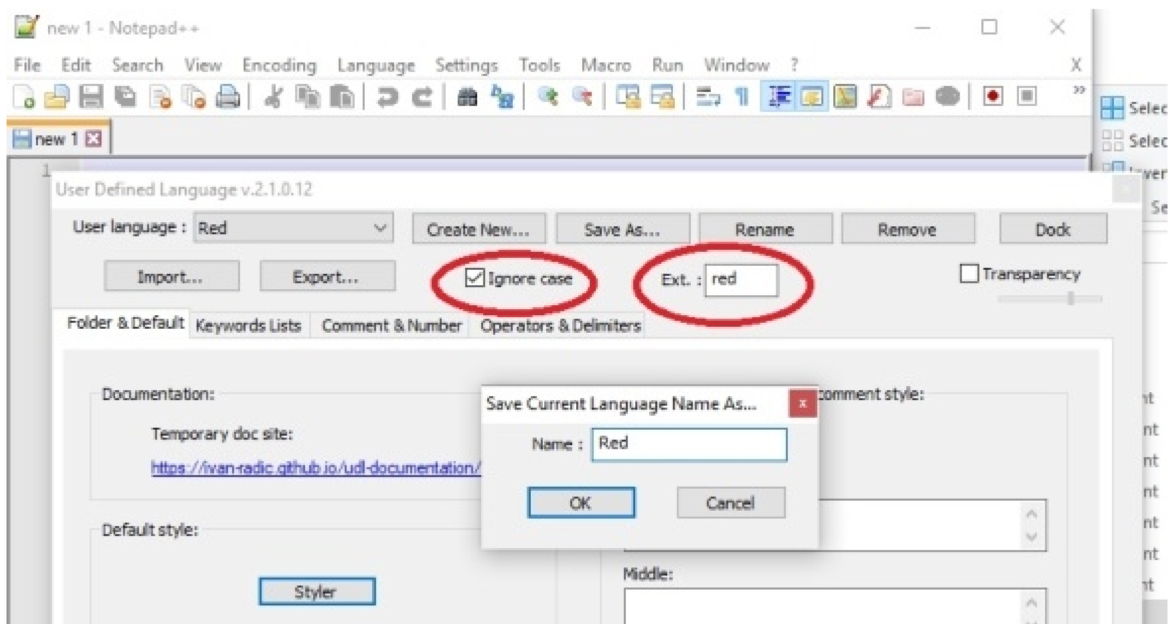
6 - Rovn ž doporu uji, aby jste si uvnit složky RedIDE vytvo ili ješt složku **Programy** pro ukládání program . Složka RedIDE nyní m že vypadat takto:



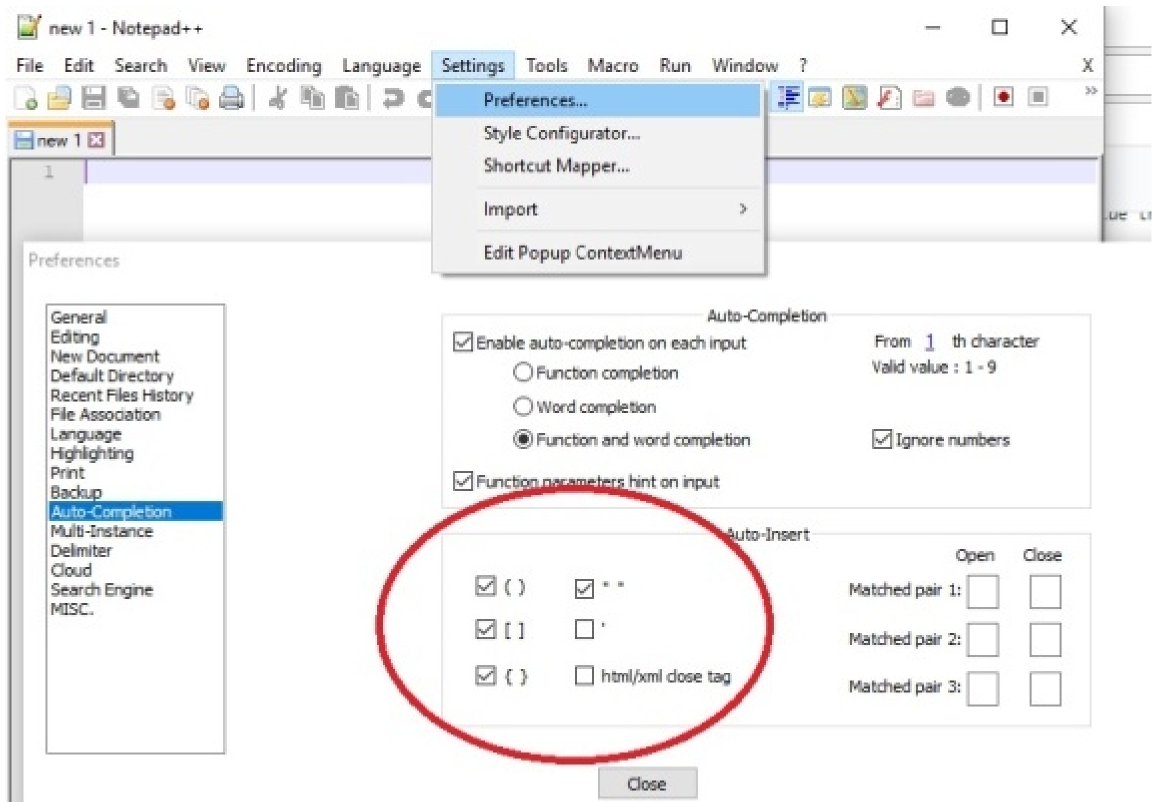
7 - V otev eném Notepad++ aktivujte nabídku “Syntaxe/Definovat vlastní syntaxi…” (p ípadn Language/Define your language...) . V otev eném okn klikn te na “Import…” a vyberte stažený “Red-lang.xml”. Vysouvací náv stí vám ekne, že byl import úsp šný.



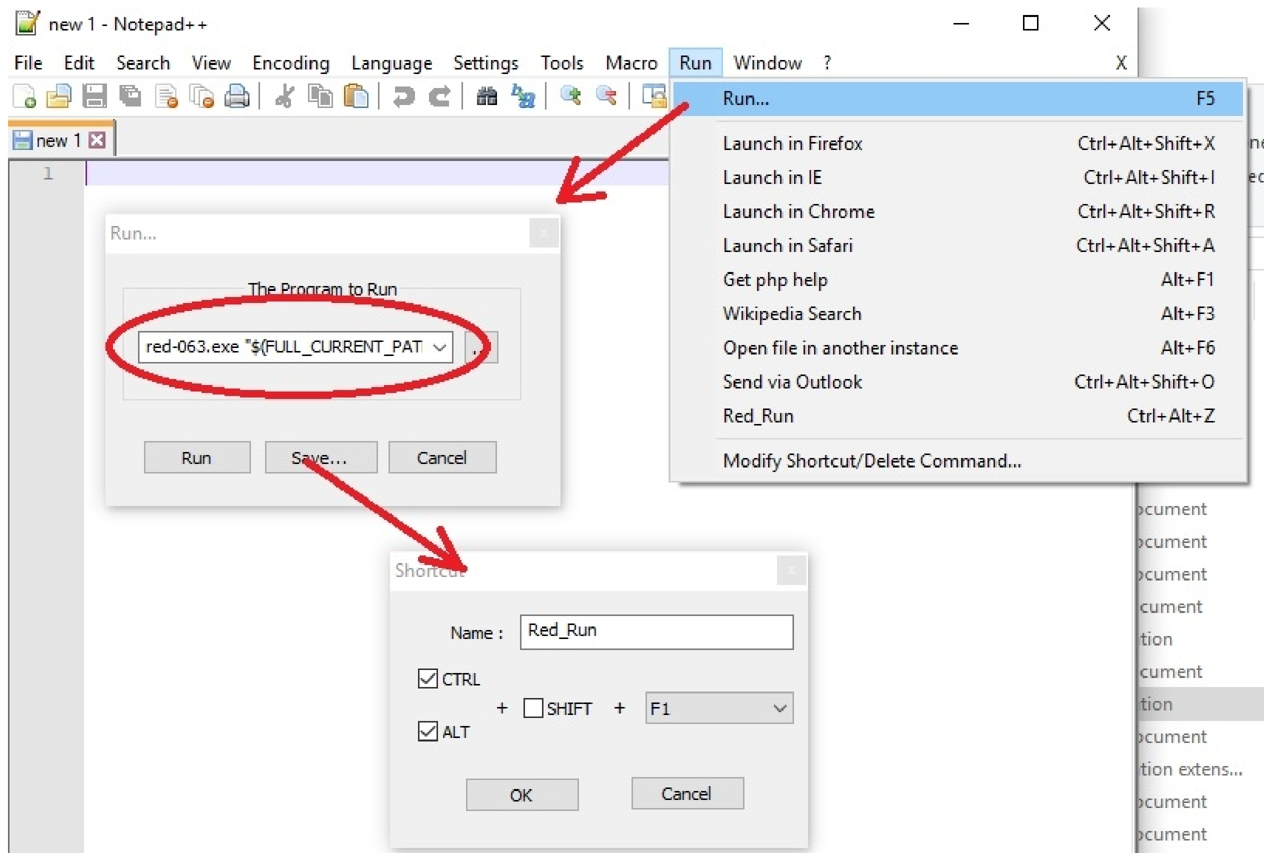
8 - Zatrhn te polí ko “Nerozlišovat malá a velká písmena”. Klikn te na “Uložit jako…” a uložte jako “Red”. Potom napište “red” do polí ka “P ípona”. Obrázek dole je ve fázi "Uložte jako".



9 - Otev te okno “Nastavení/Volby/Automatické dokon ování”. Zatrhn te polí ka pro “()”, “"” “{}” and “[]”. (If you can t find those in the options, create a “matched pair”.)



10 - Otev te 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 - nap íklad “Red_Run” (pamatujte, že pozd ji m žete chtít n co jako “Red_Compile” etc.) a klikn te “OK”.

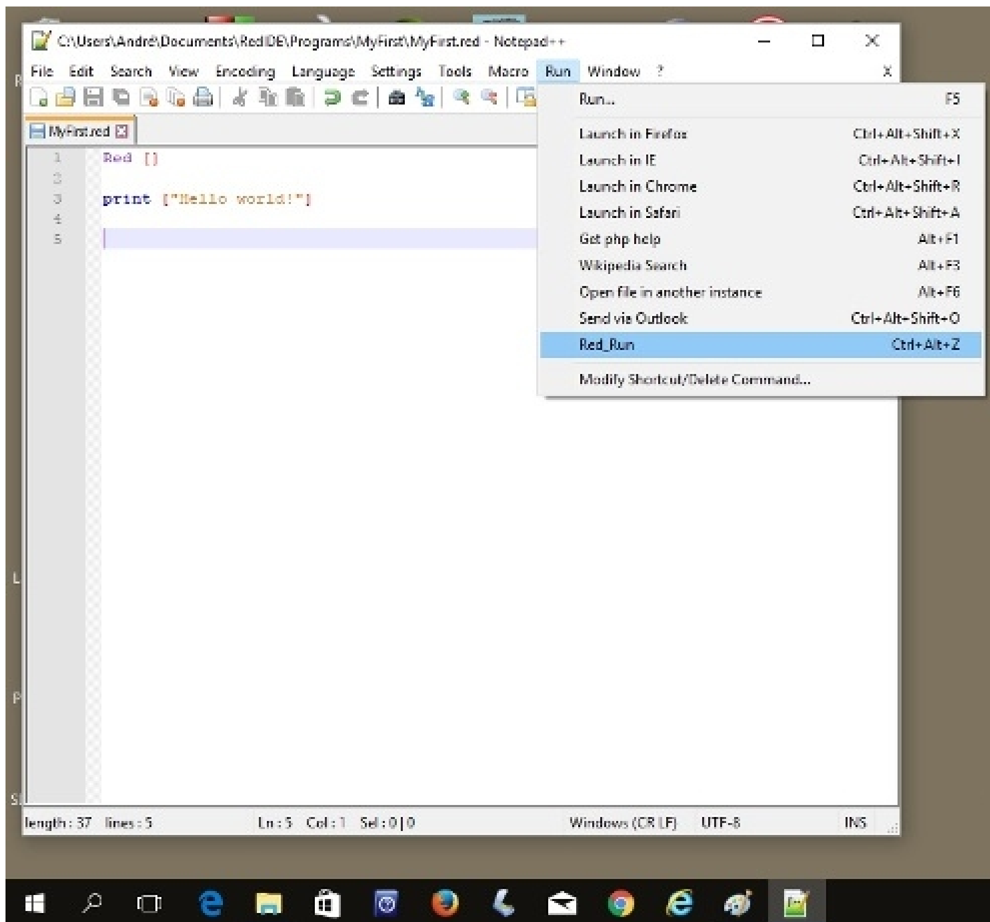


11 - Zav ete a restartujte Notepad++.

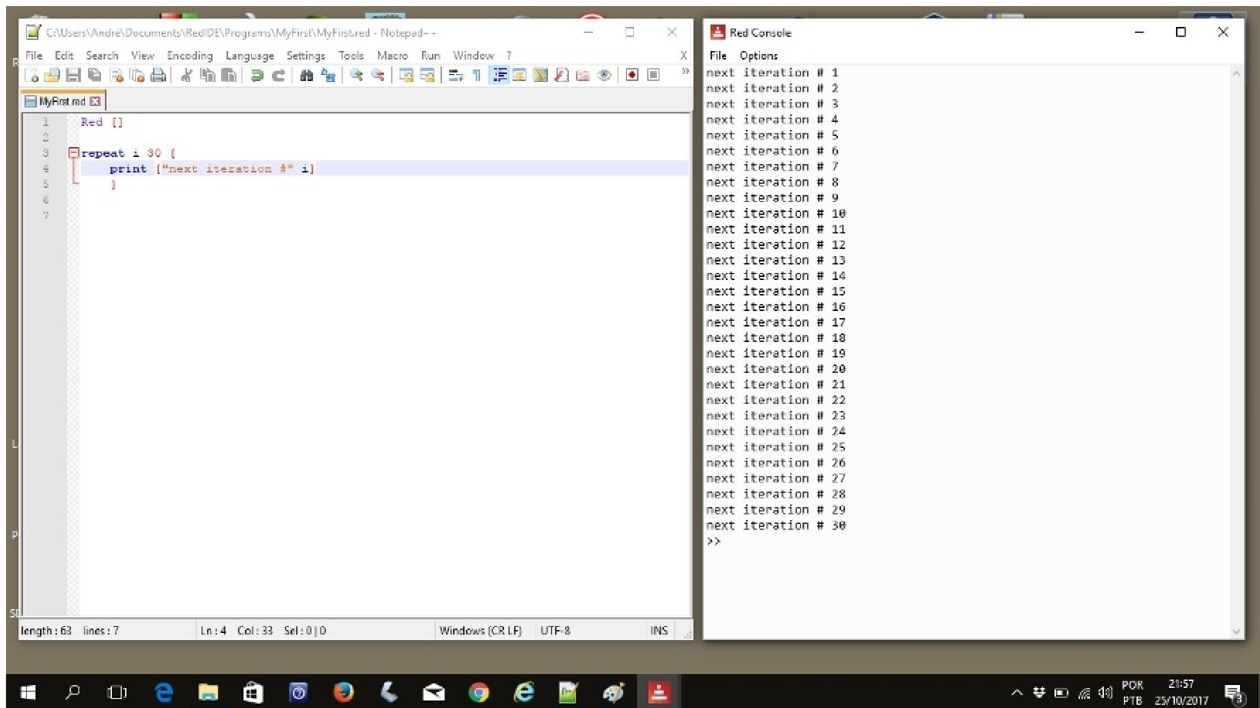
12 - Otev te v Notepad++ nový dokument a opište dolní text. V p ípad tohoto "Red[]" je Red výjime n "case sensitive".

```
Red[]
print "Hello world!"
```

13 - Uložte program s extenzí .red (“MyFirst.red”). V zapsaném skriptu se zvýrazní vybraná slova. Doporu uji jej uložit do vytvo ené složky Programy/. Klikn te “Spustit/Red_Run”. Skript by se m l otev ít v grafické konzole Redu.



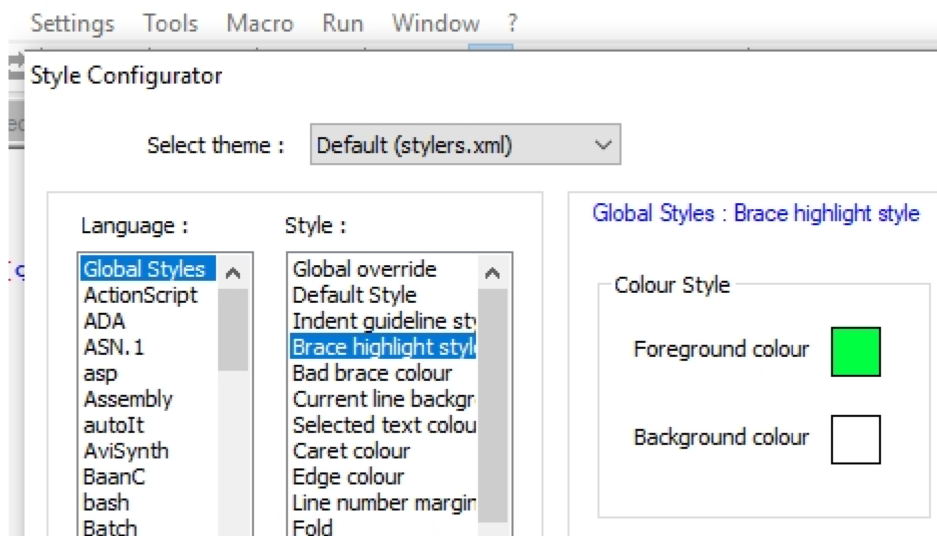
14 - Upravte rozm ry obou aplikací a míst te je úhledn vedle sebe a máte hotové IDE.



15 - Siln doporu uji vytvo it si kopii složky “RedIDE/” jako nap íklad “RedIDE_backup”.

16 - Nyní sv j skript zapíšete v Notepad++, **uložíte jej** a spustíte pokynem “Run/Red_Run”.

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



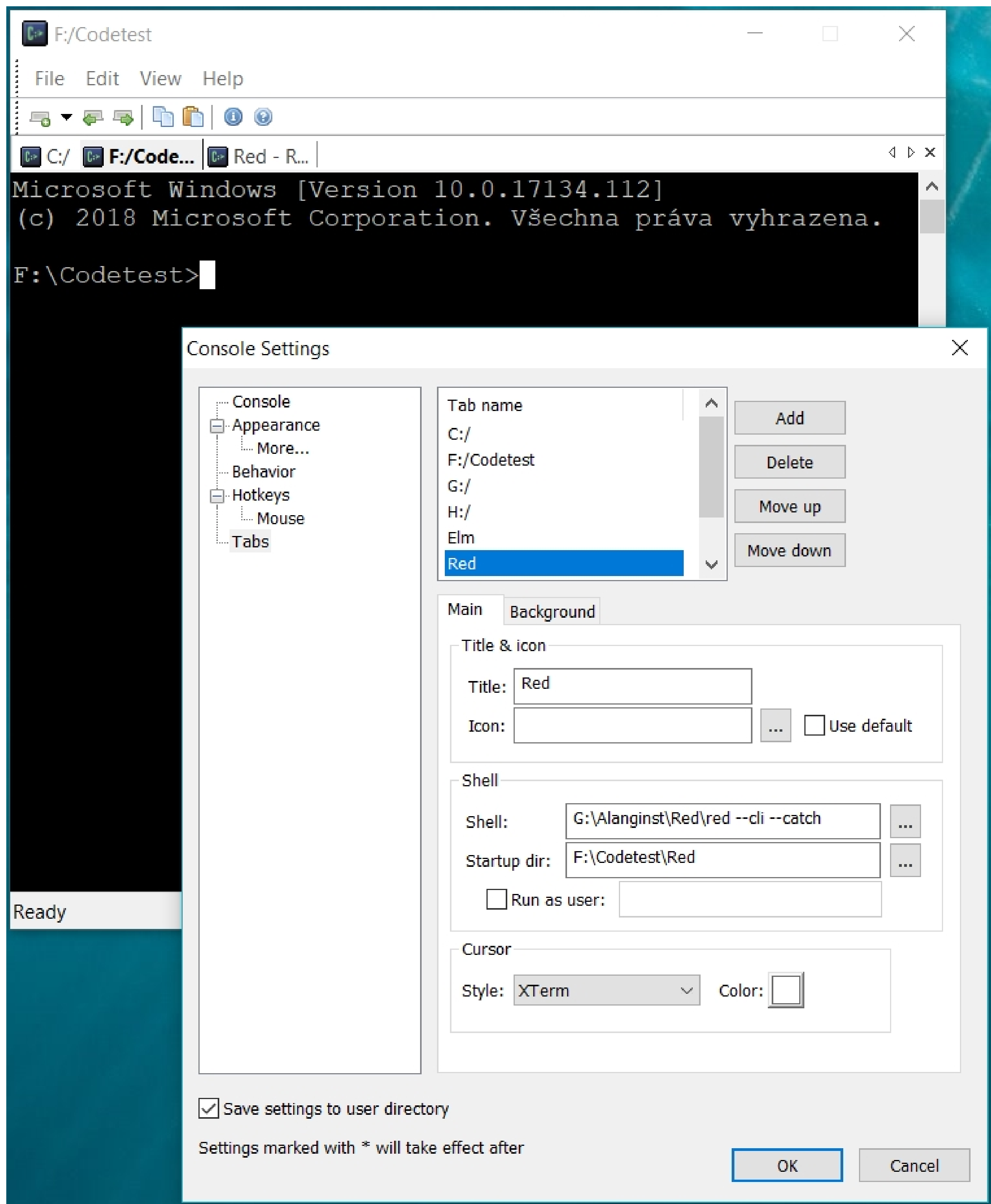
Znovu p ipomínám, že tato adaptace je možná pouze s Notepad++ 32-bit.

Created with the Standard Edition of HelpNDoc: Create HTML Help, DOC, PDF and print manuals from 1 single source

Console 2

Výborné prost edí pro práci s interaktivní konzolou Redu ve Windows je aplikace **Console 2.00**, kterou si lze zdarma stáhnout ze stránky SourceForge. Velikost instalace je menší než 2 MB.

V instalované konzole volíme **Edit** > **Settings**... kde zvýrazníme volbu **Tabs**, která nám rozbalí níže zobrazený formulá . V tomto formulá i zadáme název karty, p ípadn cestu k souboru s ikonou.
V další skupin (Shell) zadáme cestu k binárnímu souboru red.exe a p íslušné volby (viz obrázek). Na dalším ádku zadáme cestu ke složce, v níž se budou ukládat soubory, s nimiž bude interaktivní konzola Redu pracovat.



Výhoda této aplikace spo ívá v tom, že si lze najednou otev ít více karet s r znými programy i cestami, p ípadn tentýž program i cestu otev ít vícekrát s r zným obsahem.

V obrázku ilustrovaná úprava aktivace otevíraného souboru p íponami --cli --catch zp sobí, že se ve zvolené kart otev e p ímo interpreta ní konzola Redu (nikoliv Red Console). Pokud si na další kart nastavíme Red bez p ípon, bude nám otevírat voln p emistitelnou konzolu Red Console.

Zadaný p íkaz lze po vyhodnocení opakovat (i vícekrát) šipkou nahoru. Po ádku se lze

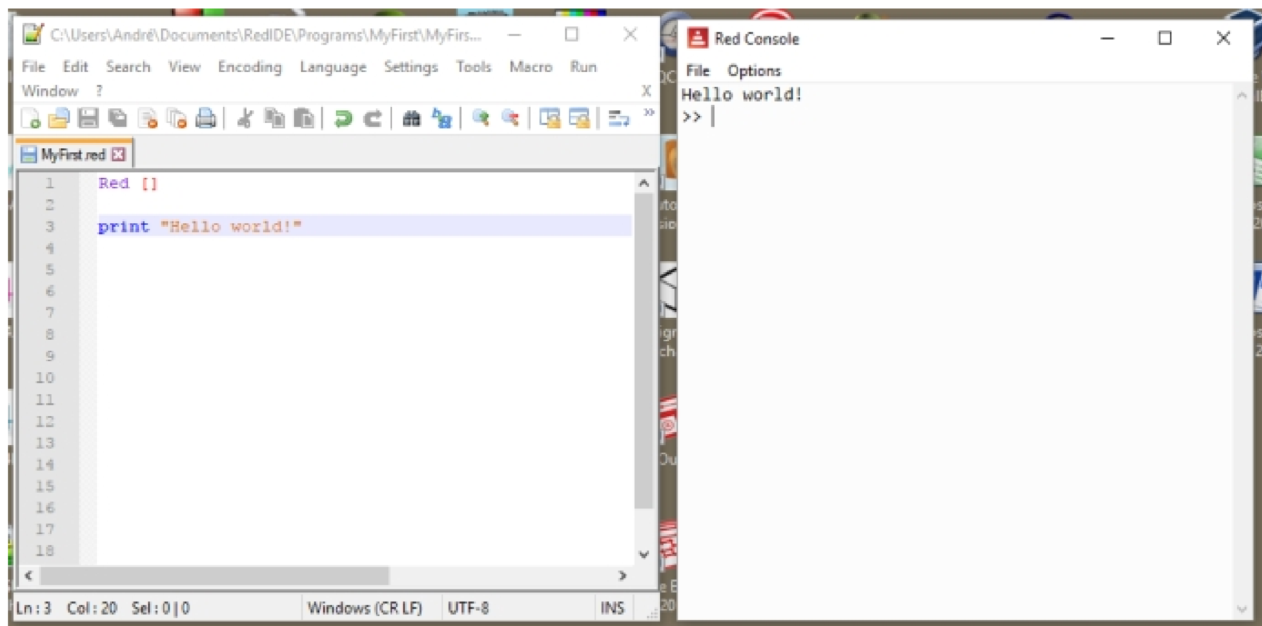
pohybovat šipkami vlevo i vpravo a klávesami *Home* i *End*. Interaktivní prost edí Redu opustíme p íkazem **q** nebo **quit**.

