

Lexer

1. Koncept	1
2. Načítání hodnot	1
2.1. <code>transcode</code>	1
3. Skenování hodnot	2
3.1. <code>scan</code>	2
4. Postupné zpracování hodnot	3
4.1. <code>load/next</code>	3
4.2. <code>transcode/next</code>	3
4.3. <code>scan/next</code>	4
5. Instrumentace lexeru	4
5.1. Schema práce lexeru	4
5.2. <code>transcode/trace</code>	4
6. Literálová skladba hodnot	7
6.1. Hrubé řetězce	7

1. Koncept

Lexer je ta část knihovny pro runtime, která zajišťuje konverzi hodnot z textové prezentace na datovou strukturu, ukládanou v paměti - ve shodě se syntaktickými pravidly Redu. Tento proces se nazývá *loading* (načítání).

Poznámka: kompilátor Redu také obsahuje dočasnou implementaci lexeru s omezenými schopnostmi ve srovnání s runtime verzí. Od této implementace bude upoštěno, jakmile se Red stane *self-hosted*.

2. Načítání hodnot

2.1. `transcode`

Syntaxe

```
transcode <input>
```

<input> : načítatná řada (binary! string!).

Vrací : jedinou hodnotu nebo blok hodnot.

Popis

Konvertuje vstupní binární řady v kódování UTF-8 na interně použitelné hodnoty ve shodě se

syntaktickými pravidly. Neshoda s těmito pravidly má za následek vyvolání chybového hlášení. Řetězcové řady (any-string!) jsou před dalším zpracováním nejprve interně konvertovány na binární řadu UTF-8.

3. Skenování hodnot

3.1. scan

Syntaxe

```
scan <input>
scan/fast <input>

<input> : skenovaná řada (binary! string!).

Vrací : rozpoznaný datový typ (datatype!).
```

Popis

Skenuje řetězec nebo binární obsah bufferu z hlediska skladebných pravidel. Zastaví se, jakmile je rozpoznána první hodnota, načež vrací její datový typ. Nenačítá se žádná hodnota, tudíž není alokována žádná paměť. Není-li skenovaný vstup platný, je bez výjimky vracena hodnota typu **error!**.

Upřesnění **/fast** poskytuje rychlost na úkor přesnosti. Skenování vrací *nejpravděpodobnější* typ (bez hlubšího ověřování) s použitím interní předskenovací procedury. V tomto případě se provádí jen velmi málo ověřování chyb. Případy, kdy se odhadovaný typ může lišit od skutečného typu jsou uvedeny v následující tabulce:

Odhadovaný typ	Skutečný typ
integer!	integer! float! (1)
word!	word! lit-word! get-word! set-word!
refinement!	refinement! word! lit-word! get-word! set-word! (2)

(1) Většinou kvůli převodu hodnot typu integer! na float! u celých čísel mimo rozsah (range).

(2) Možné typy slov jsou evokovány sekvencemi znaků s jedním či více lomítky (např: '/', '/*', '//,...).

Poznámka:

- Řetězcové řady (any-string!) jsou nejprve před zpracováním interně převedeny na binární řadu UTF-8.
- Skenování je mnohem rychlejší než načítání, neboť neprovádí alokaci žádné další paměti.
- Skenování je pomalejší než předskenování, protože provádí důkladné ověřování.

4. Postupné zpracování hodnot

4.1. load/next

Syntaxe

```
load/next <input> <var>
```

<input> : načítaná řada (binary! string!).

<var> : slovo, k němuž je přiřazena zbývající část vstupu za první načtenou hodnotou (word!).

Vrací : první načtenou hodnotu.

Popis

Načítá a vrací první hodnotu ze vstupního textu. Slovo argumentu je nastaveno na zbývající část vstupního textu.

4.2. transcode/next

Syntaxe

```
transcode/next <input>
```

<input> : načítaná řada (binary! string!).

Vrací : blok s první načtenou hodnotou a zbytkem vstupu.

Popis

Načítá první hodnotu ze vstupního textu. Vrací blok, který obsahuje:

- první načtenou hodnotu ze vstupu (any-type!)
- zbývající část vstupu za načtenou hodnotou (binary! string!)

4.3. scan/next

Syntaxe

```
scan/next <input>
```

<input> : načítaná řada (binary! string!).

Vrací : blok s typem první hodnoty a zbytkem vstupu.

Popis

Skenuje první hodnotu vstupního textu. Vrací blok, který obsahuje:

- datový typ první hodnoty ve vstupu (datatype!)
- zbývající část vstupu za skenovanou hodnotou (binary! string!)

5. Instrumentace lexeru

5.1. Schema práce lexeru

Proces tokenizace je rozdělen do jednotlivých stupňů, v nichž jsou spouštěny události tam, kde může být invokována uživatelem poskytnutá funkce se zpětným voláním (callback). Jednotlivými stupni jsou:

```
      +-> ERROR
      /
    +-> CLOSE series
    /
  +-> OPEN series
  /
-> PRESCAN token -> SCAN token -> LOAD value
    \           \           \
  +-> ERROR      +-> ERROR      +-> ERROR
```

Událostmi lexeru tedy jsou: **prescan**, **scan**, **load**, **open**, **close**, **error**.

