# Konverzia textoveho objektu do stromu

[TEXT FILE](#Vstupny subor) [1]

[TREE](#Vysledny strom) [3]

[CONFIG FOR READER](#Konfiguracny subor) [2]

Tento subor bude skonvertovany do stromu

[ERROR](#ERROR object logov a chybovych hlasok)

Opis k obrazku:

|  |  |
| --- | --- |
| # | Opis |
| 1 | Tento subor bude skonvertovany |
| 2 | Tento subor obsahuje konfiguraciu pre file reader |
| 3 | Vysledny strom |
| 4 | Konvertor |

## Vstupny subor

Vstupny subor je textovy subor v znakovej sade Unicode

## Transformacny skript

Transformacny skript obsahuje:

* Definicie [charsetov](#_Definicia_charsetu)
* Definicie [patternov](#_Definicia_patternu)
* Hlavna funkcia konverzie a pomocne uzivatelske funkcie

Transformacny skript je napisany v jazyku TREP.

### Definicia charsetu

(SET newline (var charset 'newline'))

(newline addchar '\10' '\13')

Znak \ je escape znak, za nim nasleduje decimalne cislo ukoncene znakom \ alebo medzerou alebo novym riadkom. Ak je za znakom \ znak #, tak za nim nasleduje hexadecimalne cislo

(newline addchar '\x2025' '\x2026')

Inak prikaz chars prijima retazec az do konca riadku a prida do definicie charsetu vsetky znaky az do konca riadku

Dalsia moznost ako definovat znaky v charsete je chargroup. Prijima dva argumenty, prvy a druhy znak, pricom do charsetu pridava vsetky znaky v rozsahu medzi tymito znakmi (vratane tychto znakov).

(SET alphanum (var charset 'alphanum'))

(alphanum addrange 'A' 'Z')

(alphanum addrange 'a' 'z')

(alphanum addrange '0' '9')

(alphanum addrange '\256' '\278')

(alphanum addchar '\x2026' '\x2035')

### Definicia patternu

(SET integer\_literal (var pattern integer\_literal))

Tu je priklad na definovanie charsetov

(set digit (NEW charset 'digit'))

(digit addchar '0123456789')

(SET sign (NEW charset 'sign'))

(sign addchar '+-')

A tu su priklady na definovanie patternov

(SET decpart (NEW PATTERN decpart))

(decpart ADDCHARS 1 1 '.')

(decpart ADDCHARSET 0 1 digit)

(SET float (NEW PATTERN float))

(float ADDCHARSET 0 1 sign)

(float ADDCHARSET 0 MAXINT digit)

(float ADDPATTERN 0 1 decpart)

(SET epart (NEW PATTERN epart))

(epart ADDCHARS 1 1 'E')

(epart ADDCHARSET 0 1 sign)

(epart ADDPATTERN 1 1 integer)

(SET double (NEW PATTERN double))

(double ADDCHARS 0 1 sign)

(double ADDCHARS 0 MAXINT digit)

(double ADDPATTERN 0 1 decpart)

(double ADDPATTERN 1 1 epart)

### Hlavna funkcia

Existuje viac sposobov spracovania suboru:

* char[] -> tree
* char[] -> string[] -> tree

samotna konverzia string[] -> tree moze prebiehat vo viacerych krokoch

* string[] -> tree' -> tree'' -> tree ''' -> … -> tree

samotne pole stringov (string[]) mozeme povazovat za poduzly hlavneho uzlu, takze cely problem konverzie mozeme zovseobecnit ako:

* char[] -> tree1 -> tree2 -> … -> tree

Zakladna struktura hlavnej funkcie je:

(exec

…

<commands>

…

)

### Preddefinovane objekty pre skript

Vo funkciach je potrebne pouzit iste vstavane objekty. Dva najzakladnejsie vstavane objekty su vstupny subor (InputTextFile) a vystupny object stromu (TreeNode). Oba objekty maju specificke metody a premenne. Dalsim vstavanym objektom je ErrorLog, kde sa ukladaju chybove hlasenia, oznamy a vysledny stav konverzie.

Zoznam vstavanych a predinicializovanych objektov je tento

|  |  |
| --- | --- |
| Objekt | Opis |
| Input | Object triedy InputTextFile |
| Log | Object triedy ErrorLog |
| MAXINT | maximalny integer 231 |
| MININT | minimalny integer -(231)+1 |
| true |  |
| false |  |
| null |  |

Ak sa vratime k povodne diskutovanemu procesu char[] -> … -> tree, tak prva operacia char[] -> tree1 je nazyvana "prva konverzia".

Prva konverzia moze prebiehat takto:

(loop

(if (Input feof) break)

(set s (Input checkPatternList pattern\_list))

(Tree addsubnode s)

)

Dalsie spracovania mozu zahrnat hromadne operacie nad uzlami danych typov, prechadzanie vsetkych alebo vybratych poduzlov a podobne.