Created with the Standard Edition of HelpNDoc: Generate Kindle eBooks with ease

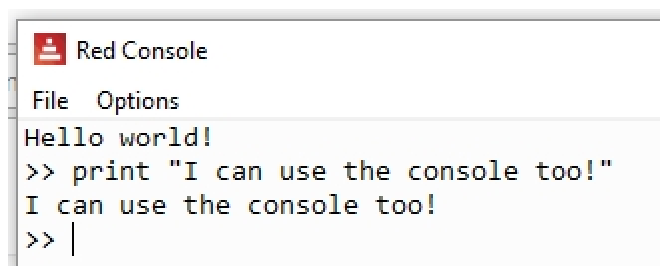
"Hello world" - spušt ní a kompilace

"Hello world" v konzole:

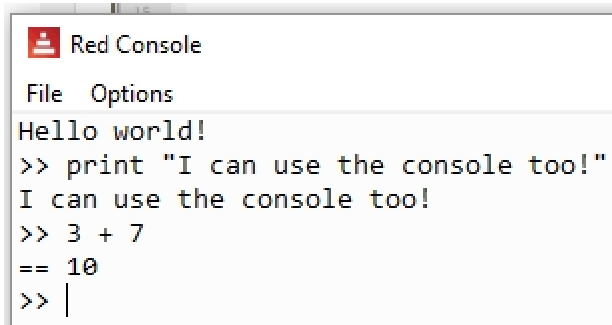
Otev te v Notepadu++ již vytvo ený skript "MyFirst.red" a spus te jej (run/red-run). M l byste v konzole dostat toto:



Okno vpravo je interaktivní konzola Red Console, n kdy obecn ozna ovaná jako REPL, což je zkratka sousloví Read, Evaluate, Print, Load . V její ploše zapište `print "I can use the console too!"` a stiskn te Enter. Na to konzola reaguje výpisem zadaného textu na následujícím ádku:



Nyní zadejte 3 + 7 a stiskn te Enter:



Všimn te si, že musíte mít mezery mezi jednotlivými slovy. Mezery jsou povinné odd lova e 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
```

Všimn te si, že jsem za 3+7 p ipsal `; no spaces!!!!!` . Red ignoruje slova za st edníkem.

Zpracování skriptu:

Interpretované programovací jazyky vykonávají kód skriptu ádek po ádku. Red není zcela interpretovaný, protože provádí jistou kompilaci p ed exekucí ale jeho program m se p esto íká skripty.

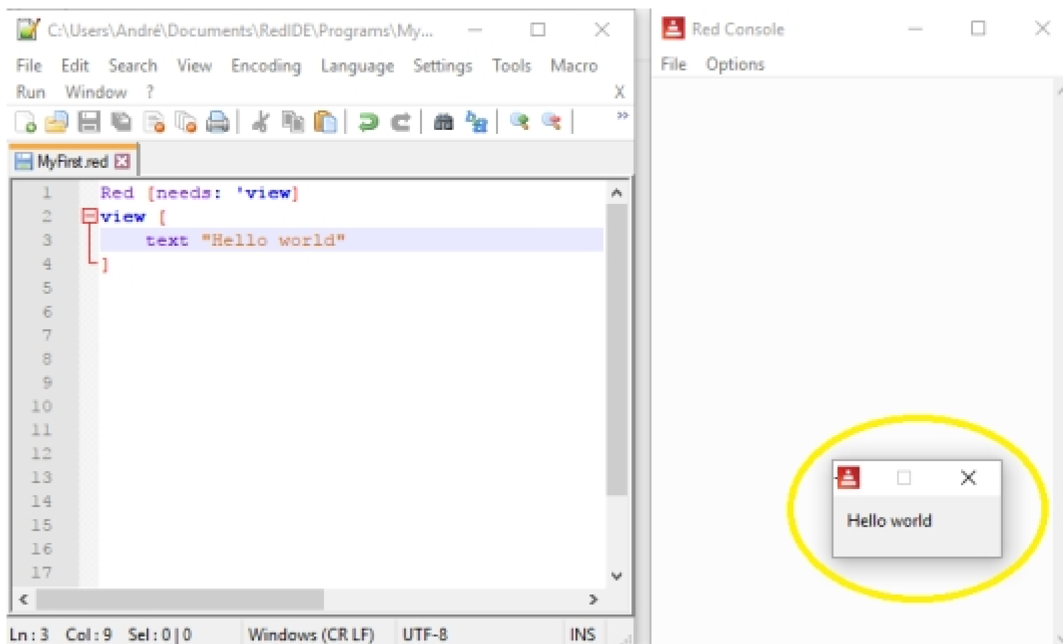
Na prvním ádku skriptu je povinn   blok `Red [ ]`, nikoliv RED, nikoliv red. Tento první blok je ur en pro metadata programu, nicmén   m že být prázdný. Úplný blok by mohl vypadat takto:

```
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 p ed tímto blokem je ignorováno!

print "Hello World!"
```

"Hello world" s grafickým rozhranním - GUI:

Jednou z nejnápadn jších vlastností Redu je jeho schopnost snadného vytvá ení program s grafickým rozhraním. Chyt e využívá vlastní API opera ního systému. Jednoduchý prográmek pro Hello world v GUI vypadá takto:

```
Red [needs: 'view]

view [
    text "Hello World!"
]
```



Pokyn `needs: 'view` v bloku záhlaví íká Redu, že má na íst grafickou knihovnu "view". Toto záhlaví s pokynem není pot ebné, zadáváme-li p íkaz `view [text "Hello World!"]` z konzoly Redu, v níž je knihovna "view" již obsažena.

Není však z ejm nic proti ni emu, když i v konzole uvedeme `needs: 'view` - již bez bloku Red [].

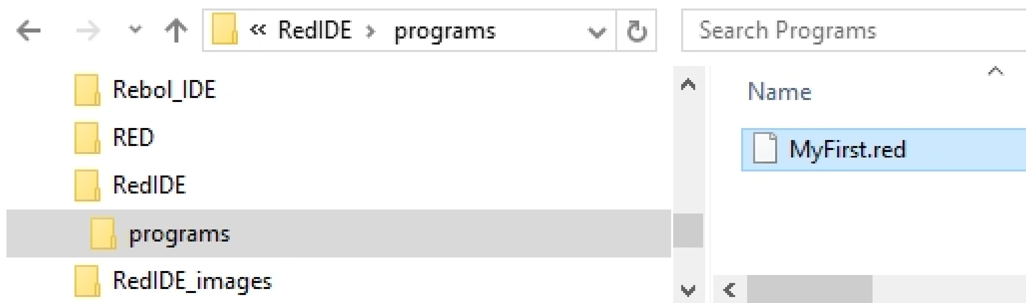
Spušt ní skriptu z konzoly:

Spušt ní kódu, zapsaného v konzole provedeme prostým p íkazem Enter, spušt ní kódu, zapsaného ve skriptu, provedeme v Notepad++ p íkazem Run/Red Run. Spušt ní skriptu z konzoly je popsáno v kapitole Exekuce kódu.

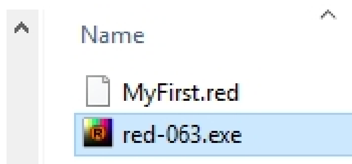
Kompilace "Hello world" na spustitelný soubor:

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

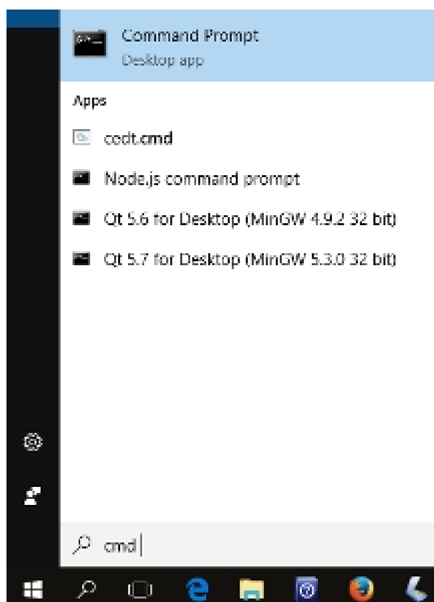
Máte-li výše uvedený program uložen jako "MyFirst.red" ve složce "programs" v adresá i "RedIDE", m l byste mít v po íta i p ibližn následující konstelaci:



Aby se nám produkty kompilace nerozsypaly do složky RedIDE, vložte do složky "programs" ješt kopii provád ky Redu:



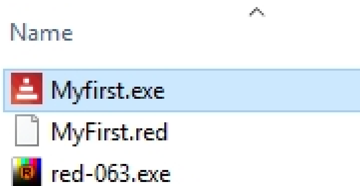
Otev te systémovou konzolu. Nevíte-li jak, napište "cmd" do vyhledávacího pole Windows a potvr te položku P íkazový ádek (Command Prompt):



V p íkazovém ádku se nacédujte do složky s provád kou Red (práv jsme ji p ekopírovali do složky "programs") a zadejte žlut ozna ený text dole:

```
C:\Users\André\Documents\RedIDE\programs> red-063.exe -r -t windows Myfirst.red
```

Red odpoví adou sd lení a asi po 10 vte inách budete mít ve složce "programy" p t dalších soubor , v etn "Myfirst.exe".



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

Dopl ující poznámky ke kompilování:

Zjistil jsem, že se kompilovaná verze programu m že chovat odlišn od interpretované verze. M l jsem problémy s p íkazy "print", které jsem do programu vložil kv li lad ní. Mám zato, že volání p íkaz konzoly v binárním (executable) souboru není to pravé o echové. Také jsem m l problémy s globálními prom nnými (slovy) uvnit funkcí; zdá se že kompilátor je neozna í jako globální. Poslední problém jsem ešil dvojím zp sobem:

1. Deklaroval jsem prom nné (p i adil hodnoty slov m) na po átku svého programu.

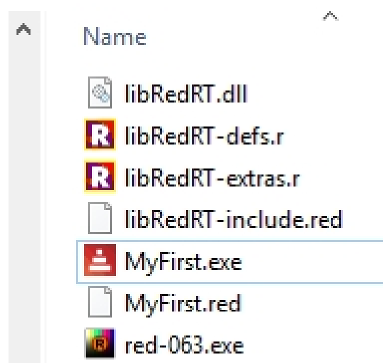
2. P i kompilaci jsem použil opci "-e" (v github není uvedena, patrn jde o experimentální volbu).

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

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

```
C:\Users\André\Documents\RedIDE\programs> red-063.exe -c MyFirst.red
```

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



Jediné dva soubory, které musí být k disposici pro spušt ní vašeho programu jsou **libRedRT.dll** a vaše provád ka (binárka), v tomto p ípad **MyFirst.exe**.

Když ovšem spustíte binárku, kompilovanou pouze s opcí `-c`, otev e Red také obt žující okno konzoly. Chcete-li se tomu vyhnout (kdo by necht l?), zadejte ješt opci `-t windows`, která zajistí kompilaci pro ur enou platformu.

```
C:\Users\André\Documents\RedIDE\programs> red-063.exe -c -t windows Myfirst.red
```

M li byste být schopni kompilace pro níže uvedené platformy ale v sou asné dob se Red stále vyvíjí a tak se m že stát, že s n kterými platformami budete mít problémy (nap . 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
```

Zp soby 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 "Dopl ující poznámky ke kompilování" výše.

Created with the Standard Edition of HelpNDoc: Full-featured EPub generator

Poznámky ke skladb

- Red je 'case insensitive' - s n kolika málo výjimkami, z nichž nejd ležit jší je ta, že program musí za ínat slovem **Red** (nikoliv RED nebo red).
- Znaky new-line jsou interpretem Redu v tšinou ignorovány. Relevantní výjimkou je `new-line` uvnit et zce (stringu).
- Red je funkcionální jazyk, což znamená, že vyhodnocuje výsledky. Po adí vyhodnocení výrazu není obvyklé a hovo íme o n m v ásti Vyhodnocení.
- Program Redu je dlouhý et z slov (words). T mito slovy mohou být bu "data" nebo "aakce".
- Slova jsou odd lena jednou nebo více mezerami (whitespaces) .
- Red si vede slovník se systémovými slovy a s uživatelsky vytvo enými slovy.
- Slova mohou být seskupována do "blok ", vymezených hranatými závorkami. Blok je pouhá skupina slov, která m že ale nemusí být vyhodnocena n jakou "akcí".
- Všechna programová data se nacházejí uvnit samotného programu. Jsou-li pot ebná externí data, jsou p idána k et zci slov programu.
- Každé slovo musí p i vyhodnocení mít n jakou hodnotu. Tato hodnota m že pocházet z:
    - slova samotného, je-li datem;

- vyhodnocení, je-li slovo akcí;
- odkazu na jiné slovo nebo blok (e.g. `myRoom: 33`).

- V Redu lze íci, že prom nná je p i azena své hodnot , nikoliv obrácen . V Redu vlastn   nejsou žádné "prom nné", pouze slova p i azená k hodnotám (dat m).

- Kopírování slov (prom nných) m že být komplikované. Chcete-li vytvo it opravdu nezávislou kopii, použijte slovo `copy` . Viz kapitola Kopírování.

- Podobn   komplikované je vyprazd ování (clearing) ad (series), k nimž pat í i et zce. Pouhé p i azení " " (empty string) nebo nuly nemusí p inést o ekávaný výsledek. Takže k nulování ady je vhodné použít p íkaz `clear`.

- Každé slovo je n jakým datovým typem. T ch má Red pozoruhodné množství. Jsou vyjmenovány v kapitole Datové typy.

- Deklarované slovo datového typu word! lze použít r zným zp sobem:

Zápis	Význam
word	Vzít p irozenou hodnotu slova. Je-li funkcí, nejprve ji vyhodnotit.
word:	P i adit (assign) slovo k hodnot .
:word	Vzít hodnotu slova bez vyhodnocení. Užite né pro získání definice funkce.
'word	Zacházet se slovem jako s hodnotou (a word symbol) - bez vyhodnocení.
/word	Považovat slovo za up esn ní (refinement). Používá se hlavn u volitelných argument .

Refinements (up esn ní)

Mnohé akce v Redu umož ují "up esn ní", které modifikuje chování p íkazu. Refinement se deklaruje p idáním slova za lomítkem "/<refinement>".

Created with the Standard Edition of HelpNDoc: Create cross-platform Qt Help files

Introspekce a nápov da

Red má mimo ádnou nápov du. Pouhým zápisem n kolika slov v konzole lze získat obsáhlé množství informací o jazyku a o konkretní entit .

function! ? (or help)

Poskytne informaci o všech vyhrazených slovech Redu a rovn ž o vašem vlastním kódu. M žete použít p íkaz `help` nebo `?`. V konzole se vytiskne informace o použití nápov dy.

```
>> ? 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 p esn nevíte co hledáte, pom že 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?]
```

M žete 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 globáln definovaných funkcí. Zkuste si to!

function! source

Zobrazí zdrojový kód mezaninové nebo uživatelem vytvo ené funkce. Vyzkoušejte si nap íklad

>> `source replace`.

function! about

Zobrazí íslo verze a datum sestavení.

Introspekce grafických objekt

Introspekci grafických objekt (piškot) si ukážeme v kapitole GUI.

Created with the Standard Edition of HelpNDoc: Easy to use tool to create HTML Help files and Help web sites

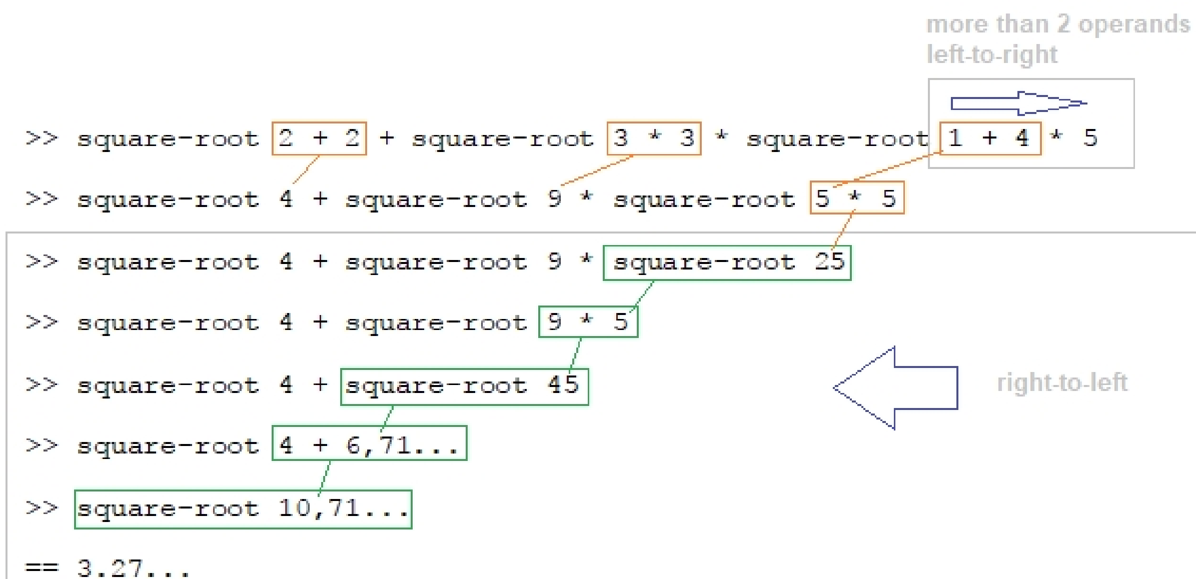
Vyhodnocení výrazu

Stále se rozhlížím po jednoduchém pravidle, vysv tlujícím proces vyhodnocování v Redu. Aktuáln mám dva oblíbené kandidáty. První je p ímo arý a snadno použitelný. Druhý není p íliš praktický ale dává nahlédnout (domnívám se) na zp sob, kterým interpret Redu "myslí".

1. První oblíbenec:

- Všechny operace s infixovými operátory, které mají pouze hodnoty (nikoliv funkce) jako operandy, jsou vyhodnocovány p ednostn . 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 t emi koncepty:

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

První koncept: Vždy zleva doprava →

V Redu se všechno vyhodnocuje zleva do prava. Neexistuje žádný systém preferencí jako u jiných jazyk (nap . se násobení neprovádí automaticky p en s ítáním). P ednost v po adí 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 itelná forma výrazu `multiply add 2 3 5` . Interpret Redu si pot ebnou konverzi provede sám.

Druhý koncept: Vyhodnotitelné skupiny.

V ur ité ásti kódu se vyskytují skupiny slov, které lze díl í exekucí redukovat na základní datový typ. Nap íklad `[square-root 16 8 + 2 8 / 2 77]` se ve skute nosti skládá ze ty vyhodnotitelných skupin: `square-root 16` ; `8 + 2` ; `8 / 2` a `77`. Popisovanou redukci lze provést p íkazem `reduce`:

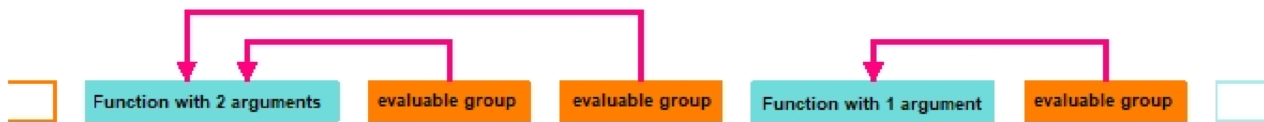
```
>> a: [ square-root 16 8 + 2 8 / 2 77]
```

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

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

T etí koncept: Funkce si vybírají své argumenty z vyhodnotitelných skupin

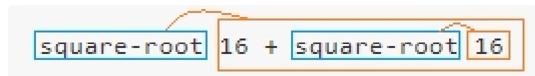
Funkce si vybírá své argumenty z následných skupin hodnot i výraz (po et o ekávaných argument je pot ebné znát) v po adí zleva doprava. Výrazy jsou p ed použitím funkcí redukovány na hodnoty.



D sledkem tohoto p ístupu je to, že výsledkem výrazu

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

není 8, jak by mnozí o ekávali, ale 4.47213595499958, protože Red vidí n co jako toto:



ímž je funkce pro jeden argument, stojící p ed skupinou s infixovým operátorem (majícím p ednost); operandy operátoru (+) jsou íslo a funkce, která se p ed použitím operátorem vyhodnotí.

Abychom získali on ch intuitivních 8, musíme použít závorky:

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

Alternativní sekvenci

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

m žeme íst jako

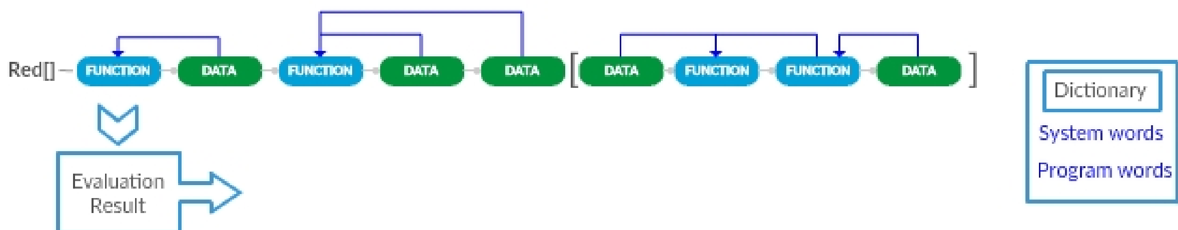
```
>> square-root (add 16 (square-root 16))  --> 4.47213595499958
```

Jiný p íklad, kombinující infixový operátor a jemu odpovídající funkci:

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

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

Pon kud zjednodušený pohled na tok výpo tu:



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

Jiné vysv tlení:

Nenad Rakocevic ve svém p ehledu Red/System Language Specification (red-lang.org - zá í 2017) uvádí toto stručné vysvětlení:

Výrazy se vyhodnocují zleva doprava. Operátory (infixové funkce) mají stejné po adí d ležitosti, krom toho, že mají p ednost p ed prefixovými funkcemi.

Created with the Standard Edition of HelpNDoc: Easily create Web Help sites

Zvýrazn ní kódu

Zvýrazn ní textu ve skriptu

Zvýrazn ní syntaxe je pro za áte níky velmi prosp šné - zejména v programovacím jazyce Red, s jeho mnoha p eddefinovanými slovy a stru ným kódem. Kdykoli je to možné, používám zvýrazn ný kód p evzatý z Notepad++.

```
Red [ ]
                        ; mezera kv li p ehlednosti
```

```
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 tu ným písmem, odezva interpreta normálním. To pom že p edejít zbyte ným chybám p i kopírování kódu z prezentovaných p íklad .

Pro lepší p ehlednost vkládám prázdné ádky mezi jednotlivé p íkazy. R žové komentá e jsou p idány p i 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

Konzola - vstup a výstup

Poznámka: vstup do ~ a výstup z konzoly m že být p i kompilování program problematický, nebo p i kompilaci je konzola mimo provoz.

native! print

`print` je funkce, která posílá data do konzoly. Za daty ješt do konzoly posílá znak `newline`, který je p i 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 ádek.

```
Red []

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

```
HelloWorld42
```

function! probe

`probe` vrací sv j argument bez vyhodnocení, na rozdíl od funkce `print`, která sv j argument vyhodnotí; `probe` m že být použit p i lad ní k zobrazení nezm n ného kódu.

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

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

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

Popsáno také zde, za p íkazem `mold`.

function! input

Vloží **string** (et zec) 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 et zec.

```
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

Exekuce kódu

Spušt ní exekuce

Sv j skript m žete uložit jako soubor s p íponou ~.red nebo ~.txt a spustit jej z p íkazového ádku jako argument aplikace Red, asi takto:

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

To spustí interpret Redu, otev e konzolu (REPL) a spustí váš skript.
V b žícím prost edí Redu však m žete provést exekuci kódu p ímo p íkazem `do`.

native! do

Vyhodnotí argumenty kódu, jinými slovy provede exekuci kódu. Argumentem m že být blok, soubor, funkce   i jiná hodnota.

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

P ed dalším pokra ováním nahlédn te do kapitoly Soubory.

Pokud jste nap íklad uložil skript jako myprogram.red, m žete jej realizovat tímto p íkazem:

```
>> do %myprogram.red
```

Nezapome te, že v tomto p ípad musí být interpret Redu a zdrojový soubor ve stejné složce; jinak musíte uvest také cestu ke skriptu.

M žete také na íst skript do interpreta Redu:

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

a spustit jej:

```
>> do a
```

M žete také na íst a provést skript, uložený jako textový soubor :

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

Uv domte si, že toto vše m žete provád t **z vnit ku programu Red!**

Zastavení exekuce

function! quit

Zastaví vyhodnocení a opustí program.

Zapíšete-li to v konzole GUI, zav e se. Zapíšete-li to na p íkazový ádek interaktivního rozhraní, pouze opustíte interpreta Redu.

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

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

function! halt

Zastaví interpretaci skriptu.

routine! quit-return

;; koketuje s OS, opatrn !

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 ízená událostí - spustí zadaný kód po zav ení aplikace (okna GUI). Spus te následující program. Vytvo íte tla ítko a když na n j kliknete, tla ítko 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

Datové typy

Každá hodnota Redu má sv j specifický **datový typ**, který ur uje rozsah p ípustných hodnot, p ípustné operace a zp sob uložení hodnot v pam ti. Datových typ   je aktuáln   46, v etn   datového typu datatype! Výpis datových typ   získáme dotazem:

```
>> ? datatype!
```

Z t chto 46 datových typ   je vytvo eno 16 skupin (typeset), majících spole né vlastnosti. N které datové typy se vyskytují ve více skupinách. Výpis t chto 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 klí ovým pojmem v jazyce Red. Vyskytují se ve   tveré typové form :

word - typ `word!`, základní formát pro p irozenou hodnotu slova

word: - typ `set-word!,` formát pro p i azení 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, ohrani ený kulatými závorkami. Používá se hlavn k vyjád ení preference aritmetických operací.

Objekt typu paren! se vytvo í p íkazem 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!

Krom 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!

ada (series) znak uvnit uvozovek nebo složených závorek. Zabírá-li et zec více než jeden ádek, je použití {} jedin   možné.

Manipulace s et zci se provádí s použitím p íkaz , popsaných v kapitolách o adách.