5.2. transcode/trace

Syntaxe

```
transcode/trace <input> <callback>
```

<input> : načítaná řada (binary! string!).

<callback> : zpětná (callback) funkce pro ošetření události lexeru (function!).

Vrací : jedinou hodnotu nebo blok hodnot.

Popis

Konvertuje vstupní binární řadu v kódování UTF-8 na interně použitelné hodnoty ve shodě se syntaktickými pravidly. Při každé události lexeru volá uživatelem poskytnutou callback funkci.

Blok specifikací callback funkce:

```
func [  
    event [word!]                ;-- aktuální stav lexeru (viz tabulka níže)  
    input [string! binary!]      ;-- odkaz na vstupní řadu v aktuální pozici načítání  
(může být změněno)  
    type [datatype! word!]       ;-- slovo nebo datový typ popisující typ tokenu nebo  
aktuálně řešenou hodnotu  
    line [integer!]              ;-- číslo řady aktuálního vstupu  
    token                        ;-- aktuální token jako úsek (slice) vstupu typu  
pair! nebo načítaná hodnota  
    return: [logic!]  
][  
    [events]                     ;-- volitelný seznam vymezených názvů událostí  
    ...body...  
]
```

Odsazení (offset) argumentu **vstupní** řady je dáno místem, kde se lexer zastavil po detekci konce tokenu. Tento offset může být modifikován callback funkcí. Je-li **chybová** událost ignorována, nedojde automaticky k pokročení vstupem; záleží na callback funkci aby nastavila **vstupní** řadu do správné pozice pro opětovné ošetření řady. Pokud se nezadaří, může to vést k nekonečné smyčce.

Při **chybové** události obsahuje argument **type** název datového typu, jenž byl částečně rozpoznán. Vyskytne-li se chyba u izolovaného znaku, jako jsou nepárové koncové závorky **)] }**, je argument **type** nastaven na **error!**, protože v takovém případě nebyl rozpoznán žádný určitý typ.

Blok těla funkce může začínat volitelným filtrovacím blokem k určení události, která má být spuštěna. To umožňuje redukovat počet zpětných (callback) volání, což se projeví lepším výkonem zpracování.

Význam některých argumentů a zpětných hodnot *závisí* na typu události. Následující tabulka obsahuje možné kombinace a účinky:

Event	Type	Token	Return Value	Description
prescan	word! datatype!	pair!	true : scan false : drop	Byl-li rozpoznán token.
scan	word! datatype!	pair!	true : load false : drop	Byl-li přesně rozpoznán typ tokenu.
load	datatype!	<value>	true : store false : drop	Byl-li token konvertován na hodnotu Redu.
open	datatype!	pair!	true : open false : drop	Byl-li otevřen nový block!, paren!, path!, map! nebo víceřádkový řetězec.
close	datatype!	pair!	true : close false : drop	Byl-li zavřen nový block!, paren!, path!, map! nebo je víceřádkový řetězec uzavřený.
error	datatype!	pair!	true : throw false : ignore	Při výskytu syntaktické chyby.

Možné hodnoty pole **type** (word! nebo datatype!) v události **scan**:

```
eof comment hex error! block! paren! string! map! path! word! refinement!
issue! file! binary! char! percent! integer! float! tuple! date! pair! time!
money! tag! url! email! ref! lit-word! get-word! set-word!
```

Možné hodnoty pole **type** (datatype!) v události **open**:

```
block! paren! string!(1) map! path! lit-path! get-path!
```

Možné hodnoty pole **type** (datatype!) v události **close**:

```
block! paren! string!(1) map! path! lit-path! get-path! set-path!
```

(1): pouze u řetězců, vymezených složenými závorkami.

Poznámky:

- Je-li při události **prescan** vráceno **false**, jsou příslušné události **scan** a **load** přeskočeny.
- Je-li při události **scan** vráceno **false**, je příslušná událost **load** přeskočena.
- Je-li upuštěno od události **open**, mělo by být upuštěno rovněž od události **close**.

Viz příklady na <https://github.com/red/code/tree/master/Scripts/lexer>

6. Literálová skladba hodnot

6.1. Hrubé řetězce

Řetězce v Redu mají pro některé znaky speciální pravidla, jako je např. použití znaku `^` coby únikového mechanismu nebo nezbytnost vybalancovat vnitřní složené závorky u řetězců, vymezených složenými závorkami.

Formát hrubého (raw) řetězce umožňuje vložit literálové řetězce bez jakéhokoliv ošetření jejich obsahu.

Syntaxe

```
%{...}%  
%%{...}%%  
%%%{...}%%%  
...
```

K vymezení řetězce lze použít libovolný počet párovaných znaků `%`. Není-li počet znaků na začátku shodný s počtem znaků na konci, dojde při načítání k chybovému hlášení.