Sluzia na to funkcie ako makelevels

(Tree makelevels 'block' '{' '}')

(Tree makelevels 'bracket' '(' ')')

### Udajove typy

#### InputTextFile - objekt citania textoveho suboru

**Metody objektu**

|  |  |
| --- | --- |
| Metoda | Opis |
| pushState | ulozi hodnoty premennych absPos, linePos a currLine do zasobnika stavu |
| popState | obnovi hodnoty premennych absPos, linePos a currLine zo zasobnika a vymaze ten zaznam zo zasobnika |
| peekState | obnovi current position zo zasobnika (zasobnik sa nemeni) |
| currentChar | vracia znak na ktory prave ukazuje aktualna pozicia, aktualnu poziciu nemeni |
| canRead | vracia priznak, ci moze citat zo vstupneho suboru (cize ci nie je na konci) |
| reset | nastavi poziciu citania na zaciatok suboru |
| validateEof | skusi urcit novu hodnotu priznaku konca suboru na zaklade aktualnej pozicie v subore a dlzky suboru |
| getChar | reads new character from input file, returns object CharEntry |
| nextPosition | return TextPosition object with position within readed text |
| setNextPosition | sets new position for reading |
| currentLine | returns integer number of line |
| currentLineOffset | returns integer number of current character (in offset from start of line) |
| currentFileOffset | returns integer number of current character (in offset from start of file) |

#### CharEntry

|  |  |
| --- | --- |
| Metoda | Opis |
| character | nacitany znak |
| eof | priznak konca subora |
| position | pozicia v subore, z ktorej bol znak nacitany (object triedy TextPosition) |

#### TextPosition

|  |  |
| --- | --- |
| Metoda | Opis |
| line |  |
| lineOffset |  |
| fileOffset |  |
| setLine |  |
| setLineOffset |  |
| setFileOffset |  |

#### Atom - objekt elementarneho vstupu

Sluzi na zapuzdrenie skupiny znakov zo vstupneho subora (INPUT) do jednej ucelenej pomenovanej jendotky. Z tychto jednotiek sa potom bude skladat strom.

Atomy mozu byt zretazene, na to sluzia metody next, last, setNext

|  |  |
| --- | --- |
| Metoda | Opis |
| endPos | koncova pozicia znakov vo vstupnom subore |
| last | vrati posledny atom v postupnosti zacinajucej aktualnym atomom |
| next | dalsi atom v zozname |
| setNext |  |
| setType |  |
| setValue |  |
| startPos | zaciatocna pozicia znakov vo vstupnom subore |
| type | text, meno atomu |
| value | text, znaky kolektovane zo vstupneho suboru |

#### AtomList - zoznam atomov

|  |  |
| --- | --- |
| Metoda | Opis |
| first |  |
| last |  |
| getAtom |  |
| current |  |
| addItem |  |
| removeAtomsWithType |  |

#### Node - uzol stromu

**Metody objektu**

|  |  |
| --- | --- |
| Metoda | Opis |
| addSubnode | prida objekty v argumentoch ako poduzly stromu |
| addCollection |  |
| name |  |
| value |  |
| setName |  |
| setValue |  |
| children |  |
| canMatchAtom |  |
| takeAllChildren |  |
| takeChildrenBefore |  |
| takeChildrenAfter |  |
| findNodeForward |  |
| findNodeBackward |  |
| getHead |  |
| getTail |  |
| removeHead |  |
| removeTail |  |

#### NodeCollection - zoznam uzlov stromu

|  |  |
| --- | --- |
| Metoda | Opis |
| getNodeList | vrati vsetky poduzly tejtoi kolekcie |
| count | vrati pocet poduzlov v kolekcii |
| first |  |
| last |  |
| addNode |  |
| addCollection |  |
| removeNode |  |
| removeFirstNode |  |
| removeLastNode |  |
| removeNodesOfType |  |
| findNodeBackward |  |
| findNodeForward |  |
| checkIsSubnode |  |
| takeChildrenBefore |  |
| takeChildrenAfter |  |
| createEncapsulationNode |  |

#### ErrorLog - spravca logov a chybovych hlasok

Metody objektu

|  |  |
| --- | --- |
| Metoda | Opis |
| addLog | prida hlasku do log subora |
| finalMessage |  |
| setFinalMessage |  |
| enterDir |  |
| addDir |  |
| goUp |  |
| validateSuccess |  |
| resolveLastError |  |

#### Charset

|  |  |
| --- | --- |
| Metoda | Opis |
| addChar | prida znaky ktore su vymenovane v argumentoch |
| addRange | prida znaky ktore su definovane zaciatocnym a koncovym znakom |
| isInverted |  |
| setInverted |  |
| containsChar |  |