```
>> 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 uvnit uvozovek, p edstavuje hodnota typu `char!` kódový bod Unicodu. Jsou to celá ísla v rozsahu hexadecimal 00 po hexadecimal 10FFFF (0 až 1,114,111).

#"A" je `char!`

"A" je `string!`

Umož uje ur ité 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 ísla. Od –2,147,483,648 po 2,147,483,647. Je-li íslo mimo tento rozsah, p i adí mu Red datový typ float!.

Poznámka: D lení 2 celých ísel dává oseknutý výsledek:

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

float!

64 bitová desetinná ísla. Prezentují se s desetinnou te kou i árkou 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 (/); zp tná lomítka (\) jsou ve Windows automaticky konvertována na oby ejná 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 p ístupu k interním položkám v tších struktur pomocí lomítka (/). Používá se v r zných situacích, nap íklad:

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

Lomítka "/" se také používají p i p ístupu k objekt m a k up esn ním.

time!

as je vyjád en ve formátu hours:minutes:seconds.subseconds. Všimn te si, že sekundy a subsekundy jsou odd leny te kou, nikoliv dvojte kou. Ke každé ásti je možný p ístup p es up esn ní (refinement). Nahlédn te do kapitoly Práce s asem.

```
>> 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 r zných 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 rovn ž ov uje platnost datum , v etn s uvážením p echodných let. Na den, m síc i rok se lze odkázat up esn ním:

```
>> 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 úpln stejné. Uvád jí celo íselné sou adnice v kartézském sou adnicovém 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 p idáním znaku "%" za íslem.

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

tuple!

Tuple (entice) je výpis t í až dvanácti celých ísel o velikosti 0 až 255, odd lených te kami. Entice jsou užite né pro prezentaci ísel verzí, IP adres a barev (nap .: 0.255.0).

S enticí lze provád t 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 p ehled barev ve vyjád ení 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

Další datové typy:

issue!

Datový typ issue! (emise) je sekvence znak , použitelná pro vyjád ení telefonního nebo seriového ísla i ísla kreditní karty. Emise za ínají znakem (#) a pokra ují až k prvnímu odd lova i (nap . k meze e). Uvnit emise lze použít v tšinu znak 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 ást uvádí komunika ní protokol (http, ftp, nntp, mailto, file, ...), za lomítky je cesta k odkazované entit .

```
>> 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 ada, která obsahuje RGB zobrazení. Podporované formáty jsou GIF, JPEG, PNG and BMP. Obrazy se obvykle na ítají 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 **up esn ní**; indikuje variaci (za lomítkem) v použití nebo rozší ení 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, nap íklad `add` , `take` , `append`, `negate, ...`

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

Seznam všech funkcí typu action! získáme p íkazem

```
>> ? action!
```

op!

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

routine!

Používá se pro p ipojení externího kódu.

binary!

Binární et zce se zapisují do složených závorek s p edsazeným znakem #. V tomto formátu lze uložit libovolnou sekvenci bajt (image, audio, spustitelný soubor, komprimovaná i zašifrovaná data).

Formát binárních dat m že být íslo se základem 2 (binár), 16 (hexadecimál) nebo 64 (tetrahexagesimál). Nep ítomnost íslice avizuje implicitní hexadecimální kódování se základem 16.

P íklad:

#{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.

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

Hash! vector! a map!

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

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

Doporu uji p ed dalším tením nahlédnout do kapitol Bloky a ady.

hash!

hash! je ada, upravená pro rychlejší vyhledávání. Protože "hašování" spot ebovává zdroje, nemá cenu vytvá et hash! pro ady, které budou prohledávány jenom n kolikrát. Pokud však má být vaše ada prohledávaná asto, vyplatí se z ní vytvo it hash! . Webová stránka Rebolu tvrdí, že vyhledávání m že být 650 krát rychlejší než u normálních ad.

```
>> 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 ady, jejichž hodnoty jsou intern vnímané jako skupiny dvojic.

vector!

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

K vytvo ení vektoru použijete p íkaz make vector!

Zatímco **hash!** umož uje rychlejší vyhledávání, **vector!** umož uje rychlejší provád ní matematických operací, nobo jsou provád ny pro celou adu najednou.

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

```
>> print a
33 44 52

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

Všimn te si, že to nejde ud lat s blokem, pat ícím rovn ž 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 klí e s hodnotami (key1: val1 key2: val2 ... key3: val3).

Mapy **nejsou** adami. Nelze u nich uplatnit v tšinu p íkaz pro ady (series).

Pro vyjmutí hodnoty z mapy použijeme p íkaz `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

Konverze datových typ :

action! to

Konvertuje jeden datový typ na druhý, nap . `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 dv ísla 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

Bloky a ady

Bloky

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

Bloky jsou skupiny hodnot a slov, ohrani ené hranatými závorkami, nap íkad: [one block], [another block [block within a block]]. Hodnoty a výrazy v bloku se implicitn nevyhodnocují. Vyhodnocení bloku se evokuje funkcemi do a reduce viz Datový typ block!.

Blok lze vytvo it výpisem závorek a hodnot, nap íklad (literálová skladba):

```
[one 2 "three"]
[print 1.23]
```

nebo funkcemi make i 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!
```

ady (series!)

Slovo **series**! je ozna ení typové skupiny (typeset!), skládající se z t chto typ : block!, paren!, string!, file!, url!, path!, vector!, hash!, binary!, tag!, email!, image!.

Obecn je **ada** (serie) uspo ádaná sada hodnot r zných typ . Prvkem serie m že být cokoli z repertoáru: data, slova, funkce, objekty, a jiné serie.

Pole - arrays

Pole (array) netvo í zavedený datový typ. Je to kolekce indexovaných (viz Navigace adami) hodnot stejného typu. Jednorozm rné pole se ozna uje jako **ádková matice** (vektor), dvojrozm rné pole definuje **matici**. T írozm rné pole se n kdy ozna uje jako tenzor. ádková matice je tvo ena jednoduchým blokem, matici tvo í blok ádkových matic, více rozm rové pole lze vyjád it jako matici hodnot typu tuple!

Zde je p íklad pole o rozm rech 2 x 3:

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

Pro p ístup k jeho element m lze použít lomítko:

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

```
== 6
```

Nebo slovní ozna ení indexu:

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

Created with the Standard Edition of HelpNDoc: Free EPub producer

Navigace adami

- První element ady se nazývá **head** (elo). Jak uvidíme, nemusí to být vždy nutn první element;
- Za posledním elementem ady je n co, emu se íká **tail** (chvost). Je to místo bez hodnoty.
- Každá ada má "**entry index**", jehož nejlepší definicí by mohlo být: "místo, kde za íná použitelná ást ady". Mnohé operace s adami mají tento **"**entry index**"** jako startovní bod. **Entry index** lze posouvat vzad i vp ed s vlivem na výsledek operace.
- Každý element ady je ozna en íselným indexem, za ínajícím 1 (not zero!) u prvního elementu.
- Po ínaje pozicí **entry indexu**, mají elementy slovní ozna ení : "`first`" pro první, "`second`" pro druhý a tak dál až po "`fifth`".

Poznámka: Pojem "**entry index**" jsem si vymyslel, nebo v dokumentaci není. Tam je "**entry index**" ozna ován pouze jako "index", což se podle mého názoru m že plést s " íslem indexu".

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

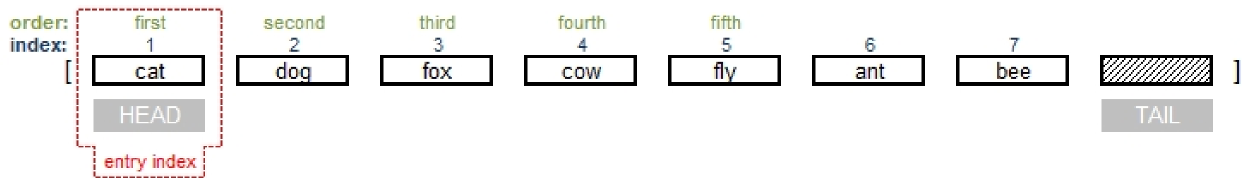
Tyto p íkazy vrací informaci o pozici **entry indexu**. Je-li entry index v ele ady, dotaz `head?` vrací `true`, jinak `false`. Tatáž logika platí pro `tail?`. Dotaz `index?` vrací íslo indexu místa, kde se aktuáln nachází **entry index**.

Následující p íklady vysv tlení objasní.

Vytvo me adu **s,** obsahující et zce **"cat" "dog" "fox" "cow" "fly" "ant" "bee"** :

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

Dostaneme n co, co vypadá asi takto:




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

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

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

action! head action! tail

P íkaz `head` p emístí **entry index** k prvnímu elementu ady, na její elo.

P íkaz `tail` p emístí **entry index** na pozici za posledním elementem ady, do chvostu.

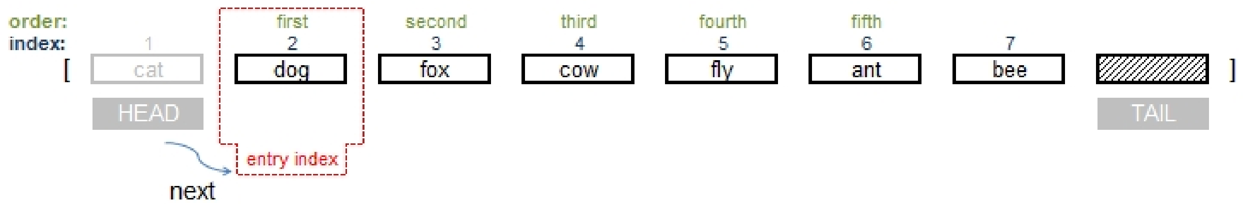
Umíst ní `head` a `tail` samo o sob adu nem ní, pouze `head` **vrací** celou adu a `tail` nevrací nic.

action! next

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

P íkaz `next` p emístí **entry index** o jeden element sm rem ke chvostu. Všimn te si, že `next` pouze **vrací** zm n nou adu ale nem ní ji. Opakované použití p íkazu `next` nezp sobí další posun vstupního indexu, protože p íkaz bychom aplikovali na p vodní, nezm n né ad .

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

P íkaz `back` je opakem p íkazu `next`: p emístí **entry index** o jeden element sm rem k elu. Provedete-li `back` v naší ad **s,** vrátí se "cat" zpátky na scénu! Nikdy nebylo zapomenuto!

To znamená, že Red staré **s** neodhodil. Je to jedna ze zvláštností Redu: data z stávají trvale vno ena v kódu.

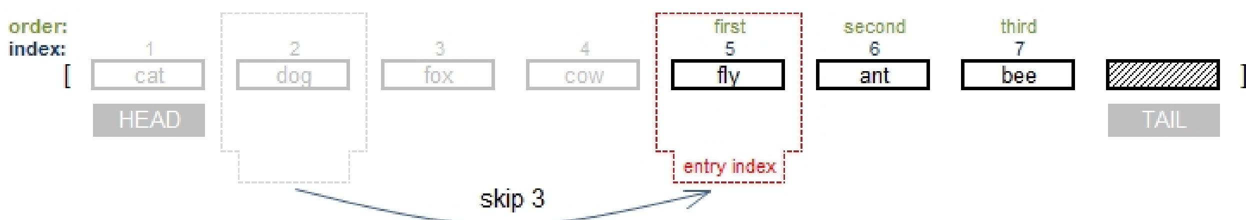
Poté, co jste p emístil index v naší ad **s** vp ed a p i adil ji k jinému slovu, nap . **b** (b: s), m žete provád t p esuny indexu vzad i vp ed pro **b** a dobývat "skryté" hodnoty **s,** protože **b** ukazuje na stejná data jako **s**.

Chcete-li vytvo it nezávislou kopii ady, musíte použít p íkaz `copy.`

Jak jsem se již zmínil d íve, v Redu jsou (na rozdíl od jiných jazyk) prom nné (slova) p i azeny k dat m a nikoliv naopak.

action! skip

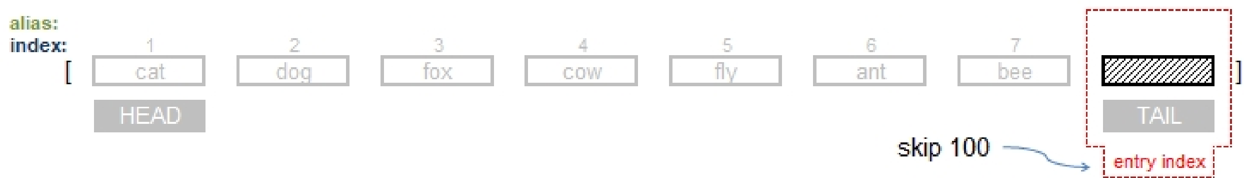
P emístí **entry index** o daný po et element k chvostu.



```
>> 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 po et element , z stane **entry index** ve chvostu.

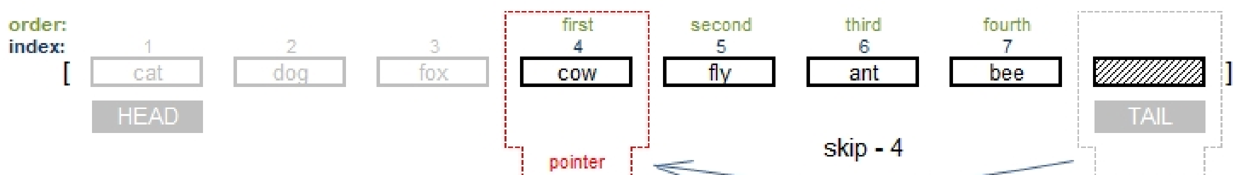
```
>> s: skip s 100
== []
```




```
>> tail? s
== true

>> index? s
== 8
```

Viditelnost element ady lze obnovit zp tnými 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

Series "getters"

Existuje takové množství p íkaz pro manipulaci s adami (series)se, že jsem je rozd lil do dvou kapitol - jednu pro p íkazy získávající informace z ad (getters) a druhou pro p íkazy, které p ímo m ní ady (changers).

P íkazy "getters" pouze vracejí hodnoty, bez zm ny ad. Nicmén   i tyto p íkazy mohou m nit ady, pokud p i adíme adu k vratné hodnot .

action! length?

Vrací velikost (délku) ady 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 ada prázdná, jinak vrací `false`.

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

>> empty? a
== false

>> b:[]
== []
```

```
>> empty? b
== true
```

action! pick

Vybere hodnotu z ady na pozici, dané druhým argumentem.

pick [0 1 2 3 4 5] 4 == 3

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

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

action! at

Vrací adu od zadaného indexu.

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

action! select a action! find

Oba p íkazy vyhledají zadanou hodnotu v ad . Hledání probíhá zleva doprava, není-li použito up esn ní `/reverse` nebo `/last`.

P i nalezení shody:

- `select` vrací následující element ady za shodou;

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

- `find` vrací celý zbytek ady od shody ke chvostu.

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

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

/part

Omezí délku prohledávané ady na daný po et element ; v zobrazení dole je prohledávaná oblast zvýrazn na:

```
>>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 uvnit prohledávané ady.

```
>> 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 [ada skupin] [shoda] [délka skupiny]

Považuje adu za sled skupin zadané délky. Požadovaná shoda se musí nalézat na prvním míst 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 ozna eny žlut a hledaná shoda mod e. Vrací zbytek bloku v etn shody.

/last

Vyhledá poslední shodu se zadaným elementem; vrátí zbytek ady.

```
>> 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

Normáln vrací `find` výsledek v etn shodujícího se elementu. S up esn ním `/tail` se vrací zbytek ady za shodným elementem, podobn 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

Up esn ní **/match** porovnává klí s po átkem ady. P ípadnou vracenou hodnotou je zbytek ady 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 ady.

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

function! extract

Extrahue hodnoty z ady v zadaném intervalu, vracejíc novou adu.

```
>> 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

P ipojí extrahované hodoty k zadané ad   (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.

native! **union**

Vrací spojení dvou ad. Duplikátní elementy jsou za azeny 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 ady za sled skupin zadané délky. Porovnává se pouze první element každé skupiny. U duplicitních vstup   se zachovává záznam první ady:

```
>> 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 element   z obou ad.

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

/case

Case-sensitive porovnávání

/skip

Považuje ady za sled skupin zadané délky.

native! intersect

Vrací pouze duplicitní výskyty element :

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

/case

Case-sensitive porovnávání

/skip

Považuje ady 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 ady za sled skupin zadané délky

native! exclude

Vypouští zadané elementy z ady a vrací seznam neopakovaných element .

```
>> 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 ady za sled skupin zadané délky

Created with the Standard Edition of HelpNDoc: Create help files for the Qt Help Framework

Series "changers"

Tyto p íkazy m ní p vodní ady !

action! clear

Smaže všechny elementy ady.

Pouhé p i azení prázdného et zce (" ") nebo nuly nemusí mít o ekávaný ú inek. Pamatování "v cí" realizuje Red mnohdy neo ekávanými zp soby. Skute né smazání ady zajistí p íkaz `clear`.

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

>> clear a
== []

>> a
== []
```

action! poke

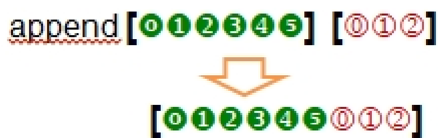
Zm ní hodnotu elementu ady v pozici dané druhým argumentem na hodnotu t etího argumentu.

```
>> 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 ady.




```
>> 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 po et p ipojovaných element .

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

/only

P ipojí p ipojovanou adu jako vno ený blok.

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

/dup

P ipojí 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 míst aktuálního entry-indexu (obvykle to je po átek ady). Zatímco `append` vrací adu od ela, `insert` ji vrací až za místem vložení. To umož uje et zit vícero operací `insert`, nebo po ítat délku vložené ásti.

insert [⓿❶❷❸❹❺] [⓪①②]

[⓪①②⓿❶❷❸❹❺]

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

>> insert a "000"
== "abcdefgh"

>> a
== "000abcdefgh"
```

insert at [⓪①②③④⑤] 3 [⓪①②]

[⓪①⓪①②②③④⑤]

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

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

>> a
== "abOOOcdefgh"
```

/part

Vloží pouze zadaný po et vkládaných element .

/only

Umož uje vložení element   jako blok.

/dup

Umož uje opakované vložení druhého argumentu v po tu, daném t etím 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 [⓪①②③④⑤] [③] [①]

[⓪①②①④⑤]

```
>> 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

T ídí adu podle velikosti hodnot.

sort [②④①③⓪] == [⓪①②③④]

```
>> 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"
== " eeggiiilnnorrssssssttu"
```

/case

Provede case-sensitive t íd ní.

/skip

Zachází s adou jako s po adím 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

T ídí pouze ásti ad

/all

Porovná všechna vybraná pole. Používá se s up esn ním 'skip' pro ur ení velikosti skupin.

/reverse

Obrátit po adí

/stable

Místo n kdy nestabilního t íd ní Quicksort se použije stabiln jší ale pomalejší algoritmus Merge.

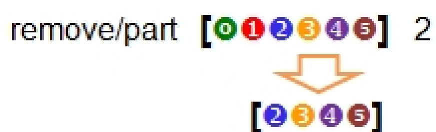
action! remove

Odebere první hodnotu ady.

```
>> 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 po et element .




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

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

Totéž lze provést p íkazem `remove at [0 1 2 3 4 5] 2`

native! remove-each

Podobn jako `foreach`, postupn vyhodnotí blok pro každý element ady. Vrací-li blok hodnotu `true`, je element odebrán z ady:

```
Red []

a: ["dog" 23 3.5 "house" 45]
remove-each i a [string? i]                    ; odebere všechny  et zce
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 ady 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 ady 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 po et element   z po átku ady 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 vno ené hodnoty.

action! move

P emístí jeden (implicitn) nebo více element   ady na jinou pozici téže nebo jiné ady.
M ní oba vstupní argumenty.

move [⓿❶❷❸❹❺] [⓪①②③④⑤]

[❶❷❸❹❺] [⓿⓪①②③④⑤]

/part

P emístí zadaný po et element

move/part [⓿❶❷❸❹❺] [⓪①②③④⑤] 3

[❸❹❺] [⓿❶❷⓪①②③④⑤]

```
>> 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]
```

P íkaz move m že být kombinován s jinými p íkazy pro p emíst ní uvnit jediné ady. Na p íklad:

```
>> 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

Zm ní po áte ní elementy ady a vrací adu po zm n . M ní pouze první adu.

change [⓿❶❷❸❹❺] [⓪①②]

[⓪①②❸❹❺] [⓪①②]

```
>> 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 m n nou délku ady vložením hodnot z jiné ady.

```
>> 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 po et duplikát zm ny.

function! alter

P ipojí element k ad nebo jej z ady odebere. Pokud `alter` zadaný element v ad nenalezne p ipojí 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 ad. Vrací první adu ale m ní ob :

swap [0 1 2 3 4 5] [0 1 2]

[0 1 2 3 4 5] [0 1 2]

```
>> 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 ad nebo elementy téže ady:

```
>> 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

P evrátí po adí element v ad :

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

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

/part omezí akci reverse na zadaný po et element :

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

Created with the Standard Edition of HelpNDoc: Full-featured Kindle eBooks generator

P ístup a formátování dat

native! get

Každé slovo Redu, nativní i uživatelsky vytvo ené, je umíst no ve slovníku. Je-li slovo sdruženo s výrazem, je uloženo i s p idruženým výrazem, který se vyhodnotí v závislosti na typu volání slova.

Chcete-li znát slovníkový popis slova, použijete p íkaz `get` . Uv domte si, že když v Redu odkazujete na slovo samotné (nikoliv jeho hodnotu), p edznamenáte slovo apostrofem ('). P íkaz `get` poskytuje význam i nativních slov ale vrací chybové hlášení, je-li použit pro konkretní hodnotu, nap . 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

P íkaz `mold` p em ní zadanou hodnotu (nap . typu `block!`, `integer!`, `series!` etc.) na et zec a **vrátí jej**:

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

>> type? mold 8
== string!

>> print [4 + 2]
6

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

Refinements:

/only - Vylou í vn jší hranaté závorky bloku

/all - Vrací hodnotu v na tení schopném formátu

/flat - Vylou í veškerou identaci

/part - Omezí délku výsledku na zadanou hodnotu

action! form

P íkaz `form` také p em ní zadanou hodnotu na et zec ale v závislosti na typu, nemusí výsledný text obsahovat extra informace o typu (jako [ ] { } a ""), jako u p íkazu `mold`.

Užite né pro Manipulace s textem.

```
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
```

Umož uje použití refinementu `/part` k omezení po tu znak .

Hlavní použití pro mold a form:

`mold` se hlavn používá pro p em nu ad na kód, který m že být uložen a interpretován pozd ji

`form` se hlavn používá pro generování oby ejného textu z ad

```
>> 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

P íkaz `probe` vytiskne sv j argument bez vyhodnocení a také jej vrátí. Lze jej s výhodou

použít p i lad ní. Vzpome te si, že p íkaz `print` sv j argument vyhodnocuje.

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

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

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

native! reduce

Vyhodnocuje výrazy uvnit bloku a vrací nový blok s vyhodnocenými hodnotami. Nahlédn te do kapitoly Vyhodnocení.

```
>> 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 p ipraveného bloku.

function! collect a keep

Shromáždí v novém bloku všechny hodnoty vybrané funkci keep z t la bloku.
Jinými slovy: vytvo í nový blok, zachovávaje pouze hodnoty ur ené funkcí keep, obvykle na základ spln ní ur ité 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 vytvo en funkcí collect
```

```
house dog
```

syntaxe pro up esn ní: **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 rozší en o vybrané prvky bloku a
```

```
one two 11 34.2 22
```

native! compose

Vrací kopii bloku, vyhodnocujíce pouze prvky typu paren!.
P íkaz compose je velmi d ležitý pro dialekt DRAW;

```
Red []

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

print a                    ; vyhodnotí vše, v etn  zano ených závorek
probe compose a    ; vrací argument s vyhodnocenými závorkami typu paren!
```

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

/deep => provést p íkaz compose i pro zano ené 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) vno ené bloky jako bloky, obsahující své hodnoty nemtudum

je zapot ebí vložit p íklad

/into - vložit výsledek do p ipraveného bloku místo vytvá ní 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

Matematika a logika

V tšina matematiky a logiky v Redu je b žná až na po adí výpo tu. Níže uvádím vý et operátor (slov), používaných pro výpo ty spolu s p ípadnými poznámkami.

Základy:

Následující skupinu tvo í jak **prefixové** (nap . add) tak **infixové** (nap . operátor "+") funkce . P ijímají hodnoty typu number!, char!, pair!, tuple! nebo vector! jako argumenty.

Funkcionální (prefixové) operátory se uvád jí p ed svými operandy (nap .: add 3 4).

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

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

action! **multiply** or op! *

action! **divide** or op! /

action! **power** or op! **

action! **absolute**

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

action! **negate**

Zm ní 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 íslu, vrací integer mezi 1 (inclusive) a argumentem (inclusive).

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

Je-li argumentem ada, p eskupí 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
```

Up esn ní (refinements):

/seed - Restartovat a randomizovat. Patrn lze použít v situaci, kdy je funkce random volána v programu opakovan . V tom p ípad nemusí být výsledek zcela náhodný, pokud se nepoužije random/seed.

/secure - TBD: Vrací kryptograficky zajišt né náhodné íslo.

/only - Vybrat náhodnou hodnotu z ady.

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

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

action! round

Vrací nejbližší celé íslo. Poloviny (nap . 0,5) se zaokrouhlují sm rem 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 ur ení "p esnosti" 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 (nap . 0.5) se zaokrouhlují sm rem k sudému íslu.

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

/down - Odsekne desetinnou ást ale íslo zachová jako float!.

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

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

/half-down - Poloviny zaokrouhlovat sm rem k nule, nikoli od nuly.

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

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

/floor - Zaokrouhlovat v negativním sm ru.

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

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

/ceiling - Zaokrouhlovat v pozitivním sm ru.

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

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

/half-ceiling - Zaokrouhluje poloviny v pozitivním sm ru.

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

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

native! **square-root**

P ijímá libovolnou hodnotu typu `number!` jako argument.

Remainders (zbytky) etc.:

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

P ijímá hodnotu typu number! char! pair! tuple! a vector! jako argument. Vrací zbytek d lení prvního ísla 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 d lení jedné hodnoty druhou.

function! **modulo**

Vrací pozitivní zbytek (modulus) po d lení prvního argumentu druhým. Je-li n které z ísel

(i ob) záporné, uvedené jednoduché pravidlo neplatí a je to mnohem složit jší - viz Vikipedia: 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 p irozeného ísla **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) p ijímají

argumenty ve stupních, umož ují ale použití argumentu v radiánech s up esn ním `/radians`. Verze s krátkými názvy (acos, cos etc.) p ijímají úhly v radiánech.

function! acos či native! arccosine

function! asin či native! arcsine

function! atan či native! arctangent

Vrací trigonometrický arctangent.

function! atan2 či native! arctangent2

Vrací úhel spojnice bodu 0,0 a x,y v radiánech, m ený proti sm ru hodinových ru i ek od kladné osy jednotkové kružnice. Vracené hodnoty se pohybují mezi -pi a +pi.

function! cos či native! cosine

function! sin či native! sine

function! tan či native! tangent

Extras:

native! max

Vrátí v tší ze dvou argument typu scalar! nebo series!

```
>> max 8 12
== 12
```

```
>> max "abd" "abcd"
== "abd"
```

P i porovnávání et zc (což jsou series!) nebo blok je postupn porovnáván každý element. P i výskytu první neshody porovnávání kon í a jako v tší je vybrán blok (string) s v tší 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]
```

P i porovnání hodnot typu pair!, vrací v tší z každého elementu:

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

native! min

Vrací menší ze dvou argument . Poznámka o `max` je zde rovn ž 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 v tší než nula a `false` je-li menší než 1.

native! negative?

Vrací `true` je-li argument menší než 1 a `false` je-li v tší než 0.

native! zero?

Vrací `true` je-li argument roven nule.

function! math

Vyhodnotí hodnotu typu block! s použitím normálních precedencí, to jest s up ednostn ním d lení a násobení p ed s ítáním a od ítáním.

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

function! within?

P ijímá 3 argumenty typu pair! - pozici bodu x,y - sou adnice x,y levého horního bodu obdélníka - ší ku a výšku obdélníka.

Vrací `true!`, nalézá-li se bod uvnit 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~ či op! and (infix)

native! equal? či op! =

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? či op! <>

action! or~ či op! 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 pam ti.

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

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

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

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

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

native! strict-equal? or op! ==

Vrací `true!` mají-li argumenty stejnou hodnotu a jsou stejného typu i velikosti (lower-case/uppercase) u et zc .

```
>> 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

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 - ur it po et hexadecimálních íslic ve výsledku

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

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

native! enbase a native! debase,

Používají se pro kódování a dekódování binárních et zc .
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; m že 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

Kryptografie

native! checksum

Po ítá 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

Kopírování

Varování pro za áte níky: P í azujete-li hodnotu slova jinému slovu , prove te to s p íkazem COPY.

```
>> var1: var2          ; jste-li si jisti, co činíte

>> var1: copy var2     ; může vám ušetřit hodiny ladění
```

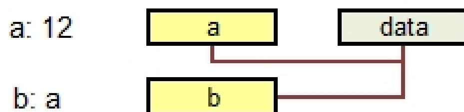
action! copy

P i adí kopii dat jinému slovu.

Lze použít ke kopírování ad a objekt .

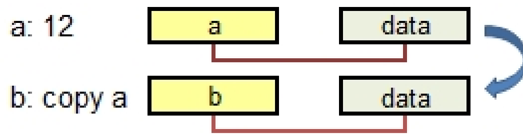
Nepoužívá se pro jednotlivé položky typu: integer! float! char! etc. Pro tyto ú ely m žeme jednoduše použít dvojte ku.

Pohle me na jednoduché p i azení (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 p i azení 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 vno ených ad (blok). P íkaz `copy` nem ní odkaz na tyto vno ené ady. Chcete-li pro tento p ípad vytvo it nezávislou kopii, musíte použít up esn ní `/deep`.

copy/part

Omezuje délku (`number!` i `series!`) kopírované ásti.

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

copy/types

Kopíruje jenom ur ité typy neskalárních hodnot.

copy/deep

Kopíruje vno ené hodnoty.

Created with the Standard Edition of HelpNDoc: Write eBooks for the Kindle

Opakování

native! loop

Provede blok opakovan v zadaném po tu.

Red[ ]

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

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

native! repeat

`repeat` je totéž jako 'loop', obsahuje však po ítadlo (index), jehož hodnota se s každou smy kou zv tšuje 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 ady 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 spln na zadaná podmínka.

```
Red[]

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

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

native! until

Prvádí blok, pokud není pln na vstupní podmínka.

```
Red[]

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

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

native! break

Ukon í provád ní smy ky p i nespln ní zadané podmínky.

/return

Totéž, navíc však vrátí zadaný text pro zjišt ní, kdy nebyla podmínka spln na.

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

```
1
2
4
<finito>
```

native! forever

Vytvo í nikdy nekon ící smy ku.

Created with the Standard Edition of HelpNDoc: Easily create Help documents

Podmínky

native! if

Provede blok p i spln ní zadané podmínky.

if <test> [block **]**

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

native! unless

Totéž jako `if not`. Provede blok pouze p i nespln ní 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 opa ném p ípad provede druhý blok.

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

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

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

native! switch

Provede blok, odpovídající zadané hodnot .

```
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 adu 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 vytvo ení složit jší ídící struktury. Ve sve nejjednodušší form 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 ozna it jako `true`. P ípadn   vrací `none`, je-li n který výraz hodnocen jako `false`.

```
all [
    33
    5 >2
    8
    12
    ]    ==> returns 12
```

```
all [
    33
    5 < 2   false ==> returns none
    8
    2 = 3
    ]
```

```
>> 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 postupn   výrazy v bloku a vrací první hodnotu, která není `false`. Jsou-li všechny vyhodnocené hodnoty `false` , vrací `none`.

any [
3 = 5
5 < 2
8 ==> returns 8
12
]

any [
3 = 5
5 < 2
9 = 3
2 = 3
] ==> returns 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

Manipulace s textem

---

function! split

Vytvo í blok, sestavený z ástí et zce, rozd leného zadaným odd lova em. P íkaz `split` je obzvlášt vhodný pro analýzu a manipulaci s textovými 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 znak : action! trim

Slovo `trim` (zredukovat) bez up esn ní odebírá bílá místa (taby a mezery) ze za átku a konce et zce (také odebírá `none` z entity typu block! nebo object!). Hodnota argumentu se zm ní.

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

trim/head - odebírá mezery pouze z   ela

```
>> 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 p erušení ádku a nadbyte né mezery

trim/all - odebere všechny mezery, nikoliv ale p erušení ádku

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

P íkaz pad rozší í et zec na danou velikost p idáním mezer. Implicitn se mezery p idávají vpravo ale s up esn ním /left se mezery p idávají vlevo. Pozor, m ní p vodní 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"
```

p ípadn s p íkazem append, což m ní p vodní ady:

```
>> 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
```

P em na ady na text: action! form

P íkaz `form` p em ní adu na string odebráním hranatých závorek a p idáním 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

Up esn ní `/part` limituje po et znak ve vytvo eném et zci.

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

Zjišt ní délky et zce: action! length?

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

P ehled manipulací

Levá strana et zce:

použitím **copy/part** :

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

Pravá strana et zce:

použitím `at` :

```
>> s: "nasty thing"
```

```
>> at tail s -5
== "thing"
```

použitím `remove/part` - pozor, m ní p vodní et zec!

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

St ední   ást   et zce:

použitím `copy/part` a `at`:

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

Vkládání   et zc :

na za átek, s použitím `insert`:

```
>> s: "house"
>> insert s "beautiful "

>> s
== "beautiful house"
```

na konec, s použitím `append`:

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

ve st ední   ásti, s použitím `insert at`:

```
>> s: "nasty thing"
>> insert at s 7 "little "

>> s
```

```
== "nasty little thing"
```

native! lowercase

Pozor, m ní p vodní et zec.

```
>> 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

Práce s asem

native! wait

Zastaví exekuci na zadaný po et vte in.

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

native! now

Vrací datum a as:

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

now/time - Vrací pouze as (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 vytvo í asový 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 m síc

```
>> now/month
== 4
```

now/day - Vrací pouze den m síce

```
>> now/day
== 2
```

now/zone - Vrací asový 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! (pond lí je den 1).

```
>> now/weekday
== 1
```

now/yearday - Vrací den roku (Julian).

```
>> now/yearday
== 92
```

now/utc - Univerzální as (no zone) (date!)

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

VID DLS rate

Nastavení asu lze také provést v dialektu VID s použitím aspektu `rate`.

Created with the Standard Edition of HelpNDoc: Free HTML Help documentation generator

Ošet ení chyb

function! attempt

Vyhodnotí blok a vrací výsledek nebo vrací `none` p i 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 p i výskytu chyby vrací hodnotu `error!`.

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

M žete se ptát pro   nepoužít `attempt` místo `error?` & `try`. Domnívám se že proto, že kombinace `error?` & `try` vrací `true` a `false` místo `none` nebo provedeného vyhodnocení. To je užite né p i použití uvnit 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** and native! **throw**

Tyto p íkazy se používají k ošet ení chyb. Delší vysv tlení lze nalézt zde.

Created with the Standard Edition of HelpNDoc: Easily create PDF Help documents

Soubory

Názvy cest

Cesty k soubor m se píší se znakem %, následovaným vý tem adresá   , odd lených lomítkem. Ve Windows si Red sám m ní zp tná lomítka na normální lomítka.

Jenom pro p ipomn ní:

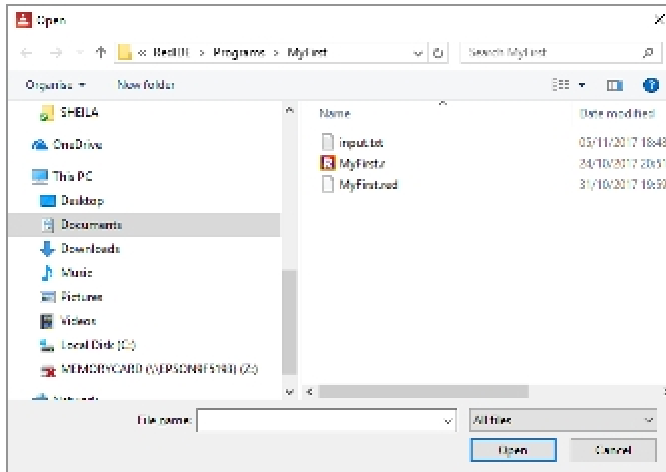
- `  /` písmeno p ed prvním lomítkem ozna uje ko enový adresá oddílu
- `./` ozna uje aktuální adresá
- `../` ozna uje rodi e aktuálního adresá e
- cesty, které neza ínají odkazem na ko enový adresá oddílu, jsou relativní
- absolutní cesta pro   asto používaný oddíl "C" ve Windows: `%/C/docs/file.txt`
- absolutní cesty nejsou strojov   nezávislé;

Grafický selektor souboru:

function! request-file

P íkaz `request-file` otev e grafické okno Pr zkumníka 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 -

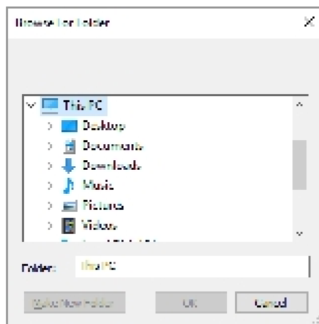
request-file/filter - Poskytnutý blok filtr ovlivní konfiguraci otev eného výb rového okna. Nap íklad, `request-file/filter ["executables" "*.exe" "text files" "*.txt"]` vytvo í roletku s nabídkou 'executables' a 'text files'.

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

Grafický selektor adresá e:

function! request-dir

P íkaz `request-dir` otev e okno se základní nabídkou adresá (User, Tento po íta , Knihovny, Sí). Pro vybraný soubor i 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 zada ilo, jinak vrací `false`.

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

Zjišt ní velikosti souboru:

native! size?

Vrací velikost v bajtech nebo `none`, když soubor neexistuje.

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

Další funkce:

cd nebo **change-dir -** m ní aktuální adresá .

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

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

dirize - p em ní argument na platný adresá ; argumentem m že 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 adresá , používaný programem

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

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

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

suffix? - vrací p íponu souboru, nap . exe, txt

what-dir - vrací cestu k aktuálnímu adresá i 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 cest a vrací vy išt nou 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

Psaní do souboru

Psaní do textového souboru:

action! write

Provede zápis do souboru, který vytvo í, pokud neexistuje.

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

P ipojení a manipulace s textovým souborem:

write/append

Op tovný zápis do souboru zp sobí vymazání p vodního obsahu. Chcete-li jenom p ipojit (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 ad na vlastní ádky:

P idejme nyní další soubor se t emi ádky:

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

P idávání dalších ádk :

```
>> 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.
```

V zte, že jste mohli psát write/lines/append. Po adí up esn í nemá na nic vliv.

Vým na znak   v souboru:

Pro vým nu znak , po ínaje 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."

Up esn ní

/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,   i jiná data do souboru, URL, bindruhu (binary) nebo et zce.

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 p epsání) obsahuje vý et: `11 22 "three" "four" "five"`

D ležité použití p íkazu `save` je p i ukládání skript Redu, které mohou být interpretovány s použitím akce `do` :

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

P íkazy `do`, `load` a `save` se lépe chápou, považujete-li konzolu Redu za monitor starého po íta e z 80. let, realizující n kterou variaci základního jazyka. M žete program `load` (na íst), `save` (uložit) nebo `do` (provést).

Created with the Standard Edition of HelpNDoc: Generate Kindle eBooks with ease

tení ze souboru

Na tení celého souboru do jednoho et zce:

action! read

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

Nyní prom nná "a" obsahuje celý obsah souboru:

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

Na tení souboru do bloku po jednotlivých ádcích:

Chcete-li íst soubor jako adu, v níž je každý ádek 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 => te pouze zadaný po et jednotek (source relative).

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

read/binary => p esn zachovává obsah

read/lines => p em ní na blok et zc

read/info =>

read/as => te se zadaným kódováním, implicitn je 'UTF-8.

function! load

Na tení souboru do bloku po jednotlivých slovech, odd lených mezerou:

Pro tento ú el použijeme `load` místo `read`:

```
>> a: load %mySecondFile.txt
== [First line Second line Third line.
          Fourth line Fifth...

>> print pick a 2
line
```

tení a psaní binárních soubor :

Pro tení i psaní binárního souboru jako je image nebo zvuk, se používá up esn ní `/binary`. Následující kód na te bitmapové zobrazení do prom nné a zapíše jej do souboru s jiným názvem:

```
>> a: read/binary %heart.bmp
== #{
424D66070000000000003600000028000001E00000014000000010...
>> write/binary %newheart.bmp a
```

load/header => TBD.

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

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

load/next => na te pouze následující hodnotu.

load/part =>

load/into => vloží obsah do existujícího bloku

load/as => zadání typu na ítaných dat; použijeme NONE pro na tení jako kód

Created with the Standard Edition of HelpNDoc: Write eBooks for the Kindle

Funkce

Funkce musejí být p ed použitím deklarovány a proto se píší v p ední ásti 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
```

Prom nné uvnit funkce, vytvo ené 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

Prom nné uvnit funkce se slovem `function` jsou **lokální**, to jest, jsou p ístupné pouze uvnit definované funkce.

```
Red []
mysum: function [a b] [
      mynumber: a + b
      print mynumber
]
mynumber: 20
mysum 3 4
print mynumber
```

R zné výsledky:

```
7
20
```

Vnucená globální p ístupnost prom nných up esn ním /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 p edem ur it :

```
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 p ipustit více datových typ :

```
Red []
mysum: function [a [integer! float!] b[integer!]] [print a + b]
print mysum 3.2 4
```

```
7.2
```

P ípadn   lze zadat nad ízený 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 bu poslední funkcí vyhodnocená hodnota nebo hodnota explicitn ur ená slovem `return`:

```
Red []
myfunc: function [] [
       8 + 9
       3 + 3
       print "got here"     ; this executes
       10 + 5                    ; this is returned
]
print myfunc
```

```
got here
15
```

P íklad s 'return'; slovo `return` je poslední aktivní p íkaz funkce:

```
Red []
myfunc: function [] [
       8 + 9
       return 3 + 3                   ; this is returned
       print "never got here"  ; NOT executed
       10 + 5
]
print myfunc
```

```
6
```

Vytvá ení vlastních up esn ní:

Je možné vytvá et vlastní up esn ní funkcí, podobná nativním up esn ním Redu: Jednotlivá up esn ní jsou bulíny (boolean values), ov   ované 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 up esn ní nejsou argumenty povinné.

P i azení funkcí ke slov m (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

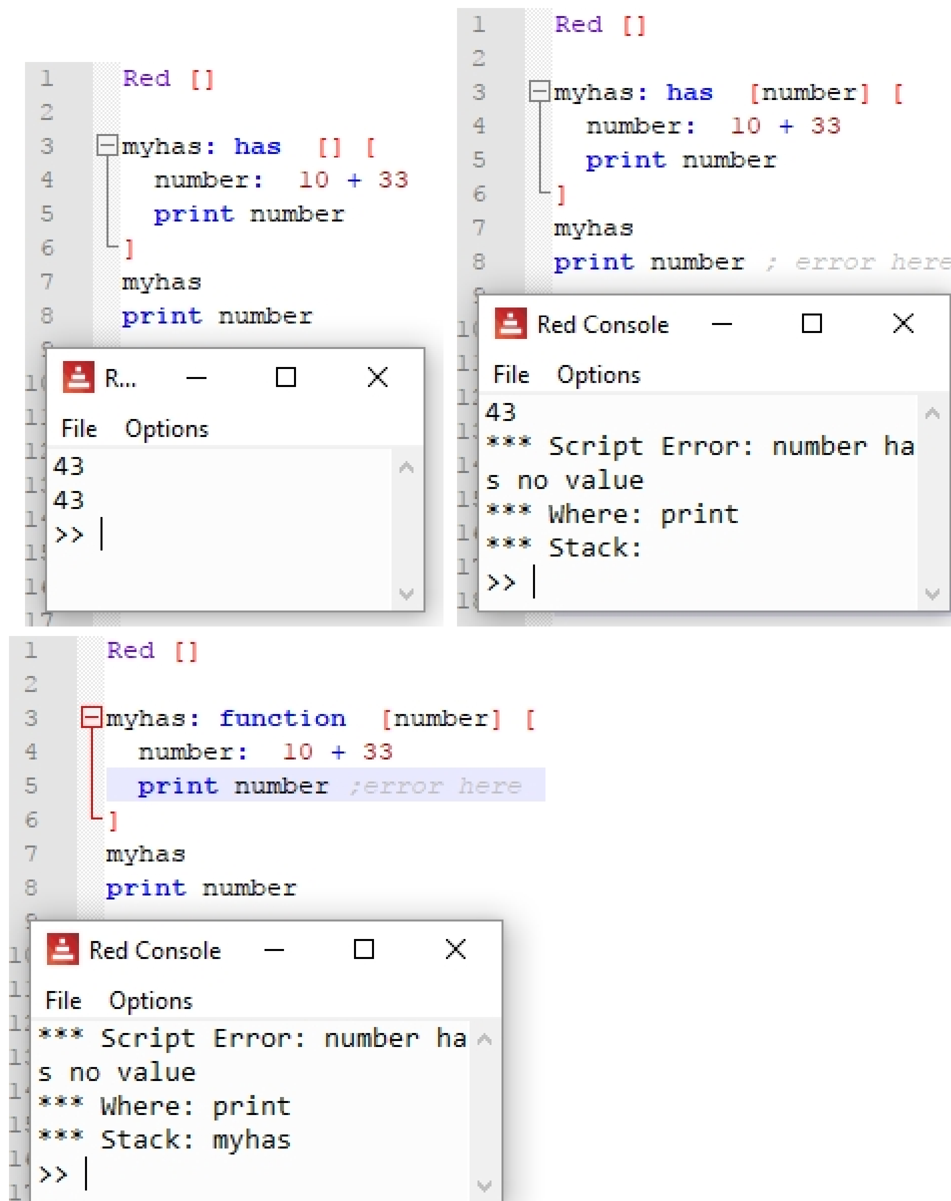
Pokud naše rutina jenom n co provádí - bez **argument** a **lokálních prom nných**, použijeme slovo `does` :

```
Red []
greeting: does [
      print "Hello"
      print "Stranger"
]

greeting
```

```
Hello
Stranger
```

native! has

Pokud naše rutina nepoužívá žádné externí argumenty ale má lokální prom nné, použijeme slovo `has`. To p em ní p ípadný argument na lokální prom nnou. Porovnejte níže uvedené t i programy. První používá `has` bez argumentu, slovo `number` je tedy globální prom nná. Druhá obsahuje `number` jako argument, jenž je však uvnit funkce redefinovaný na lokální prom nnou. Pokud zm níme název tohoto argumentu, je funkcí ignorován, nebo argument není u tohoto typu funkce povinný. V t etí ukázce vidíme, že v definici funkce musí být argument reáln použit, nikoliv redefinován a že funkce musí být volána s hodnotou argumentu.



native! **exit**

Opustí funkci bez vracení jakékoli hodnoty.

R zné typy vestav ných funkcí

- **mezaninové** funkce jsou vytvo eny v jazyce Red a jejich zdrojový kód lze íst s použitím p íkazu `source <název fce>`.

Výpisy funkcí následujících typ lze získat dotazem (help i ?) na název typu:

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

Created with the Standard Edition of HelpNDoc: Free EBook and documentation generator

Objekty

Objekt je kontejnér, který sdružuje data i funkce a je (tém vždy) p i azený ke slovu (prom nné). Pro p ístup k atributu objektu používáme lomítko (neboli formát podobný cest). Tímto se Red liší od v tšiny jazyk , které v podobných situacích používají te ku.

Vytvo ení objektu:

action! make object! , function! context a function! object

K vytvo ení objektu použijeme p íkaz `make object!` nebo jeho kratší varianty `object` p ípadn `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 p i deklaraci objektu (constructor code). Zde nap íklad p íkaz 'print' je posléze nep ístupný.:

```
>> 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 klí ové 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 d d ní:

Pouhé p i azení objektu k jinému jménu vytvá í pouhou kopii **ukazovátka** (pointer) na tento objekt. Zm ní-li se originál, zm ní 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 vytvo ení 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 vytvo it nový objekt, který d dí z jiného objektu, použijeme sekvenci: `make zdroj [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
```

P ípustný formát zadávaného pole je lit-word! ('word). Hodnotu prom nné lze evokovat p íkazem `obj/a`.

Created with the Standard Edition of HelpNDoc: Easily create Web Help sites

Reaktivní programování

Reaktivní programování je p kn popsané v oficiální dokumentaci.

Reaktivní programování vytvá í interní mechanizmus, který automaticky aktualizuje stav objektu A p i zm n objektu B - bez volání funkcí i subrutin.

Red používá objektov orientovaný reaktivní model push.

Reaktor je reaktivní objekt, jehož zm na spustí zm ny v jiných objektech. Vytvá í se p íkazem `make reactor!`.

Reaktivní vztah (statický nebo dynamický) je deklarace vztahu mezi p íslušnými poli reaktivn propojených objekt . Vytvá í se funkcemi `is` nebo `react`.

P íkaz **make reactor!** a funkce **is**

Velmi bazální p íklad 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, vytvá í nekone nou smy ku!
```

```
?house
house
?fly
fly
?bee
bee
```

Reaktor m že 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, vytvá í nekone nou smy ku!
```

```
?33
35
?45
47
```

deep-reactor!

Stejn jako má `copy` up esn ní `/deep` pro p ístup k zano eným hodnotám (bloky uvnit 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 p íkaz pro vytvá ení reaktivních grafických aplikací GUIs. Nahlédn te prosím do GUI-Pokro ilá témata.

function! clear-reactions

Odstraní bezpodmíne n 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`. Up esn ní `/target` kontroluje, zda je pole cílem místo zdrojem a vrátí první reakci, zacílenou na toto pole nebo vrátí `none` p i absenci shody.

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

function! dump-reactions

Vytvo í seznam registrovaných reakcí pro ladící ú ely.

Created with the Standard Edition of HelpNDoc: News and information about help authoring tools and software

Rozhraní OS

---

native! call

Provádí p íkazy z shellu. V tšinou je to totéž jako z p íkazového ádku systémové konzoly (CLI) ale existuje n kolik úchylek.

Následující kód otev e Windows Explorer:

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

Toto rovn ž chodí:

```
>> str: "explorer.exe"
== "explorer.exe"

>> call str
== 11916
```

Ovšem, následující kód vytvo í proces ale neotev e Notepad na obrazovce:

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

Chcete-li chování více podobné zápisu p íkazu v shellu, musíte použít up esn ní `/shell`:

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

Další up esn ní:

call/wait

Spustí p íkaz a eká na jeho ukon ení. Bu te opatrní, použijete-li up esn ní /wait pro

proceduru, kterou nebudete um t ukon it (jako call "notepad.exe" naho e), Red bude ekat... a   ekat.. až do nekone na.

call/input - zadáváme string!, file! nebo binary!, jež bude p esm rováno 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 p ijme p esm rovaný stdout z p íkazu. Všimn te si, že výstup je p ipojen.

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

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

Toto vytvo í (dlouhý) et zec 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 p íkaz s I/O p esm rovaný do konzoly (CLI console only at present, does not work with Red's normal GUI console).

native! write-clipboard & read-clipboard

Zapisuje do a   te 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

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

GUI - P ehled

V následujících kapitolách budou jednotlivé elementy a parametry grafického rozhraní (**GUI**) - kontejnery (**containers**), dispozice (**layouts**), piškoty (**faces**) a aspekty (**facets**) popsány zevrubn . Pro orientaci a v domí vzájemné souvislosti je vhodné, uvést n kolik základních informací p edem.

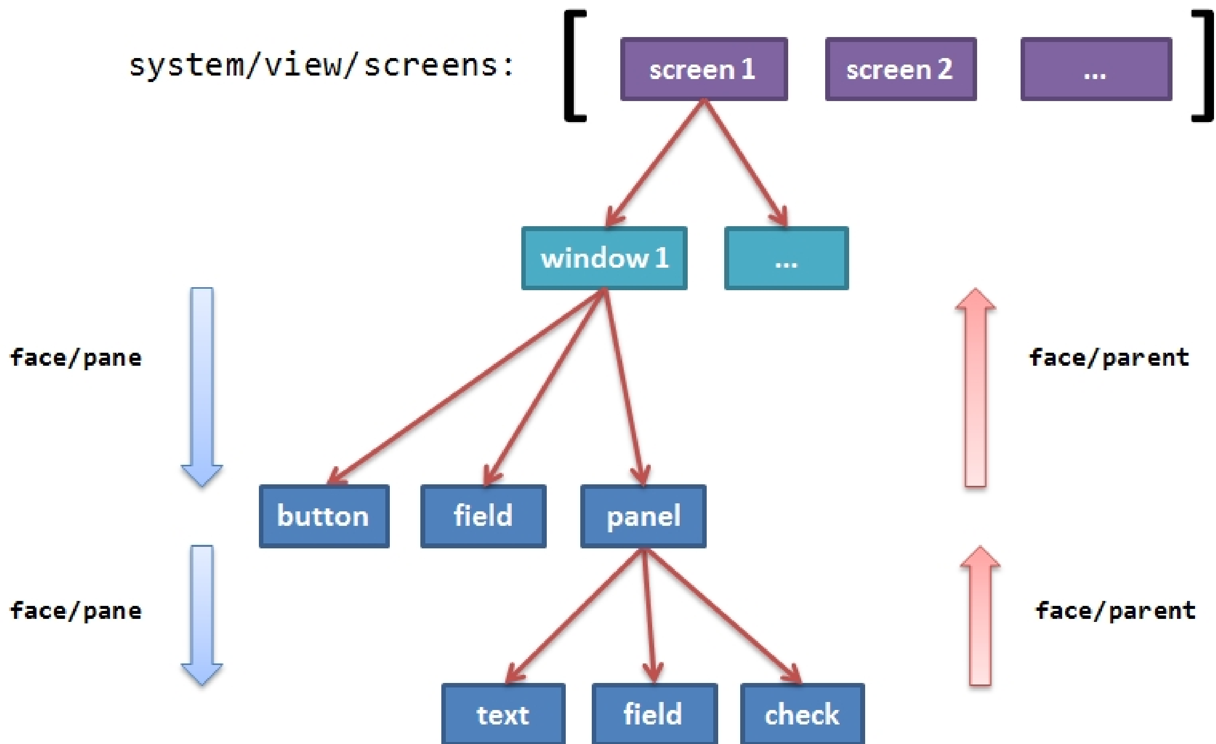
V následujícím schematu uvedené elementy screen, window, button, ..., jsou všechno **piškoty** (faces), kterých je aktuáln 18 (area, base, button, camera, check, drop-down, drop-list, field, group-box, panel, tab-panel, progress, radio, slider, screen, text, text-list, window).

Piškot **screen** zastupuje obrazovku po íta e a m že obsahovat jeden i více piškot window. Piškot **window** (okno) je kontejnerem pro další i zano ené piškoty. Tyto dva elementy mají dominantní postavení; od ostatních piškot se liší zejména tím, že si je interpret Redu vytvo í automaticky p i aplikaci n kterého z dalších deklarovaných piškot .

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

Aplikací se míní realizace p íkazu **view** na základ intern vytvo eného stromu piškot .

Vzájemnou hierarchii piškot vyjad ují aspekty **pane** (zeznam d tí, na které piškot ukazuje) a **parent** (odkaz na rodi ovský piškotq). Aspekt je aktuáln 23, vyjad ují vlastnosti jednotlivých piškot a jsou nedílnou sou ástí každého piškotu.



P knou prezentaci vztah mezi piškoty screen, window a button získáme realizací tohoto kódu:

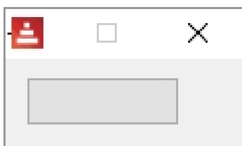
```
>> view [b: button [print b]]
```

p ípadn

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

p ípadn

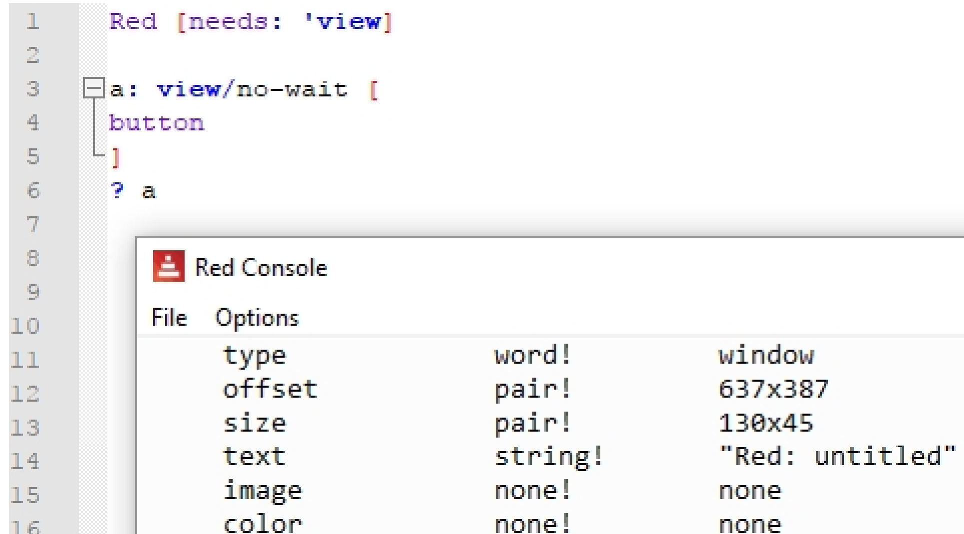
```
>> view [button [probe face]]
```



Stiskem levého tla ítka myši s kurzorem v tmavošedém poli spustíme provedení p íkazu (`print b`, p ípadn `probe face`). V okn interaktivní konzoly Redu se vytiskne výpis všech aspekt s p i azenými hodnotami u všech t í použitých piškot (screen, window a button). Zano ení piškot je deklarováno v aspektu parent a pane. **V ele doporu uji si tento postup vyzkoušet.**

Výpis aspekt v konzole je dvakrát zano ený. Piškot typu button odkazuje svým aspektem parent na piškot windows a ten zase na screen. Takto jsou jednotlivé komponenty grafického objektu vzájemn propojeny.

Výpis aspekt pro jednotlivé piškoty (krom screen a window) lze také získat realizací tohoto kódu:



Zadáte-li p íkaz ? a samostatn v následn otev ené konzole, kterou si p edtím roztáhnete na celou ší ku obrazovky, bude váš výpis p ehledn jší.

Jednoduchý za átek

Red vytvá í grafické prost edí (GUI) tak, že jej popíše v bloku **view**. Tento popis je velmi p ímo arý a ve své nejjednodušší form by mohl vypadat takto:

view [

widget (face)

widget (face)

widget (face)

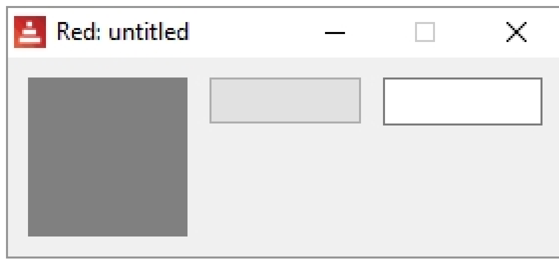
]

Má-li být skript kompilován, musí mít v záhlaví `Red [needs: view]` . Je-li spoušt n z interak ní konzoly Redu, není tohoto záhlaví zapot ebí, nebo konzola modul View již obsahuje.

P íklad skriptu:

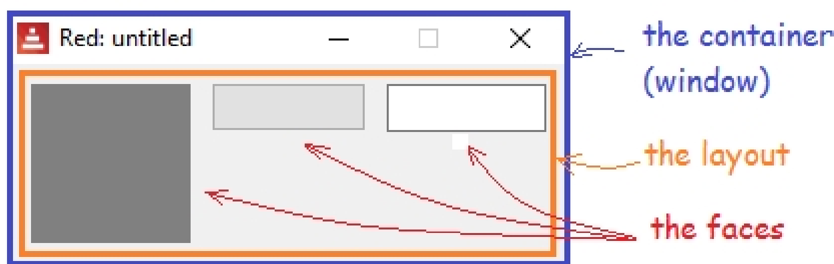
```
Red [needs: view]

view[
        base
        button
        field
]
```

A jeho výsledné GUI:



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

K následujícímu obrázku nutno poznamenat, že **layout** je pouze uspo ádání následných piškot base, button a field v piškotu (kontejeneru) window, nikoliv kontejner jak nazna eno :



P íkazy pro uspo ádání piškot (layout commands) se píši p ed vlastní definice piškot :

```
view [
        Layout command
        Layout command
        widget (face)
        widget (face)
        widget (face)
    ]
```

V následující ukázce deklaruje p íkaz `below` (layout command) azení piškot 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 p íkazy dispozice (layout commands) umož ují podrobn jší specifikaci, jako je velikost, barva, atp. Aspekty (**facets**) piškot ur ují jejich vlastnosti a vzájemné vztahy.

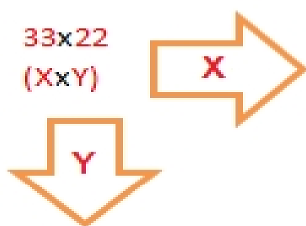
Sou ástí piškotu m že být také blok p íkaz (actor neboli aktér), definující akci piškotu. Tento blok je posléze obsažen v aspektu **actors**.

view [

Container settings	Container settings details
Layout command	Layout command details
face	Face details (facets) [face action]
face	Face details (facets) [face action]
face	Face details (facets) [face action]

]

Note:
Red's coordinate system



P íklad 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 po ít s jednotlivými zadanými hodnotami již na základ   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 zprost edkovat, uložený v aspektu actors. Zvláštním dobrodiním je skute nost, že na po adí aspekt   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` umož uje p ipojovat up esn ní, která m ní okno samotné (nikoliv dispozici uvnit okna). Tato up esn ní (refinements) jsou deklarována hned za slovem `view` a precizována ve stejn   azených samostatných blocích za hlavním blokem `view` :

view / refinement1/ refinement2... [

Container settings	Container settings details
Layout command	Layout command details
face	Face details (facets) [face action]
face	Face details (facets) [face action]
face	Face details (facets) [face action]

] [refinement1 details] [refinement2 details]

Zavedeným up esn ním funkce `view` jsou **flags** (slova, m nící zobrazení nebo chování okna) a **options** (další vlastnosti okna ve formátu 'name: value'). Tato slova mají dva významy - jednak to jsou názvy aspekt   a za druhé to jsou hromadná ozna ení skupiny hodnot.

V následujícím skriptu se up esn ním /flags íká, že zobrazované okno je modálního typu (modal) a lze m nit jeho velikost (resize), zatímco up esn ní /options zajistí zobrazení okna v levém horním rohu obrazovky s posunem (offset) 50 pixel   dol   a 50 pixel   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:

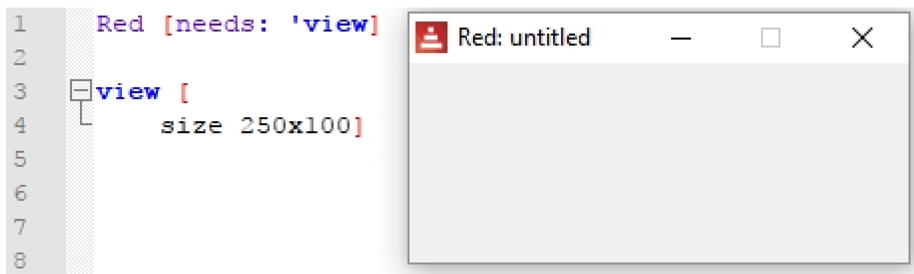
Podrobn jší popis **up esn ní** pro window je uveden v následujcící kapitole.

Created with the Standard Edition of HelpNDoc: Free EPub and documentation generator

GUI - Nastavení kontejneru

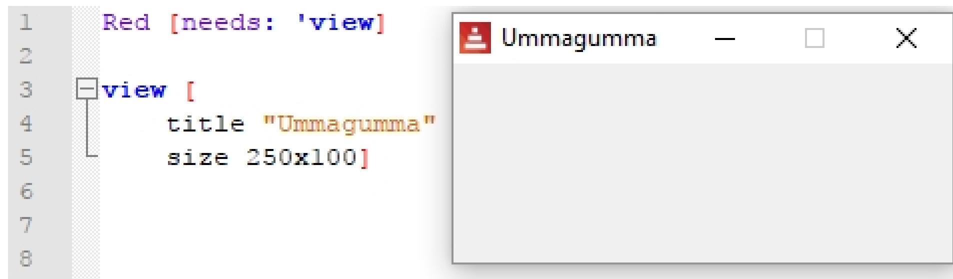
Tato nastavení definují úpravy okna (windows), které bude obsahovat další prvky GUI. Na rozdíl od hodnot aspekt je nutné hodnoty pro parametry okna uvést jejich p íslušnými názvy: **size**, **title** a **backdrop**:

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 ást titulku (min. 212 x ...), nelze jej p emis ovat. Implicitní velikost okna je 110x110 px.

VID DLS **title** - Nastaví titulek v záhlaví okna.



VID DLS **backdrop** - Nastaví barvu pozadí okna.

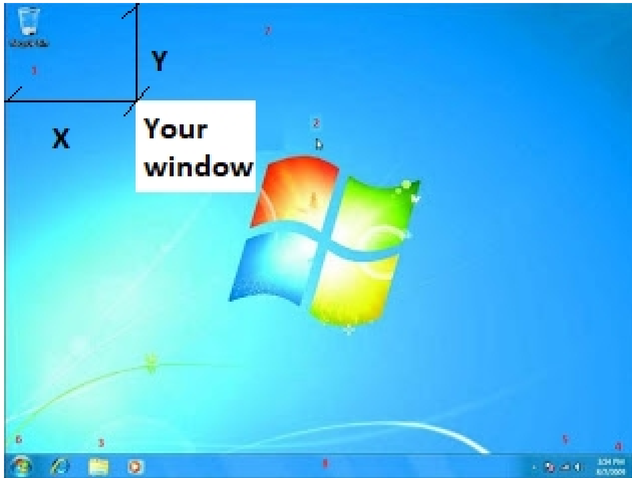
Up esn ní - refinements

Další možností modifikace okna (windows) jsou up esn ní: **options, flags** a **no-wait.** Options a flags jsou aspekty piškot a definují se v blocích za hlavním blokem `view` .

Options je aspekt, který ur uje vlastnosti piškotu window ve formátu [name: value]. Hodnotou v samostatném bloku za blokem view lze vyjád it dva parametry okna:

- **offset** - odsazení okna od levého horního rohu obrazovky, uvád né v pixelech ve formátu pair!
- **size** - velikost okna v pixelech ve formátu pair! (lze tedy velikost okna zadat dvojím zp sobem

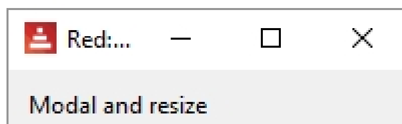
```
Red [needs: view]

view/options [                          ; /deklarace up esn ní
    button 100x40 "click me" [quit]]
    [offset: 100x70]                    ; definice up esn ní
```



Flags - je aspekt, který rovn ž m ní zobrazení nebo chování piškotu window. Disponuje adou parametr (**flag**):

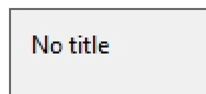
- **modal** - vyžaduje pozornost (focus) tím, že zmrazí všechna ostatní okna, dokud modální okno nezav ete. Vytvo íte-li okno s flagy modal, no-title a no-border, je velmi obtížné se jej zbavit.
- **resize** - umožní zm nu velikosti okna (implicitní je stálá velikost).

```
Red [needs: 'view]
View/flags [ size 200x30 text "Modal and resize" ] [modal resize]
```



- **no-title** - nezobrazuje název okna

```
Red [needs: 'view]
View/flags [ text "No title" ] [no-title]
```

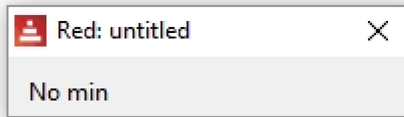


- **no-border** - odebere ráme ek kolem okna

```
Red [needs: 'view]
View/flags [ text "No border" ] [no-border]
```

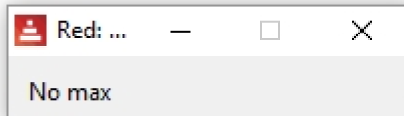
- **no-min** - eliminuje tla ítka 'min' a 'max' v záhlaví okna

```
Red [needs: 'view]
View/flags [ size 200x30 text "No min" ] [no-min]
```



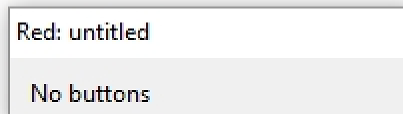
- **no-max** - tla ítko 'max' se zobrazí jako neaktivní

```
Red [needs: 'view]
View/flags [ size 200x30 text "No max" ] [no-max]
```



- **no-buttons** - nezobrazí žádné tla ítko (maximize, minimize, close) v záhlaví okna

```
Red [needs: 'view]
View/flags [ size 200x30 text "No buttons" ] [no-buttons]
```



- **popup** - pouze pro Windows - iní z okna výsuvné okno (popup). Umož uje ostatním okn m z stat aktivní. Zav e se p i p esunu fokusu na jiné okno.

No-wait - je up esn ní pro funkci view, p ikazující aby nebyla aplikována implicitní smy ka událostí (jaká že? ...)

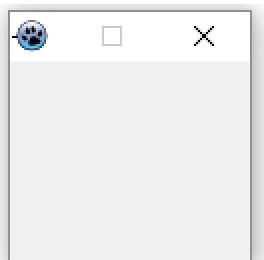
Actors

Viz dedikovanou kapitolu.

Zadání ikony - Chodí jenom p i kompilaci kódu, nikoliv p i jeho interpretaci!

Nejde o nastavení okna ale snad se to sem hodí. Chcete-li zadat svoji ikonu do okna, musíte ji p idat v deklaraci záhlaví skriptu :: `icon: <path-to-icon>`:

```
1  Red [needs: 'view
2       icon: %project1.ico]
3
4  view []
5
6
7
8
```



Created with the Standard Edition of HelpNDoc: Free Qt Help documentation generator

GUI - Layout

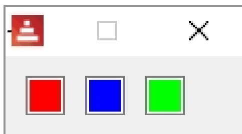
Rozložení (layout) piškot uvnit piškotu **window** je ízeno t mito pokyny: across, below, return, space, origin, at, pad. Implicitní nastavení pro window je:

- po átek (origin) : *10x10*
- mezera (space) : *10x10*
- sm r (direction) : *acros*
- p i azení (alignment): *top*

P i azení m že pro pokyny across, below a return nabývat hodnot *top, middle, bottom, left, center, right nechodí, neznám syntaxi*.

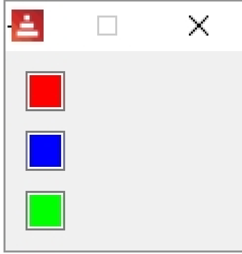
VID DLS across (top | middle| bottom)

```
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
]
```



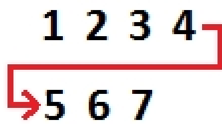
VID DLS below (left |center | right)

```
Red [needs: view]

view [
        below
        area 20x20 red
        area 20x20 blue
        area 20x20 green
]
```



VID DLS return

M ní zadaný mód **across**:

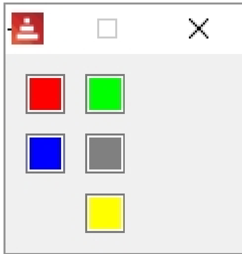



```
Red [needs: view]

view [
        across
        area 20x20 red
        area 20x20 blue
        return
        area 20x20 green
        area 20x20 gray
        area 20x20 yellow
]
```

M ní zadaný mód **below:**

```
Red [needs: view]

view [
        below
        area 20x20 red
```

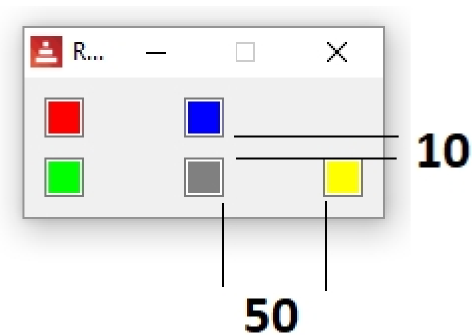
```
        area 20x20 blue
        return
        area 20x20 green
        area 20x20 gray
        area 20x20 yellow
]
```



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
]
```

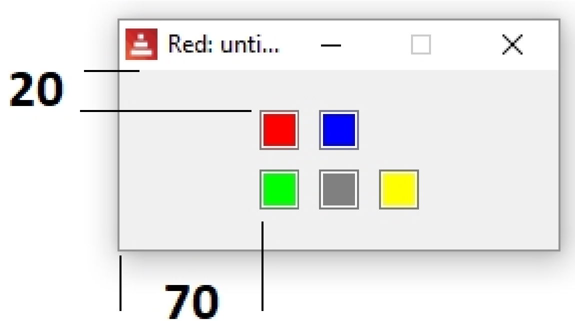


VID DLS origin

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 absolutn   zadané pozice. Neovliv uje umíst ní ostatních piškot .

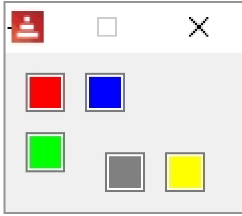
```
Red [needs: view]

view [
        across
        area 20x20 red
        area 20x20 blue
        return
        area 20x20 green
        at 2x5
        area 20x20 gray
        area 20x20 yellow
]
```

VID DLS pad

M ní dispozici okna relativním odsazením následujících piškot   od p vodn   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 exeku ní p íkaz, popsaný v kapitole Running code. V tomto p ípad je použit pro spušt ní kódu uvnit kontejneru view.

Lze bez problém 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
]
```

Uvnit okna musí být blok pod ízen p íkazu "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

GUI - Piškoty (faces)

Piškoty neboli **faces** (neboli widgets)**,** jsou základní stavební kameny grafického rozhraní. Interní sou ástí každého piškotu je sada 23 **aspekt** (facets), jimiž jsou ur ovány jednotlivé parametry piškotu. T mito parametry je ur ován nejen vzhed piškot , ale i jejich odezva na r zné události a souvislost s jinými piškoty.

Seznam aspekt (facets)

Facet	**Datatype**	**Povinné?**	**Použití**	**Popis**
type	word!	yes	all	Typ grafické komponenty
offset	pair!	yes	all	Ofset reflektovaného kurzoru od po átku vlevo naho e
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 i skrýt piškot.
selected	integer!	no	some	Index aktuáln vybraného elementu seznamu.
flags	block!, word!	no	some	Klí ová slova, m nící zobrazení nebo chování piškotu.
options	block!	no	some	Další vlastnost piškotu ve formátu [name: value]
parent	object!	no	all	Odkaz na rodi ovský piškot,

				pokud existuje.
pane	block!	no	some	Seznam d tí, zobrazených uvnit piškotu.
state	block	no	all	Info o interním stavu piškotu *(pouze u Viewengine).*
rate	integer!, time!	no	all	asova piškotu. Integer udává frequenci, time udá- vá trvání, none zastavuje.
edge	object!	no	all	*(reserved for future use)*
para	object!	no	all	Pokyny pro umíst ní textu.
font	object!	no	all	Nastavení atribut fontu
actors	object!	no	all	Uživatelsky vytvo ené ovlada e událostí.
extra	any-type!	no	all	Volitelná uživatelská data p ipojená k piškotu.
draw	block!	no	all	Seznam p íkaz Draw, jež mají být provedeny.

Datové typy hodnot aspekt :

Hodnoty aspekt se zadávají p ímo, bez uvedení jejich ur ení, které intern provádí interpret na základ jejich datového typu.

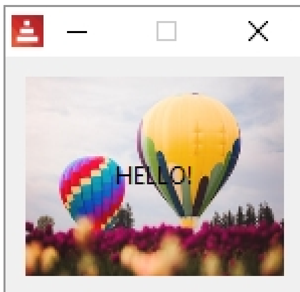
Datový typ	Aspekt	Ú el použití
integer!	rate, selected	Ur uje ší ku piškotu, výška z stává implicitní. U panel udává po et ad i sloupc .
pair!	offset, size	Ur uje ší ku a výšku piškotu.
tuple!	color	Ur uje barvu pozadí piškotu ve formátu rgb, nap . 211.9.84.
issue!	color	Ur uje barvu pozadí piškotu použitím hexadecimálního zápisu, nap íklad #d30954.
word!	color	Ur uje barvu pozadí piškotu slovem, nap . aqua
string!	text	Ur uje text, zobrazovaný piškotem.

percent!		Nastavuje aspekt `data` (užite né pro piškoty `progress` a `slider`).
logic!		Nastavuje aspekt `data` (užite né pro piškoty `check` a `radio`).
image!	image	Nastavuje obrázek, zobrazený jako pozadí piškotu.
url!		Na te zdroj, na n jž URL ukazuje.
block!		Udává akci pro implicitní i deklarovanou událost piškotu. U panel ur uje obsah.
get-word!		Jako aktéra používá existující funkci.
time!		Ur uje asový úsek delší než 1 sec v aspektu `rate`

P íklad aplikace aspekt :

```
Red [needs: view]

view [
     base "HELLO!" 130x100 %balloon.jpeg   ; base je piškot, následují
hodnoty  aspekt  text, size, image
]
```

(soubor `%baloon.jpeg` musí být uložen ve stejném adresá i jako provád ka `red.exe`)



Atributy aspektu font

Aspekt font má 9 p eddefinovaných formátovacích atribut : name, size, style, angle, color, anti-alias?, shadow, state, parent. Prost ednictvím atributu parent mohou na daný aspekt odkazovat i jiné piškoty.

P i deklaraci se název atributu uvádí za poml kou, nap íklad **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 font : `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 rodi ovský piškot

P íklad 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, p íslovce
```

Klí ová slova (keywords):

Klí ovými slovy (typu word!) se p i azují slovní hodnoty r zným aspekt m (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.

P ípadn slouží jako hodnoty up esn ní p i formování piškotu (window):

- /flags :: modal, resize, no-title, no-border, no-min, no-max, no-buttons, popup
- /options:: on-(enter, change, select, key)

Akté i - actors

Slovo **actors** (akté i) je název jednoho z aspekt a rovn ž spole né ozna ení funkcí pro ošet ení událostí. Aktér (actor) je funkce, která popisuje aktivitu piškotu p i výskytu jisté události - viz další kapitola.

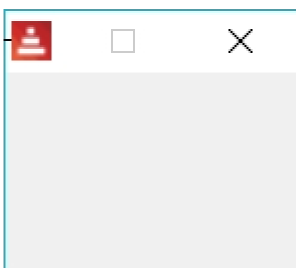
Piškoty - faces

Knihovna **view** disponuje t mito **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 podrobn ji popsány.

Rozší ením piškotu **text** jsou piškoty **h1** až **h5**, rozší ením piškotu **base** jsou piškoty **box** a **image**.

Nejvýše postaveným objekty jsou diskrétní piškoty **screen** a **window**. Tyto piškoty se nedají samostatn deklarovat. Jsou však automaticky p ítomny p i každé vizualizaci bloku funkcí **view**, nap íklad i prázdného:

```
Red [needs: view] ; "needs: view" je pot ebné pouze p i kompilaci
skriptu

view []
```



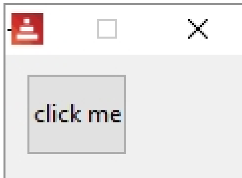
Aktivita piškot

V tšina piškot (vyjímaje screen a window) si ví rady s výskytem n jaké události. Tou událostí m že být kliknutí myší klávesy, stisknutí klávesy na klávesnici nebo provedení výb ru. Výpis t chto událostí je uveden na po átku další kapitoly. Implicitn nastavené události (ovlada e) jsou dále uvedeny u každého piškotu.

Pro tla ítka (buttons) je touto implicitn ošet enou událostí stisk levé klávesy myši (down) a

v následující ukázce spouští p íkaz `quit`, který kon í program. Místo funkce `[quit]` lze použít nap íklad `[print "Tornado Loo"]`.

```
Red [needs: view]

view [
      button 50x40 "click me" [quit] ; piškot, aspekt, aspekt, aktér
]
```



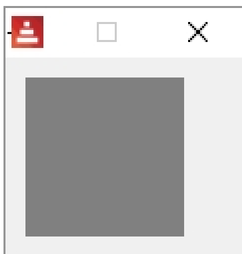
Citlivost piškotu na událost se projeví jen v tom p ípad , je-li deklarace piškotu dopln na blokem s názvem definované funkce nebo s definicí uživatelské funkce - neboli aktérem.

P ehled piškot

base (on-down)

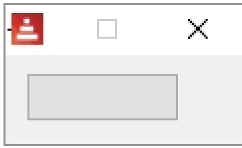
Piškot **base** bez parametru zobrazí uvnit piškotu **window** (jehož rodi em je piškot **screen**) tmavošedý (128.128.128) tverec (80x80 px) bez ohrani ení.

```
Red [needs: view]

view [
      base
]
```



button (on-click)

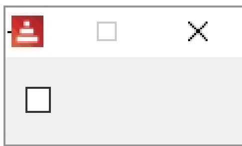
```
Red [needs: view]

view [
      button
]
```



check (on-change)

Piškot reaguje na zm nu stavu zatržení uživatelem.

```
Red [needs: view]
view [
        check
]
```



Aktuální stav zatržítka je uložen v aspektu ***data*** *(true nebo false)*

```
Red [needs: 'view]
view [b: check "unchecked"[either b/data           ;Red's if-else
                           [b/text: "checked"]    ;if "data" is true!
                           [b/text: "unchecked"]  ;if "data" is false!
                          ]
     ]
```

radio (on-change)

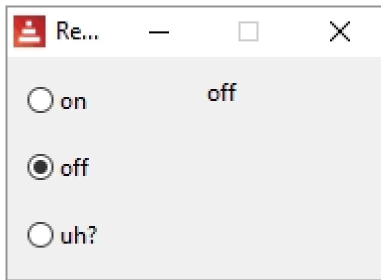
Událost, registrující u piškotu vstup textu nebo výb r ze seznamu, se nazývá *change.*

P i každé volb   (zde on, off, uh?) umožní realizaci p íslušného aktéra - p íslušného bloku.

Piškoty radio tvo í obsah kontejneru **window**. V kontejneru m že být zatrženo pouze jedno výb rové tla ítko **radio**.

```
Red [needs: view]
view [
        r1: radio "on" [t/text: "on"]
        t: text "none"      ; jen p ed první volbou
        return
        below
```

```
        r2: radio "off" [t/text: "off"]
        r3: radio "uh?" [t/text: "uh?"]
] ; prom nná, piškot, aspekt, aktér
```

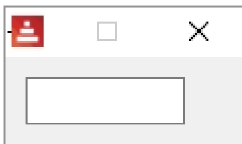


field (on-enter)

Piškot field slouží k vložení textu.

Text v poli piškotu field je uložen v aspektu ***data***. Chodí to však obojím sm rem. Zm níte-li obsah aspektu ***data*** (*.../data value*), zm ní se zobrazený text v poli.

```
Red [needs: view]
view [
        field
]
```

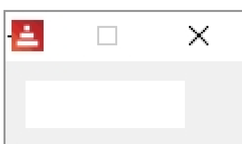


V této ukázce se vytiskne zápis v poli do konzoly p i (každém) stisku Enter:

```
Red [needs: view]
view [
        f: field [print f/text]
] ; prom nná, piškot, aktér
```

Piškot `field` podporuje flag `no-border` :

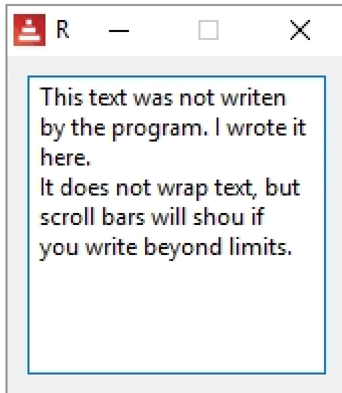
```
Red [needs: view]

view [field no-border]
```



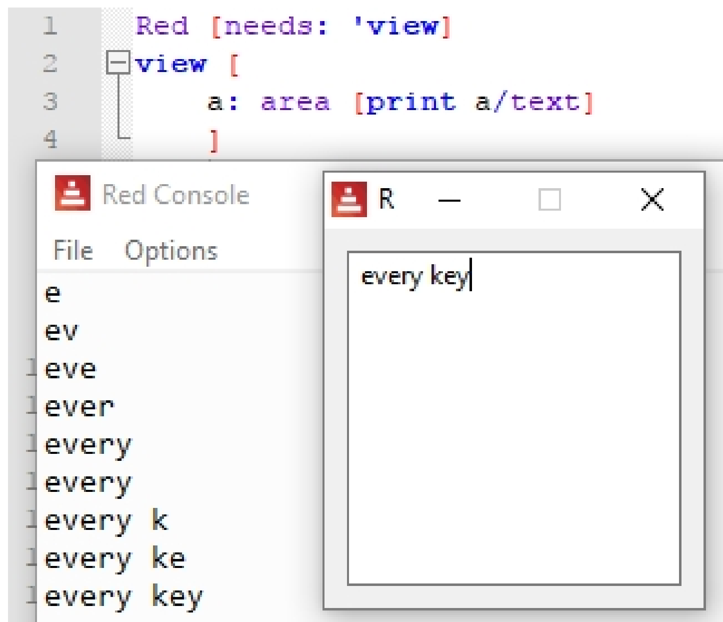
area (on-change)

Text uvnit piškotu `area` je uložen v aspektu ***text***. Do zobrazeného okna lze psát více ádkový text:

```
Red [needs: view]
view [
        area
]
```



Živ jší aplikaci vytvo íme p ipojením aktéra *[print a/text]*, který tiskne aktuální obsah aspektu ***text*** do konzoly. Aktér je aktivován stiskem každé klávesy na klávesnici, v etn p íkazu Enter:

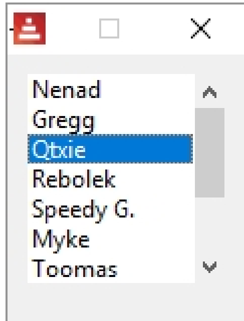


text-list (on-change)

Procházené et zce jsou uloženy v aspektu ***`data`***. Index vybraného et zce 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 p ipojit

```
pick face/data face/selected
```

```
    [ t/text: pick face/data face/selected ] viz drop-down
```

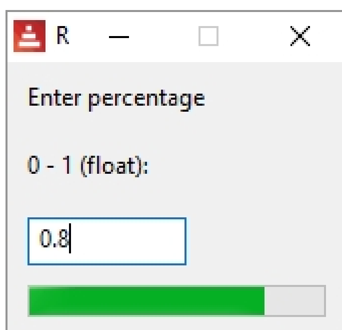
This would be the same as : `pick ["Nenad" "Greg" "Qtxie" "Rebolek" (...)] 3`

===

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
]
```

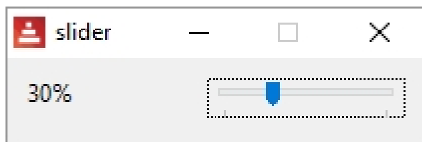


slider (on-change)

Aktuální procentní hodnota je uložena v aspektu data jako datový typ `percent!` nebo `float!`.

```
Red [needs: view]
view [
      title "slider"
      t: text "Percentage"
      slider 100x20 data 10% [t/text: to string! face/data]
]    ; ozna ení aspektu 'data' je nadbyte né ale užite né pro
porozumn ní kódu
```

Pohybem jazý ku m níte hodnotu procenta. Implicitn nastavená hodnota je 10% :



Všimneme si zde, jak je zadána hodnota aspektu data. V tomto p ípad by šlo slovo 'data' vynechat - na rozdíl od deklarace piškotu text-list, kde p i azovanou hodnotou k aspektu data byl blok.

panel (on-down)

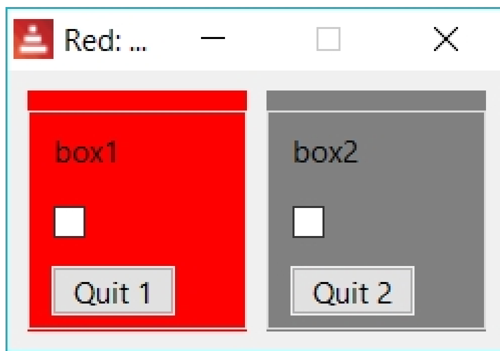
Vytvá í spole ný piškot (uvnit piškotu window a screen) pro umíst ní dalších piškot .

```
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. Rovn ž slouží pro umíst ní dalších piškot . 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]]
]
```



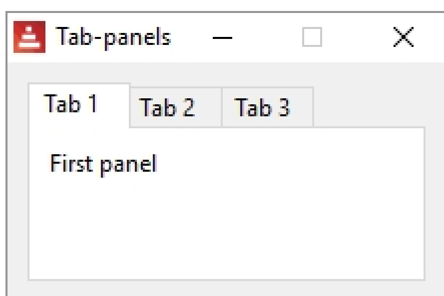
tab-panel (on-select)

Vytvo í sadu karet s oušky, formou p ekrývajících se panel   v piškotu window. Každý panel m že obsahovat více piškot . Informace o sestav   jsou uloženy v t chto aspektech:

`<tab-panel>/data` - blok s názvy oušek (string!).
`<tab-panel>/pane` - odkaz na d tské piškoty (block!).
`<tab-panel>/selected` - index vybraného panelu nebo hodnota `none` .

Na rozdíl od piškotu group-box je piškot tab-panel expilicitn   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]
```

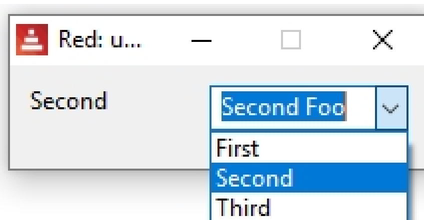
```
view [
        Title "Tab-panels"
        tab-panel 110x140 [
                "Tab 1 " [
                below
                text font-color blue "First panel"
                button "quit" [quit]
                check "check to quit" [quit]
                ]
                "Tab 2 " [text "Second panel"]
        ]
]
```

drop-down (on-select, on-change, ~~on-enter~~)

Roletka s vertikálním seznamem et zc . Aspekt `data` p ijímá jakékoli hodnoty ale pouze et zce jsou p idávány do seznamu a zobrazeny. Ne et zcové hodnoty lze použít pro vytvo ení asociativních ad (arrays) s použitím et zc   jako klí   .

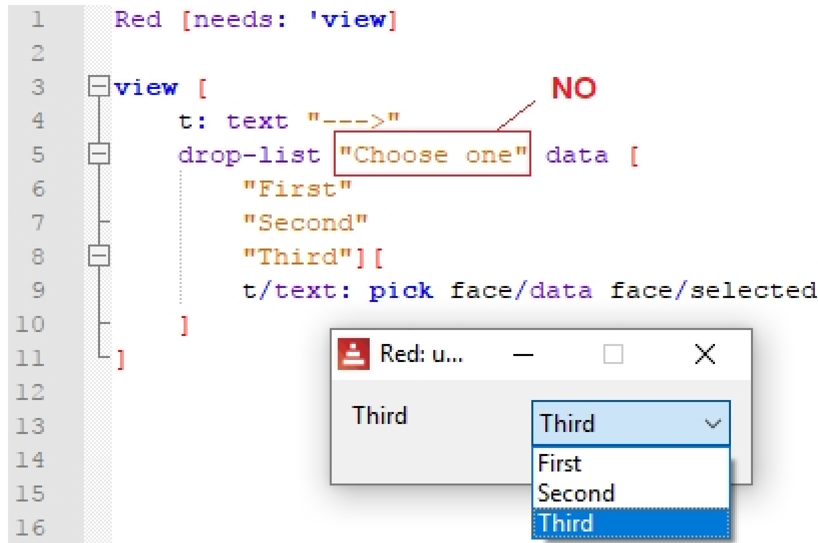
Aspekt `selected` obsahuje celo íselný index vybrané položky. Aspekt `text` obsahuje aktuáln   vybranou položku. Zobrazí se p i stisku Enter.

```
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)

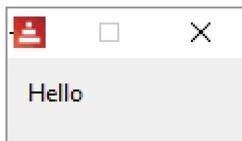
Stejné jako drop-down, akorát že se nezobrazí et zec aspektu text a název položky se projeví hned po jejím výb ru v roletce.



text (on-down)

Piškot **text** je autonomní entita s rodi i **window** a **screen** (p ípadn panel, p ípadn jiný piškot).

```
Red [needs: view]

view [
        text "Hello"          ; piškot text plus hodnota aspektu text
]
```



camera (on-down)

Nutno doplnit komentá

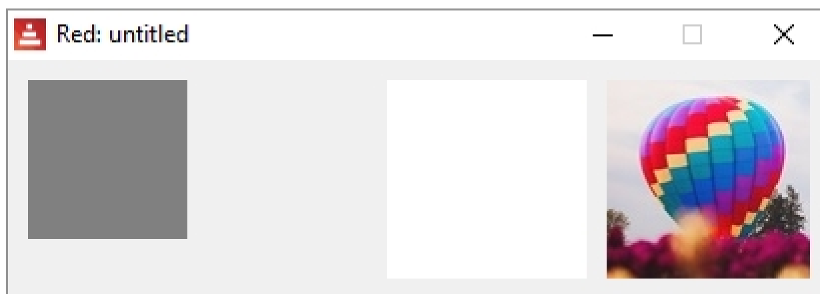
box (on-down) image (on-down)

Piškoty **box** a **image** jsou variantami piškotu **base** (type: ´base).

box bez parametru vymezí v nad azeném piškotu window transparentní tverec (80x80 px) bez ohrani ení

image bez parametru vymezí v nad azeném piškotu window bílý tverec (100x100 px) bez ohrani ení. 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 aspekt , stejn 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 klí ové slovo
      button "underline" underline
      button "strike" strike
      return
      button "top" 70x70 top          ; hodnoty aspekt  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! (nap . aqua) nebo tuple! (0.255.255) i issue! (#00ffff) - bez uvedení názvu aspektu. V následující ukázce vidíme deklaraci 32 piškot base uvnit automaticky generovaného piškotu window:

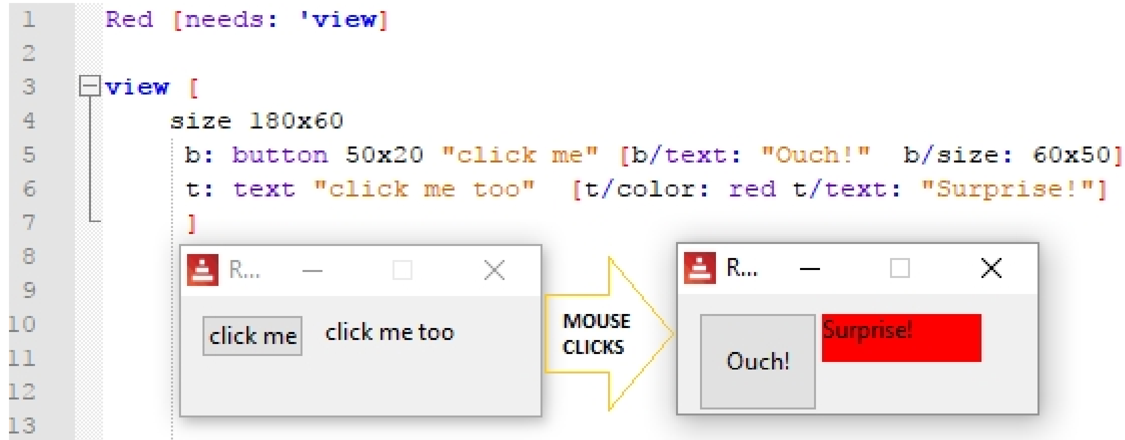
```
Red [needs: view]
view [
        base 30x30 aqua  text "aqua"        base 30x30 beige text "beige"

        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 p knou paletu. Kdybychom popis barev v uvozovkách použili bez anotace text, šlo by o aplikaci **aspektu text** a popisky by byly uvnit barevných polí. Takto jde o deklaraci **piškotu text** se zadanou hodnotou aspektu text.

Dynamická zm na aspekt

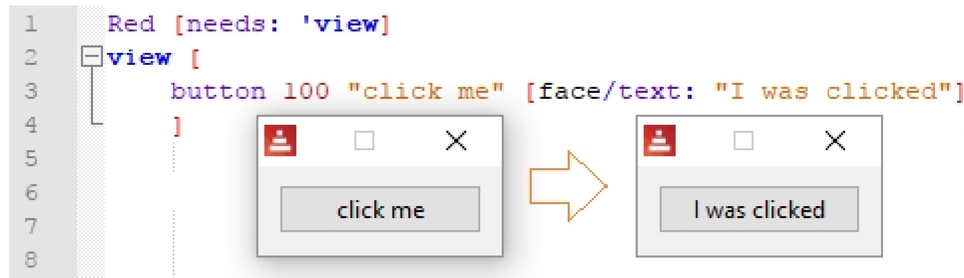
P i b hu programu lze m nit hodnoty jejich aspekt   (facets):



Všimn te si, že p i deklaraci jednotlivých aspekt piškotu zadáváme p ímo jejich p íslušné hodnoty (`50x20 "click me"`), zatímco p i zm n hodnot (zde v hranatých závorkách) uvádíme rovn ž název p íslušného aspektu (`b/text: "Ouch!"`), p i emž m žeme uplatnit i aspekt p vodn nepoužitý (`t/color: red`), p ípadn m žeme odkázat na aspekt jiného piškotu, nap íklad:

```
  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

GUI - Události a akté i

Události - events:

Stisk klávesy na klávesnici i myši, p ejížd ní kurzorem p es zam ené pole na obrazovce, výb r položky - to všechno jsou události, které lze podchytit vhodn napsanou definicí piškotu a spojit je s n jakou aktivitou - implicitn definovanou nebo uživatelsky ur enou. V následující tabulce je výpis aktuáln použitelných názv událostí:

Jméno	Prost edek	Akce
down	mouse	Stisknuté levé tla ítko myši.
up	mouse	Uvoln né levé tla ítko myši.
mid-down	mouse	Stisknuté prost ední tla ítko myši.
mid-up	mouse	Uvoln né prost ední tla ítko myši.
alt-down	mouse	Stisknuté pravé tla ítko myši.
alt-up	mouse	Uvoln né pravé tla ítko myši.
aux-down	mouse	Stisknuté pomocné tla ítko myši.
aux-up	mouse	Uvoln né pomocné tla ítko myši.
drag-start	mouse	Piškot za íná být tažen.
drag	mouse	Piškot je tažen.
drop	mouse	Od tažení piškotu bylo upušt no.
click	mouse	Klik levým tla ítkem myši (pouze pro button).
dbl-click	mouse	Dvojklik levým tla ítkem myši.
over	mouse	Kurzor myši p echází p es 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 rovn ž produkovány všechny mezilehlé události.
move	mouse	Došlo k pusunu okna.
resize	mouse	Byla zm n na velikost okna.
moving	mouse	Okno je posouváno.
resizing	mouse	M ní se velikost okna.
wheel	mouse	Otá í se kole ko 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

		(krom kláves Ctrl, Shift a Menu).
key-up	keyboard	Uvoln ná klávesa klávesnice.
enter	keyboard	Stisknutá klávesa Enter.
focus	any	Piškot práv p ešel do režimu *focus*.
unfocus	any	Piškot práv ztratil *focus*.
select	any	V piškotu je provád n výb r z více možností
change	any	V piškotu došlo ke zm n v d sledku vstupu uživatele (vložení textu nebo výb r ze seznamu).
menu	any	Výb r položky z menu.
close	any	Zav ení okna.
time	timer	Prob hla prodleva, nastavená aspektem `rate` piškotu.

N které události jsou defini n spojeny s daným piškotem, p ípadn aspektem (nap íklad aspekt rate má asova `on-time)`. Každý piškot je schopen akceptovat jenom vymezenou sadu událostí. Uživatelsky definované p ipojení události k piškotu se provádí za vý tem aspekt :

```
view [
        face  facet facet [action facet]
                on-event [action]
                on-event [action]
        face2 facet facet [action facet]
                on-event [action]
                on-event [action]
    ]
```

P íklady n kterých událostí

Všechny p íklady si vyzkoušejte v konzole. Termín nap . on-down je spojení obecné p edpony on- a názvu události z tabulky.

down, over :

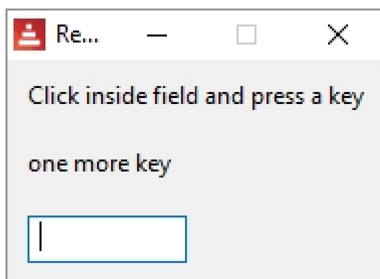
```
Red [needs: view]
view [
        t: area 40x40 blue
        on-down [quit]
        on-over [either t/color = red [t/color: blue][t/color: red]]
]
```

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 p íklad bliká zadaný text zadanou frekvencí (zde jednou za vte inu - viz kapitola GUI- Pokro ilá témata):

```
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) - p i zav ení okna se má v konzole vytisknout "bye!" - (ale nevytiskne)

```
Red [needs: view]

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

Akté i - actors

Slovo **actors** (akté i) je název jednoho z aspekt a rovn ž spole né ozna ení funkcí pro ošet ení událostí. Tyto funkce jsou definovány v bloku za náv stím on-<event>. Výpis p ípadných událostí je uveden v tabulce na za átku 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]]
]
```

Spušt ní 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 vnit ní náhled interpreta na skladbu realizovaného kontejneru. Jde o výpis všech aspekt   (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 p íslušejícími hodotami.

Vidíme, že první aspekt parent má ve svém defini ním bloku rovn ž kompletní sestavu aspekt , stejn   jako jeho zano ený další aspekt parent.

Poznámka:

- Dotekové (touch) události nejsou realizovatelné ve Windows XP.

- Jedna i více `pohybových` událostí vždy p edchází událost `move`.
- Jedna i více událostí `měnících rozměr` vždy p edchází událost `resize`.

Created with the Standard Edition of HelpNDoc: Easily create EPub books

GUI - Introspekce událostí

Datový typ event!

Krom názv jednotlivých událostí (viz p edchozí kapitola) existuje entita, zvaná datový typ **event!**, který obsahuje 13 funkcí, použitelných k introspekci provád ného kódu.

Instance datového typu event!

Název	Typ hodnoty	Vracená hodnota
`type`	word!	Název události
`face`	object!	Piškot, v n mž k události došlo
`window`	object!	Okno, v n mž k události došlo
`offset`	pair!	Sou adnice kurzoru myši, relativn k piškotu p i výskytu události. Pro události gest vrací sou adnice st edního bodu.
`key`	char! word!	Stla ení klávesnice
`picked`	integer! percent! word!	Nová položka, vybraná v piškotu (`integer!` `percent!`). P i události myši `down` v piškotu `text-list` vrací index položky pod kurzorem nebo `none`. P i události `wheel` vrací po et (+/-) oto ení kole ka. P i události `menu` vrací odpovídající ID položky (`word!`). P i gestu `zoom` vrací procento relativního zv tšení/zmenšení. U jiných gest je jeho hodnota závislá na OS (Windows: pole `allArguments` ze struktury GESTUREINFO).
`flags`	block!	Vrací výpis s jedním i více flagy (away, down, mid-down, alt-down, aux-down, control, shift)
`away?`	logic!	Vrací `true`, opouští-li kurzor hranice piškotu . Platí pouze p i aktivní události `over`.
`down?`	logic!	Vrací `true`, byla-li stisknuta levá klávesa myši
`mid-down?`	logic!	Vrací `true`, byla-li stisknuta st ední 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 flag 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 p íkazem

```
>> event/ <název typu>
```

Událost myši:

V níže uvedeném zjednodušeném p íkladu vytiskneme typ události a sou adnice kurzoru p i 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 p i vhodné volb události pro klávesnici - zde nap íklad událost `key`:

```
Red [needs: view]

view [
    area 100x100
    on-key [
            print event/type
            print event/offset     ; chodí podívn
            print event/key
        ]
]
```

```
key
-59x84
r
key
-36x59
s
key
-116x79
o
```

Zdá se, že n které piškoty negenerují události klávesnice. Když nap íklad ve výše uvedeném p íkladu zam níme 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

GUI - Pokro ilá témata

VID DLS style

N kdy je zapot ebí nadefinovat si vlastní piškot. Pro tento ú el použijeme funkci `style`. P knou ukázku tohoto postupu lze nalézt (anglicky) zde .

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   ; zm na hodnoty aspektu color
	]
]
```

function! view a unview

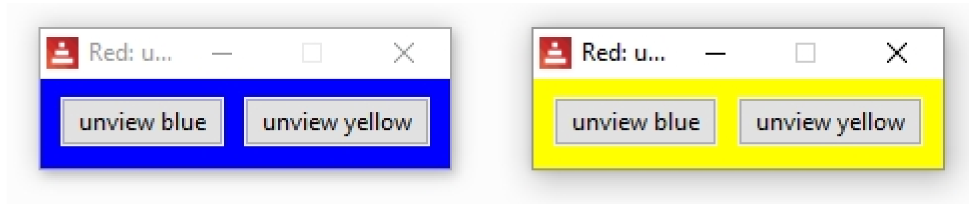
Vícero oken v obrazovce

Funkci `view` lze také použít k zobrazení oken, jež byla definována v jiné ásti kódu. Ovšemže, funkce `unview` zobrazení (view) ruší. Následující kód vytvo í dv identická, ale nezávislá okna (rozdílné face trees) v r zných 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 up esn ním `/only` se zam í 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
```



Up esn ní pro funkci `view`:

/tight => nulový odstup od výchozího bodu
/options =>
/flags =>
/no-wait => žádné ekání

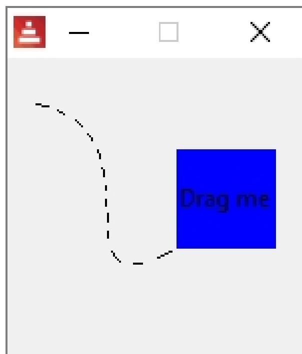
Up esn ní pro funkci `unview`:

/all => Zav e všechna zobrazení
/only => Zav e zadané zobrazení

VID DLS loose

Klí ové slovo (p íslovce) `loose` umož uje vle ení piškotu myší po okn :

```
Red [needs: view]

view [
      size 150x150
      base blue 50x50 "Drag me" loose
]  ; piškot; aspekty color, size, text; p íslovce
```

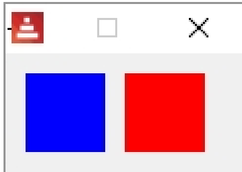


VID DLS all-over

K události `on-over` normáln dochází v okamžiku, když kurzor "vstupuje" nebo "opouští" plochu piškoutu. Zadáte-li flag (p íslovce) `all-over` , potom všechny události, k nimž dojde v situaci, kdy je kurzor v ploše piškotu (nap . pohyby nebo kliknutí), generují událost `on-over`.

V následující ukázce m ní levý tverec barvu pouze p i "vstupu" i "opušt ní" piškotu kurzorem, avšak pravý tverec m ní barvu také p i kliknutí nebo pohybu kurzoru v ploše piškotu:

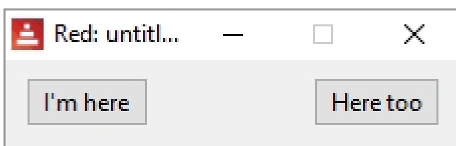
```
Red [needs: view]

view [
      a: base 40x40 blue
            on-over [either a/color = red [a/color: blue][a/color: red]]
      b: base 40x40 blue all-over
   ; prom nná, piškot, aspekt, aspekt, p íslovce
            on-over [either b/color = red [b/color: blue][b/color: red]]
]  ; událost, aktér s variantními reakcemi
```



VID DLS hidden

P íslovce `hidden` iní piškot neviditelným. Možné použití je pro vytvo ení skrytého piškotu s využitým aspektem `rate` (zde není použito), takže lze mít asování bez zobrazení piškotu.

```
Red [needs: view]

view [
      button "I'm here"
      button "I'm not" hidden
      button "Here too"
]
```



VID DLS disabled

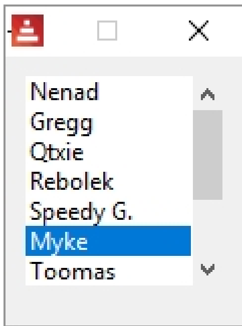
P íslovce `disabled` deaktivuje piškot (nedojde k žádné události, dokud není `enabled`).

```
Red [needs: view]

view [
      button "I should quit, but I don't" disabled [quit]
      button "Quit" [quit]
]
```

VID DLS select

P íslovce `select` ozna í p ednastavený výb r v piškotu `text-list`.

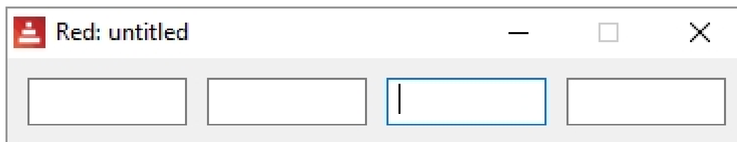
```
Red [needs: view]
view [
        tl: text-list 100x100 data  [
        "Nenad" "Gregg" "Qtxie" "Rebolek"
        "Speedy G." "Myke" "Toomas"
        "Alan" "Nick" "Peter" "Carl"
        ] select 6
        [print tl/selected]
]
```



VID DLS focus

P íslovce **focus** dodá implicitní zam ení ozna enému piškotu p i prvním zobrazení okna. Zam ení m že mít pouze jeden piškot. Je-li deklarováno více focus , má implicitní zam ení ten poslední.

```
Red [needs: view]
view [
    field
    field focus
    field focus
    field
]
```

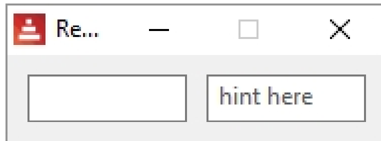


VID DLS hint

P íslovce **hint** zprost edkuje textové sd lení uvnit piškotu `field`. P i zadání nového obsahu (editací pole nebo aspektu `face/text`) stávající text zmizí.

```
Red [needs: view]
view [
    field
    field hint "hint here"
```

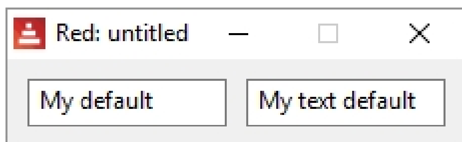
```
]
```



VID DLS default

p íslovce **default** definuje po áte ní hodnotu pro aspekt `data`, vrací-li konverze hodnoty aspektu `text` hodnotu `none`. Aktuáln 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       ; p i azeno p íslovcem "default"
            print b/text
            print b/data       ; error: hodnota aspektu data nebyla
ur ena
      ] ]
```




```
My default
My default
My text default
*** Script Error: My has no value
*** Where: print
*** Stack: view layout do-safe
```

VID DLS with

ekn me, že chcete vytvo it piškot s hodnotami aspekt , které se vyhodnocují p i realizaci piškotu. Vyhodnocení nem žete uskute nit coby argumenty piškotu, tudíž je zadáte pomocí slova `with` .

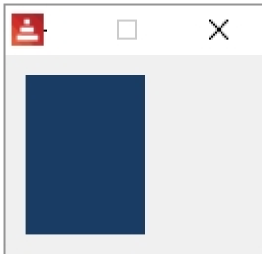
Toto nechodí:

```
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 asova em. Když asova (timer) "tikne", generuje se událost `on-time`. Pamatujte, že argumentem `rate` je "kolikrát za vte inu", takže `rate` 20 je rychlejší než `rate` 5.

Tento kód zp sobí blikání textu:

```
Red [needs: view]

view [
        t: text "" rate 2
        on-time [either t/text = "" [t/text: "Blink"] [t/text: ""]]
]
```

Tento kód vytvo í jednoduchou animaci, kde modrý piškot base cestuje nap í 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 asové úseky delší než 1 vte inu, 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 klí ové slovo (funkce), které spojuje chování jednoho piškotu s daty jiného piškotu.

Klasický p íklad:

```
Red [Needs: view]

view[
    progress 100x20 20% react [face/data: s/data]
      s: slider 100x20 20%
]
```

Piškot progress reaguje na pohyb posuvníku piškotu slider:

Up esn ní funkce react

/link => Spojí objekty reaktivním vztahem.

/unlink => Zruší existující reaktivní vztah.

/later => Spustí reakci p i p íští zm n .

/with => Ur í vybraný pišktot (interní použití).

function! layout

Funkce `layout` se používá k vytvo ení grafických útvar   bez jejich bezprost edního použití. Funkce definuje blok s popisem útvaru a s aktérem 'unview' nebo 'view'. Je aktivována pozd jším odkazem na její jméno.

Následující kód nap íklad zobrazí jedno okno a po jeho zav ení 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 p es celou obrazovku:

Následující skript vytvo í zobrazení p es celou obrazovku :

```
Red [needs: view]

view [size system/view/screens/1/size]
```

Created with the Standard Edition of HelpNDoc: Produce Kindle eBooks easily

Draw

Obšírn jší informaci o DSL Draw lze nalézt v dokumentaci Redu, odkud jsou popisy p íkaz p evzaty. Sou ástí DSL Draw je sub-dialekt Shape.

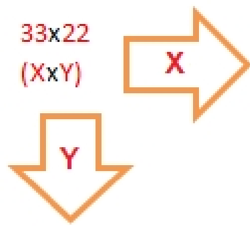
Funkce **draw** se používá ke kreslení rovinných útvar . Tyto útvary jsou vyjád eny jako seznamy (block!) direktiv pro funkci draw.

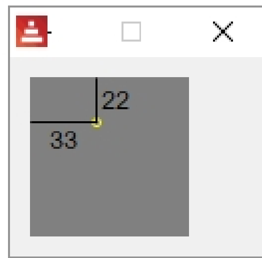
Celkem 30 p íkaz dialektu draw lze rozd lit do ty skupin:

- **Kreslící instrukce**: line, triangle, box, polygon, circle, ellipse, arc, curve, spline, image, text, font
- **Vlastnosti áry**: 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

Sou adnicový systém

Note:
Red’s coordinate system



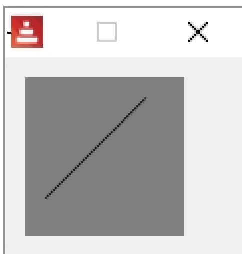


Kreslící instrukce

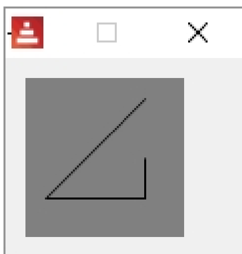
DRAW line

Nakreslí úse ku mezi dv ma body. Je-li zadáno více bod , kreslí se další úse ky, napojované na p edchozí.

```
Red [needs: view]
view [
        base draw [line 60x10 10x60]
]
```



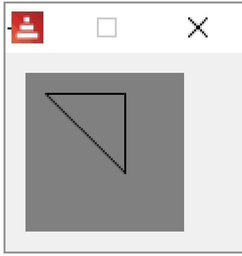
```
Red [needs: view]
view [
        base draw [line 60x10 10x60 60x60 60x40]
]
```



DRAW triangle

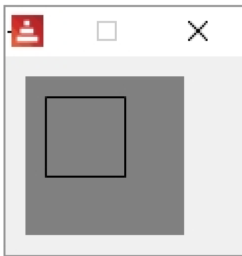
```
Red [needs: view]
```

```
view [
        base draw [triangle 10x10 50x50 50x10]
]
```



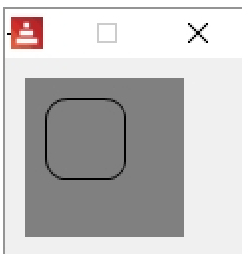
DRAW box

```
Red [needs: view]
view [
        base draw [box  10x10       50x50]
      ;             top-left    bottom-right
]
```



se zaobleným rohem:

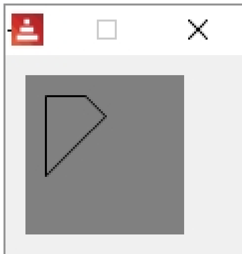
```
Red [needs: view]
view [
        base draw [box     10x10         50x50       10]
;                         top-left  bottom-right   corner-radius
]
```



DRAW polygon

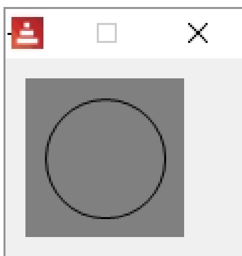
```
Red [needs: view]
view [
```

```
        base draw [polygon 10x10 30x10 40x20 30x30 10x50]
        ; polygon se uzav e automaticky
]
```



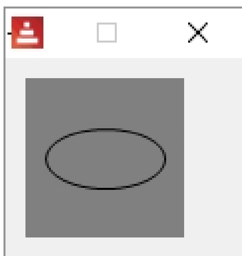
DRAW circle

```
Red [needs: view]
view [
        base draw [circle  40x40     30]
                      ;      center    radius
]
```



elipsa p íkazem circle:

```
Red [needs: view]
view [
        base draw [circle  40x40     30          15 ]
                      ;     center   radius-x   radius-y
]
```

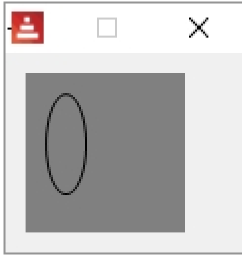


DRAW ellipse

Elipsa se kreslí do pomyslného obdélníku. Zadávají se jeho body *top-left* a *bottom-right*.

```
Red [needs: view]
view [
        base draw [ellipse 10x10 20x50]
```

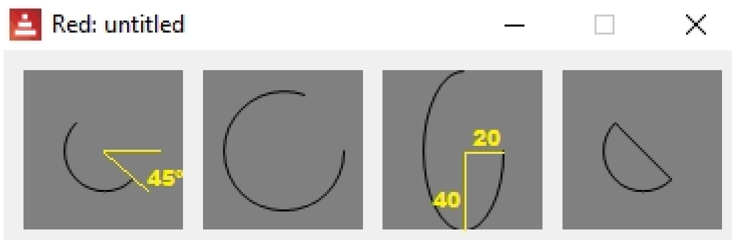
```
]
```



DRAW arc

Kreslí kruhový oblouk ze zadaného st edu (pair!) a se zadanými polom ry (rovn ž pair!). Oblouk je vymezen dv ma úhly ve stupních. První úhel se m í od nuly, druhý úhel od prvního. Lze použít klí ové slovo `closed` pro oblouk, uzav ený spojnicemi konc oblouk s jeho st edem.

```
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 k ivku ze 3 nebo 4 bod :

- 3 body: 2 koncové, 1 kontrolní - kvadratická Bezierova k ivka
- 4 body: 2 koncové, 2 kontrolní - kubická Bezierova. k ivka

```
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 k ivku ze 4 bod : po áte ní 1. kontrolní 2. kontrolní
koncový:
      base draw [curve 10x60 25x15 40x15 70x60]
```

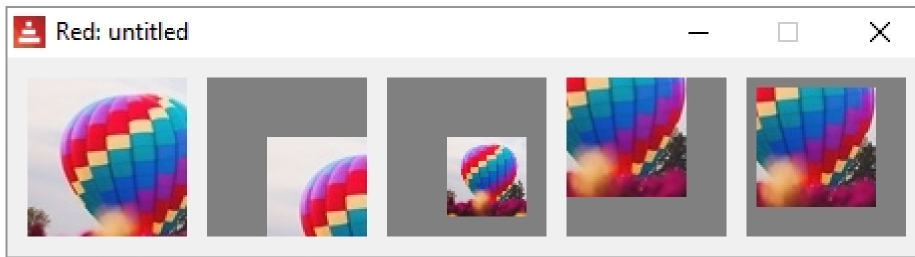
```
        ;poté k ivku ze 3 bod : po áte ní kontrolní koncový:
        base draw [curve 10x60 25x15       70x60]
]
```

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]
]
```

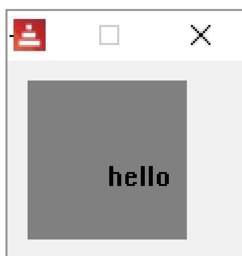


There is also a color command (key color to be made transparent) and a border command, but I couldn't make those work yet.

```
;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

Vyb r fontu pro zobrazení textu

native! Funkce compose

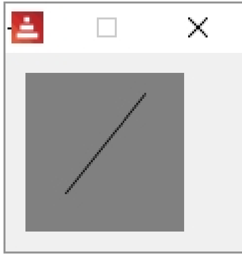
Draw neprovádí vyhodnocení výraz . Následující skript evokuje chybové hlášení:

```
Red [needs: view]
view [
        base draw [line 60x10 (2 * 10x30)]
]
```

Vyhodnotit výraz nám pom že funkce `compose`, nejlépe s up esn ním `/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

Vlastnosti áry

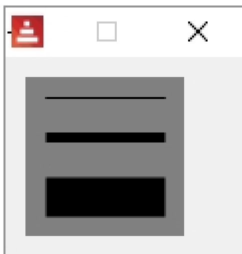
Krom geometrické konfigurace lze pro áru (line) ur it její tloušťku (line-width), zp sob spojení (line-join), zp sob jejího ukon ení (line-cap) a její vyhlazení (anti-alias).

Její barevné a jiné vlastnosti se zadávají prost ednictvím nástroje pen a pen-fill - viz další kapitola.

DRAW line-width

Nastaví novou tloušťku (width) ar.

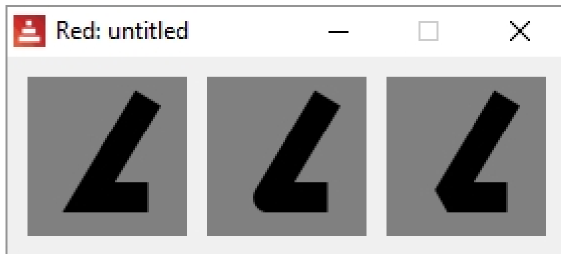
```
Red [needs: view]

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 zp sob spojování ar p i kreslení. P ípustné 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
    ] ]
```



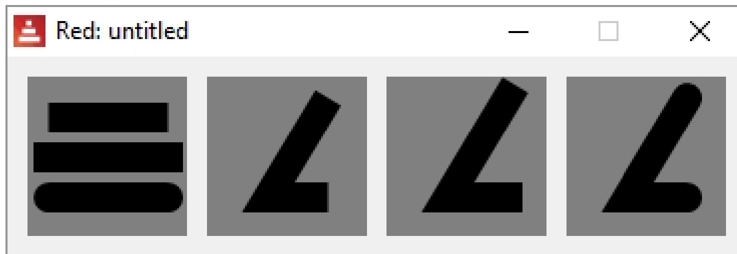
miter-bevel usekne špi ku tvaru miter na tvar bevel, p esáhne-li její délka stanovenou mez - obvykle desetinásobek tloušťky ar.

DRAW line-cap

Definuje tvar ukon ení áry. 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 [
```

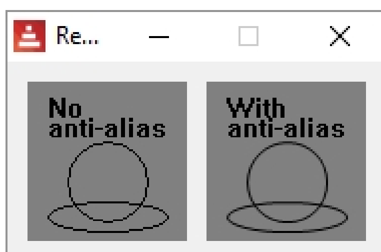
```
            line-width 15
            line-cap square
            line 60x10 30x60 60x60
        ]
        base draw [
            line-width 15
            line-cap round
            line 60x10 30x60 60x60
        ]
]
```



DRAW anti-alias

Vyhlazení zubatych obrys (anti-alias) dává pohledn jší 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 po áte ní nastavení
        text 10x5 "With"
        text 10x15 "anti-alias"
        circle 40x50 20
        ellipse 10x60 60x15
] ]
```



DRAW **pen** < color, (linear, radial, diamond) gradient, pattern, bitmap >

P íkaz ur uje barevné provedení áry.

DRAW **fill-pen** < color, (linear, radial, diamond) gradient, pattern, bitmap >

P íkaz definuje výpl ový režim pro uzav ené tvary.

Created with the Standard Edition of HelpNDoc: Full-featured Kindle eBooks generator

Barva, gradienty a vzory

Barevné a jiné vlastnosti áry se zadávají prost ednictvím p íkaz pen a pen-fill.

DRAW **pen** < color, (linear, radial, diamond) gradient, pattern, bitmap >

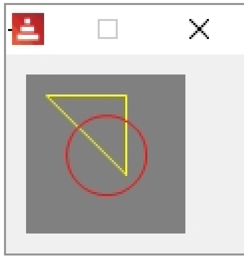
P íkaz ur uje barevné provedení áry.

DRAW **fill-pen** < color, (linear, radial, diamond) gradient, pattern, bitmap >

P íkaz definuje výpl ový režim pro uzav ené tvary.

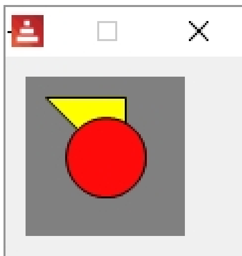
DRAW **pen** <color>

```
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
      ] ]
```



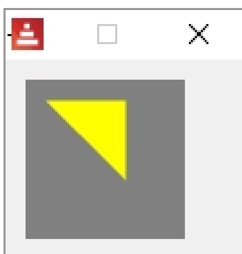
DRAW **fill-pen** <color>

```
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

P evzato z [dokumentace Redu]

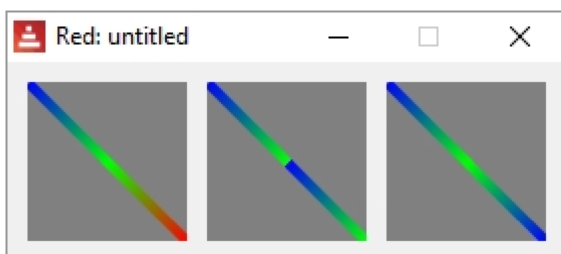
```
<pen/fill-pen> linear <color1><offset> ...
<colorN><offset><start><end><spread>

<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` (aktuáln je `pad` totéž jako `repeat` pro Windows).

Je-li použita metoda spread, definují koncové body sm r gradientu. Nejsou-li body zadány, rozvíjí se gradient v horizontálním sm ru uvnit 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

P evzato z [dokumentace Redu]

```
<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` (aktuáln je `pad` totéž jako `repeat` ve Windows).

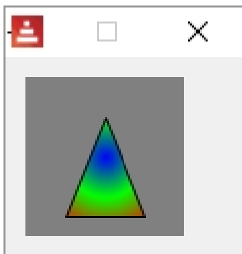
Radiální gradient bude vykreslován od ohniska k okraji kružnice, definované st edem a polom rem. Po áte ní barva se vykreslí v ohnisku, koncová na okraji kruhu.

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

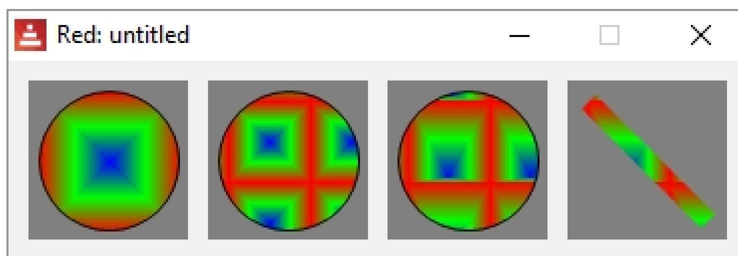
P evzato z [dokumentace Redu]

```
<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` (aktuáln   je `pad` totéž jako `repeat` ve Windows).

Kárový gradient se vykreslí ve sm   ru od ohniska k okraji kára, definovaného horním a dolním rohem. Po áte ní 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
st edu
            circle 40x40 35
      ]
      base draw [
            fill-pen diamond  blue green red 10x10 50x50  ;sou adnice
gradientu "box"
            circle 40x40 35
      ]
      base draw [
            fill-pen diamond  blue green red 10x10 50x50 30x48  ;
p idané 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** - výpl   vzorkem

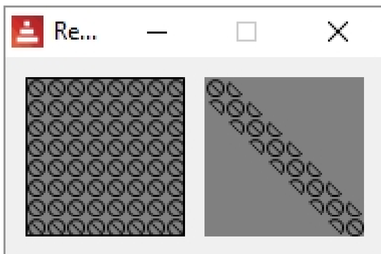
P evzato z [dokumentace Redu]

```
<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 vypln ný obdélník:
      base draw [
            fill-pen pattern 10x10 [
                  circle 5x5 4
                  line 3x3 7x7
            ] box 0x0 79x79
            ]
            ; potom vypln nou diagonálu:
      base draw [
            pen pattern 10x10 [
                  circle 5x5 4
                  line 3x3 7x7
            ]
            line-width 15
            line 0x0 79x79
            ] ]
```



DRAW **bitmap** - výpl obrázkem

P evzato z [dokumentace Redu]()

```
<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!).
```

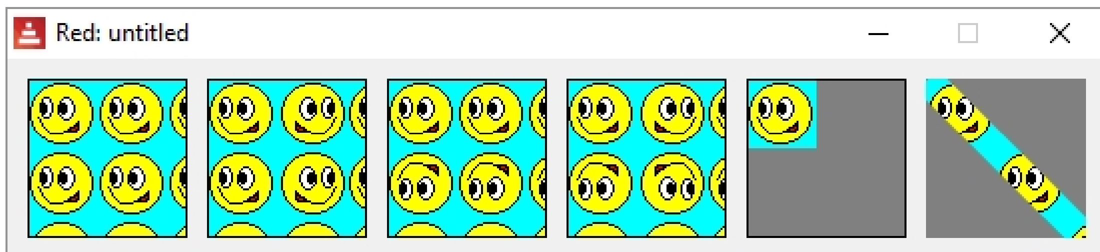
Nastaví obrázek jako vzorek pro výpl ové operace. V režimu mode jsou akceptovány následující hodnoty: `tile` (default), `flip-x`, `flip-y`, `flip-xy`, `clamp` (p ipnout).

Výchozím bodem vypl ovaného obdélníku (box) je 0x0 a koncovým bodem je velikost obrázku - oboje typu pair!.

Výpl ovým obrázkem pro následující p íklad 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  ; zrcadlov  m nit podél osy
x
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-y  ; zrcadlov  m nit podél osy
y
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-xy ; zrcadlov  m nit podél os x
i y
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage clamp   ; p ipnout jako na nást nku
            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

Transformace v rovin

Popisy p íkaz push, rotate, scale, translate, skew a transform jsou p evzaty z dokumentace Redu.

DRAW push nutno doplnit

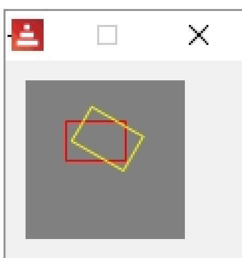
DRAW rotate

```
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 pravoto ivou rotaci ve stupních kolem daného bodu. Není-li zadán nepovinný parametr `center`, provede se rotace kolem po átku aktuálního sou adného systému. Negativní hodnoty lze použít pro levoto ivou (counter-clockwise) rotaci. Je-li jako poslední argument zadán blok, aplikuje se rotace pouze na p íkazy v bloku.

P i 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 neotá ený
            rotate 30 40x40     ; úhel oto ení, st ed otá ení
            pen yellow
            box 20x20 50x40     ; obdélník otá ený
            ] ]
```



DRAW scale

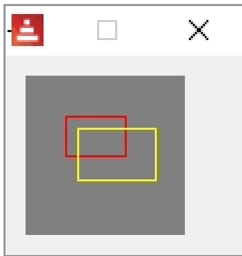
```
scale <scale-x> <scale-y> [<commands>]
scale 'pen <scale-x> <scale-y>
scale 'fill-pen <scale-x> <scale-y>

<scale-x>  : měřítko pro osu X (number!).
```

```
<scale-y>  : měřítko pro osu Y (number!).
<commands> : (optional) příkazy dialektu Draw.
```

Nastaví velikost zv tšení. Zadané hodnoty jsou násobitelé; hodnoty v tší než jedna zv tšují, hodnoty menší než jedna zmenšují. Je-li jako poslední argument zadán blok, aplikuje se zm na velikosti pouze na p íkazy v bloku.

P i použití literálových slov (lit-words) `'pen` nebo `'fill-pen` se zm na velikosti aplikuje na aktuální pen nebo fill-pen.

```
Red [needs: view]

view [
      base draw [
           pen red
            box 20x20 50x40    ; obdélník nezv tšovaný
            scale 1.3 1.3      ; zv tšení o30% v obou sm rech
            pen yellow
            box 20x20 50x40    ; obdélník zv tšovaný
            ] ]
```



DRAW translate

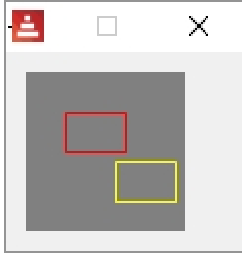
```
translate <offset> [<commands>]
translate 'pen <offset>
translate 'fill-pen <offset>

<offset> : velikost posunu (pair!).
```

Nastaví po átek pro následné kreslicí p íkazy. Víceré p íkazy `translate` mají kumulativní ú inek. Je-li blok zadán jako poslední argument, použijí se posuny pouze u p íkaz z tohoto bloku.

P i použití literálových slov (lit-words) `'pen` nebo `'fill-pen` se posunutí aplikuje na aktuální pen nebo fill-pen.

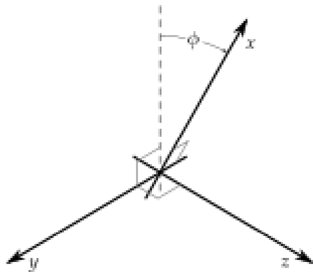
```
Red [needs: view]

view [
      base draw [
```

```
pen red
 box 20x20 50x40      ; neposouvaný obdélník
 translate 25x25
 pen yellow
 box 20x20 50x40      ; posouvaný obdélník
 ] ]
```



DRAW skew - zkosení

Zkosený sou adnicový systém je tehdy, když jeho osy nejsou ortogonální.



P íkaz skew nato í osu x nebo osu y o zadaný po et stup .

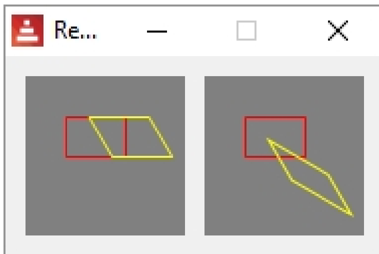
```
skew <skew-x> <skew-y> [<commands>]
skew 'pen <skew-x> <skew-y>
skew 'fill-pen <skew-x> <skew-y>

<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 sou adných os, zadané úhlem pooto ení ve stupních. Není-li zadáno `<skew-y>`, p edpokládá se, že je nulové. Je-li blok zadán jako poslední argument, použije se zkosení pouze u p íkaz z tohoto bloku.

P i 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 sm ru osy x
      pen yellow
      box 20x20 50x40    ; zkosený obdélník
      ]
base draw [
     pen red
      box 20x20 50x40    ; nezkosený obdélník
      skew 30 30         ; zkosení 30° ve sm ru obou os
      pen yellow
      box 20x20 50x40    ; zkosený obdélník
      ] ]
```



DRAW transform

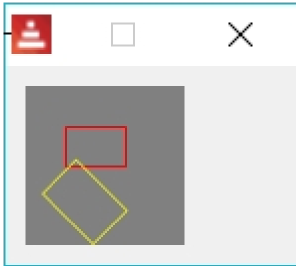
```
transform <center> <angle> <scale-x> <scale-y> <translation>
[<commands>]
transform 'pen <center> <angle> <scale-x> <scale-y> <translation>
transform 'fill-pen <center> <angle> <scale-x> <scale-y>
<translation>

<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 pooto ení, zm na velikosti (scaling) a posun jedním p íkazem. Je-li blok zadán jako poslední argument, použije se transformace pouze u p íkaz z tohoto bloku.

P i 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     ;st ed rotace, úhel, m  ítka,
posuny
      pen yellow
       box 20x20 50x40            ; transformovaný obdélník
       ] ]
```

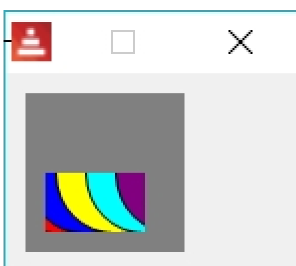


DRAW clip

Ur uje pravoúhlý vý ez definovaný dv ma body (start, end) nebo libovoln tvarovaný vý ez, definovaný blokem p íkaz sub-dialektu Shape. Takto ur ený vý ez je platný pro všechny následné p íkazy dialektu Draw. Je-li blok zadán jako poslední argument, použije se vý ez pouze u p íkaz 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 vý ezu se stávajícím m že být zadán jako jeden z následujících zp sob :

- `replace` (default)
- `intersect`
- `union`
- `xor`
- `exclude`

Created with the Standard Edition of HelpNDoc: Full-featured EBook editor

Sub-dialekt Shape

Sub-dialekt Shape umož uje vytvá et tvary (kresby) jako bloky. Použití dialektu shape má tuto skladbu:

shape (p íkazy dialektu Shape)

Specifické rysy tohoto dialektu jsou:

- každý kreslící p íkaz za íná na aktuální pozici pera
- pozici pera lze m nit bez kreslení zadáním cílové polohy p íkazem **move**
- tvary jsou automaticky uzavírány - spojením posledního bodu s prvním
- generované tvary lze zadat do p íkazu **fill-pen** pro vytvá ení individuálních výplní
- kreslící p íkazy používají implicitn absolutní (relativní k piškotu) sou adnice; relativní (relativní k poslední pozici) sou adnice jsou aktivní ve verzi 'lit-word p íkazu.

V argumentu funkce **shape** m žete použít tyto p íkazy a vlastnosti:

move, arc, curve, curv, qcurve, qcurv, hline, vline, line, line-width, line-join, line-cap, pen, fill-pen.

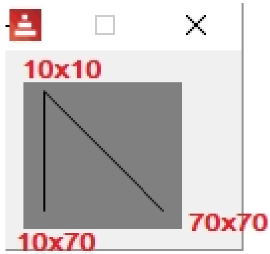
Mod e ozna ené entity se deklarují stejn jako v dialektu Draw.

move

P emístí pero na novou pozici, nic se nekreslí:

```
Red [needs: view]

myshape: [
        line 10x10 70x70   ; line from 10x10 to 70x70
```

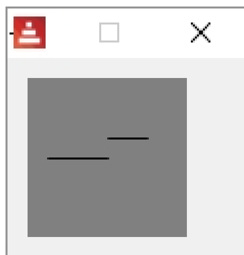
```
        move 10x70          ; p emístí pero na pozici 10x70
        line 10x10          ; kreslí  áru z aktuální (10x70) pozice
      ]

view compose/deep/only [base draw [shape (myshape)
      ] ]
```



Relativní pozice

Sou adnice se stanou relativní, p idá-li se apostrof p ed p íkaz:

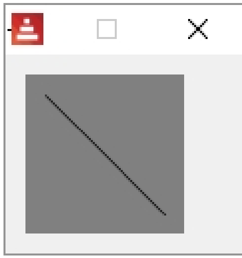
```
Red [needs: view]

myshape: [
        line 10x40 40x40   ; horizontální  ára do st edu
        'move 0x-10        ; posun aktuální pozice vzh ru
        'line 20x0         ; konec  áry relativn  k aktuální pozici
      ]

view compose/deep/only [base draw [shape (myshape)]]
```



line

Nakreslí úse ku mezi dv ma body. Je-li zadáno více bod , kreslí se další úse ky, napojované na p edchozí.

```
Red [needs: view]

myshape: [line 10x10 70x70]

view compose/deep/only [
        base draw [
            shape (myshape)
        ] ]
```



Všimn te si up esn ní `compose/deep/only` a závorek kolem `myshape`. Tyto náležitosti jsou z ejm p i práci s DSL Shape povinné.

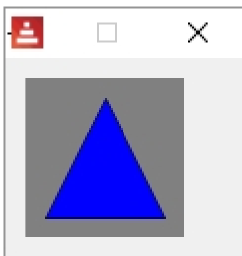
Up esn ní /deep p ikazuje vyhodnotit zano enou závorku.

Up esn ní /only p ikazuje ... nutno doplnit.

Automatické uzavírání tvaru

V následujícím p íkladu jsou explicitn zadány pouze dv p ímky. P íkaz `fill-pen` byl p idán jenom pro zvýrazn ní uzav ené plochy:

```
Red [needs: view]

myshape: [
      line 10x70 40x10 70x70
]
view compose/deep/only [base draw [ fill-pen blue shape (myshape)]]
```

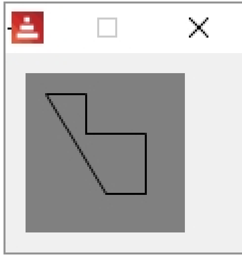


hline a vline

Kreslí horizontální nebo vertikální p ímku 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 ára se posléze uzav e
]
```

```
view compose/deep/only [base draw [shape (myshape)]]
```

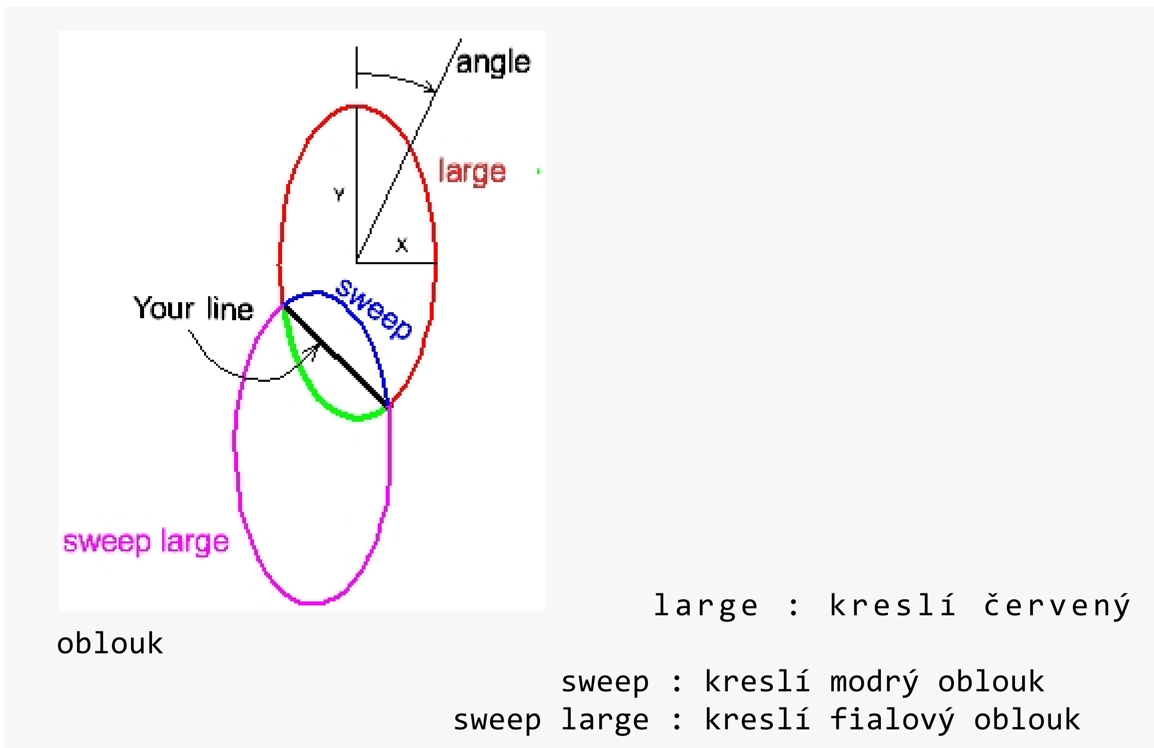


arc (liší se od Draw-arc)

```
 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 polom ry r-x, r-y a úhlem odklonu vodorovné osy elipsy od sou adnicové 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 dopl ku sweep, large kreslí zelený oblouk. Oklon elipsy v obrázku je 0 stup .

Všimn te si, že zadáváte pouze koncovou pozici oblouku. Po áte ní pozice je daná aktuální polohou pera. Je-li tedy **arc** prvním p íkazem argumentu pro **shape,** musíte se nejprve p esunout (`move`) do po áte ní pozice.

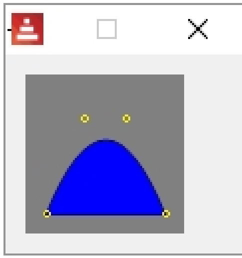
```
Red [needs: view]

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 vytvo í eliptický oblouk.

curve

Kreslí kubickou Bezierovu k ivku, danou t emi až ty mi body. P ipouští absolutní nebo relativní (`'curve`) sou adnice.

```
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)
      ] ]
```



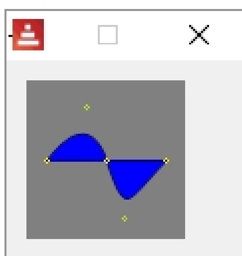
curv , qcurve a qcurv

Všechny t i p íkazy generují Bezierovy k ivky s absolutními i relativními (') sou adnicemi.

- **qcurv** kreslí hladkou kvadratickou Bezierovu k ivku z 1+1 bod .
- **qcurve** kreslí kvadratickou Bezierovu k ivku z alespo 1+2 bod ;
- **curv** kreslí hladkou kubickou Bezierovu k ivku z alespo 1+1 bod ;

Tyto k ivky používají po áte ní, koncový a kontrolní bod i body. P kná animace je na Wikipedii, viz 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

Parse

Parse je vno ený jazyk (DSL - Domain Specific Language) pro parsování vstupních ad (series) s použitím gramatických pravidel tohoto jazyka. Používá se pro ov ení shody,

vyjímání a úpravu vstupních dat. Ve všech p ípadech se používá funkce **parse** v této skladb :

```
parse < input > < rules >
```

```
<input> : jakákoli hodnota řady (any-string!, any-block!, binary!)
<rules> : blok s pravidly
```

Parsování za íná na za átku vstupní ady. P i úsp šné konfrontaci s pravidly (success) postupuje k dalšímu lenu ady. Parsování kon í výskytem neúsp chu a operace vrací FALSE nebo úsp šným pr chodem celou vstupní adou a operace vrací TRUE.

Významnou úlohu p i sestavování gramatických pravidel mají **klí ová slova**, která lze rozd lit do n kolika kategorií:

Ov ení shody:

ahead, end, none, not, quote, skip, thru to

Pr b h 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 p echodu k ov ování dalšího vstupního elementu je úsp šné hodnocení p edchozího. Funkce vrací true, konvenují-li všechny elementy vstupní ady se zadanými pravidly, p ípadn vrací false, pokud tomu tak není.

Nejjednodušším p íkladem je prostá kontrola, zda každý element vstupního bloku je roven svému prot jšku 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 v tší p ehlednost popisu parse, p epišme výše uvedený p íklad do jiného formátu. Kontrolní blok se formáln   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 úsp šné, a došli jsme na konec  ady
; 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 b žný 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"]          ; alternativn  použitelné hodnoty
      "elk"
]
print parse b [
      "fox"
      ["rat" | "owl"| "cat" | "whatever"]
      "elk"
]
```

```
true
true
```

Obsah vstupního bloku lze upravit zadaným vypušt ním element :

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
    4 skip            ; p ikázaný skok p íkazem "4 skip"
      "elk"
      "cat"
]
```

```
true
```

```
Red[]

a:  ["rat" "rat" "rat" "rat" "elk" "cat"]

print parse a [
    4 "rat"             ; po et stejných prvk
      "elk"
      "cat"
]
```

```
true
```

```
Red[]

a:  ["rat" "rat" "elk" "cat"]

print parse a [
    0 4 "rat"           ; po et stejných prvk  zadán p ípustným rozsahem
```

```
        "elk"
        "cat"
]
```

```
true
```

Introspekce provád ného pr b hu parse:

```
parse-trace < input > < rules >
```

Up esn ní funkce parse:

/case =>
/part =>
/trace =>

Created with the Standard Edition of HelpNDoc: Easily create Help documents

Matching

Klí ová slova: skip, thru, to, ahead, end, none, not, quote

PARSE skip

Bez argumentu p esko í 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
```

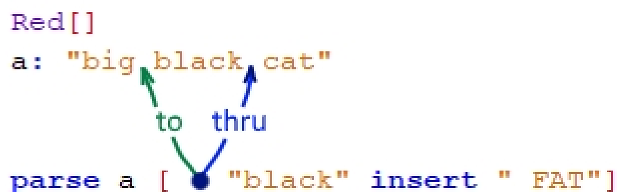
S argumentem p esko í zadaný po et element :

```
print parse "axxb" ["a" 2 skip "b"]
```

```
true
```

PARSE to a thru

P esko í elementy až k ozna enému místu vstupního bloku. Slovo **to** nastaví po átek parsování p ed zadaný element, slovo thru nastaví po átek parsování za zadaný element.



PARSE to

```
Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat" "bat"]  ; input block

print parse a [

    to "elk"               ; p esko í všechny elementy p ed "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"   ; p esko í všechny elementy v etn  "elk"

   "cat" "bat"   ; zbytek je konformí s pravidlem
]
```

```
true
```

Klí ové slovo thru je prosp šné, i když procedura skon í hodnocením false:

```
Red[]

s: "dnes je nádherný den"
parse s [thru "nádherný" change "" " a slunný"]
```

```
false
```

```
s
== "dnes je nádherný a slunný den"
```

K   emu zde došlo:

Vstupní et zec byl konfrontován s p íkazem ke skoku za slovo "nádherný", jehož následná mezera byla nahrazena dopl kem " 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 skute nosti je to mnohem složit jší - viz Ungaretti # while - ahead

PARSE end

Vrací true, jestliže byly pozitivn 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** ozna uje pozici na konci každé sekvence. Interpret si jej dosadí sám, pokud není uveden.

PARSE none

Vždycky vrací true. Je to pravidlo pro všechny p ípady

```
Red[]

a:  ["fox" "dog" "owl" "rat"]

print parse a ["fox" "dog"
         none (print "Servus")        ; p íkaz 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" p idat
```

```
true
```

```
print parse a ["fox" "dog" not "owl" skip "rat"]    ; museli jsme "owl"
p esko it
```

```
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

Iterace

Klí ová slova: some, any, opt, while.

Aplikace pravidla (rule) m že být nepovinná nebo r zným zp sobem opakovaná.

Klí ové slovo i 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 i vícekrát

any	opakovat pravidlo nula i vícekrát
opt	shoda s pravidlem žádná nebo jedna

Ur itý po et opakování - p íklady

```
>> parse "fogfogfog" [3 "fog"]          ; určeno přesně
== true
```

```
>> parse "fogfogfog" [0 5 "fog"]        ; určeno rozsahem
== true
```

Neur itý po et opakování - p íklady

PARSE some, PARSE any, PARSE opt

Aplikuje pravidlo následného vno eného bloku (jednou i vícekrát, nikdy nebo alespo 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 alespo  jednou
         "cat" "owl"  ]
true

parse a [   any ["elk" "cat"]           ; nikdy nebo alespo  jednou
         "owl"  ]
```

```
true

parse a [    any ["fig"]                ; nikdy nebo alespo   jednou
         "elk" "cat" "owl"   ]
true

parse a [    any ["fig"]                ; nikdy nebo alespo   jednou
         "elk" "owl"   ]
false                                  ; nikdy se vztahuje jen na
subpravidlo any
```

```
parse a [    opt ["fig"]                ; nikdy nebo alespo   jednou
          "elk" "cat" "owl"   ]
true

parse a [    opt ["elk" "cat"]          ; nikdy nebo alespo   jednou
         "owl" ]
true

parse a [    opt ["elk" "owl"]          ; nikdy nebo alespo   jednou
         "cat" ]
false *
```

* Pokud se vstup neshoduje s pravidlem **opt**, p esko í parse toto pravidlo a kontroluje tentýž vstup následným pravidlem.

Jiný p íklad pro **opt**:

```
hd: "horskádráha"                      ;  et zec s 11 elementy

 parse hd [opt "horská" "dráha"]      ; == true

 parse hd [opt "horská" "práva"]      ; == false
```

PARSE **while**

Podobn   jako **any**, aplikuje pravidlo následného vno eného bloku nikdy nebo alespo jednou, dokud neselže; nestará se o p ípadnou zm nu vstupu:

```
a:  ["elk" "cat"]

print parse a [ any ["fox" "dog"] "elk" "cat" ]

true


print parse a [ while ["fox" "dog"] "elk" "cat" ]

true
```

Ve skute nosti je to mnohem složit jší - viz Ungaretti # while.

Created with the Standard Edition of HelpNDoc: Easy EBook and documentation generator

Extrakce

Klí ová slova: set, copy, keep, collect, collect set, collect into.

P ipomínka: P íkaz "print" provádí úpravu výstupu. Sb rný blok ["fox" "rat"] nap íklad, upraví na fox rat.

PARSE set , PARSE copy

V obou p ípadech p i adí 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 klí ovým slovem **any** se výstupy liší. P íkaz **set** vrací pouze první platnou hodnotu, p íkaz **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 klí ovým slovem **collect**, **collect set** a **collect into**. P ipojí do sb rného bloku kopii první hodnoty uspokojené regule.

PARSE collect

Vytvo í sb rný blok, zapln ný p íkazem **keep.** P íkaz parse zde nevrací *true* i *false.*

```
a:  ["fox" "dog" "owl" "rat" "elk" "cat"]        ; vstupní blok

print parse a [
```

```
collect[
keep "fox"     ; success, pat í do sb rného bloku
"dog" "owl"     ; success, ale nepat í do sb rného bloku
keep "rat"      ; success, pat í do sb rného bloku
keep "cow"     ; fail, proto nepat í do sb rného bloku
"cat"   ] ]          ; success, ale nepat í do sb rného bloku
```

```
fox rat
```

PARSE collect set

Na rozdíl od **collect** vrací true i false. Vytvo ený sb rný blok p i adí zadané prom nné (zde b).

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]        ; input block

 print parse a [                       ; bez "print" nevrátí "false"
     collect set b [                   ; vytvo í sb rný 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 i false. Vytvo ený sb rný blok p i adí p edem p ipravené prom nné (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 výb r syntaxí set-word

P i parsování lze ke zbytku vstupu p i adit prom nnou, by je výsledek negativní:

```
Red[]

a: ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [ "fox" "dog" b: ]     ; p i adí zbytek slovu b
probe b
```

```
false
["owl" "rat" "elk" "cat"]
```

Created with the Standard Edition of HelpNDoc: Full-featured EBook editor

Úprava vstupu

Klí ová 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 p íklad:

```
b: "Tvé velké o i"
parse b [thru "velké" insert " hn dé"]          ; mezera!
```

```
false
```

```
print b
```

```
Tvé velké hnědé oči
```

PARSE remove

Odebere shodující se ozna ené elementy ze vstupu.

```
a: ["fox" "dog" "owl" "rat"]
parse a [    "fox" remove "dog"        remove "owl" "rat"]
```

```
true
```

```
print a
```

```
fox rat
```

Jiný p íklad:

```
a: "Tvé velké o i"
parse a [ to "velké" remove "velké"]          ; == false
print a
```

```
Tvé oči
```

PARSE change

M ní ozna enou 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

Control flow

Control flow:

Klí ová 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 výb ru alternativnímu pravidla:

```
                true
if (logic test) [rule1 | rule2] rule rule ...
                false
```

```
>> 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

Introspekce

Introspekce:

Pr b h funkce **parse** lze zobrazit pomocí funkce **parse-trace**.

PARSE parse-trace

Provede parsování vstupu a rovn ž vytiskne pr b h 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
```

Pr b h procesu parsování lze také zjistit vhodným vložením p íkaz **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