#### ParserAtom

|  |  |
| --- | --- |
| Metoda | Opis |
| testValue |  |
| setTestValue |  |
| testType |  |
| setTestType |  |
| match |  |

#### NamedChar

|  |  |
| --- | --- |
| Metoda | Opis |
| value |  |
| setValue |  |

#### NamedString

|  |  |
| --- | --- |
| Metoda | Opis |
| value |  |
| setValue |  |

#### Pattern

|  |  |
| --- | --- |
| Metoda | Opis |
| matchingMethod |  |
| setMatchingMethod |  |
| name |  |
| setName |  |
| addChar |  |
| addChars | prida do definicie pattern novy zaznam pre znaky  1. arg je minimum  2. arg je maximalny pocet  3. arg a dalsie su znaky |
| addCharset | pride do definicie pattern novy zaznam pre pomenovanu mnozinu znakov  1. arg je minimalny vyskyt  2. arg je maximalny vyskyt  3. arg je meno charsetu |
| addPattern | prida do definicie pattern novy zaznam pre pattern (subpattern)  1. arg je minimum  2. arg je maximum  3. arg je meno patternu |
| addAtom | prida do definicie patternu novy zaznam pre atom  1. arg je minimum  2. arg je maximum  3. arg je testovana hodnota typu  4. arg je testovana hodnota hodnoty  5. arg je meno, ktore sa priradi uzlu v strome, ak tento atom bude najdeny |
| parserTextToAtomList | 1. arg je INPUT  navratova hodnota je ATOMLIST |
| parseAtomListToTree | 1. arg je ATOMLIST  navratova hodnota je TREE |
| parseTextToAtom | 1. arg je INPUT  navratova hodnota je ATOM |

## Vysledny strom

* Strom obsahuje jeden korenovy uzol
* Uzol ma tieto udaje
  + Typ
  + Hodnota
  + Zoznam atributov
  + Zoznam poduzlov

Trieda uzol a jej premenne a metody boli opisane v kapitole [TREE](#_TREE_object_stromu).

## Konvertor

Konvertor vykovana nasledovne cinnosti v uvedenom poradi:

* nacitava konfiguracny subor
* vytvori objekty pre vstavane objekty skriptu
* vykonava hlavnu funkciu v skripte

Vysledok konverzie je v objekte strom, alebo v objekte chyby.

## Dodatocne upravy v strome

Na dodatocne upravy v strome sluzi jazyk na upravy stromu (TREP - TREe Processing?). Syntax a semantika je rovnaka ako pri jazyku LISP (LISt Processing), ale mena funkcii a globalnych objektov su ine.

Pseudo-formalne priklady su napriklad:

(FOR *range* *actions*)

(MOVE *node* *node*)

(INSERT *node* *node* )

# BNF gramatika jazyka TREP je:

*script* :

*s\_expression*

*s\_expression* :

*atomic\_symbol*

'(' *s\_expression . s\_expression* ')'

*list*

*list*:

'(' *s\_expression s\_expressionlist* ')'

*atomic\_symbol:*

*scalar\_number*

*scalar\_string*

*identifier*

*expression\_operator*

*identifier:*

*non\_digit\_char identif\_charlist*

*non\_digit\_char:*

**'A' .. 'Z'**

**'a' .. 'z'**

**'\_'**

*identif\_char:*

**'A' .. 'Z'**

**'a' .. 'z'**

**'0' .. '9'**

**'\_'**

*scalar\_number:*

*digitlist*

*digit:*

**'0' .. '9'**

*scalar\_string:*

*q\_char q\_char*

*dq\_char dq\_char*

*dq\_char non\_a\_charlist dq\_char*

*q\_char non\_s\_charlist q\_char*

*q\_char:*

*'*

*dq\_char:*

*"*

*esc\_char:*

\

*non\_a\_char:*

<all-chars-except-\-and-">

*esc\_char* <all-chars>

*non\_s\_char*:

<all-chars-except-\-and-'>

*esc\_char* <all-chars>

*expression\_operator:*

**= (SET)**

**+ (PLUS)**

**++ (INC)**

**- (MINUS)**

**-- (DEC)**

**\* (MULT)**

**\*\* (POW)**

**/ (DIV)**

**% (MOD)**

**< (LT)**

**<= (LE)**

**> (GT)**

**>= (GE)**

**<> (NE)**

**!= (NE)**

**== (EQ)**

**& (BAND)**

**&& (AND)**

**| (BOR)**

**|| (OR)**

**^ (BXOR)**

**^^ (XOR)**

**! (NOT)**

**~~ (LIKE)**

**!~ (NLIKE)**

# Jazyk TREP

## Vstavane syntakticke konstrukcie

### break

prerusenie vykonavania cyklu

### defun

(defun *name arguments body*)

|  |  |
| --- | --- |
| part | note |
| name | token (name of function) |
| arguments | list of tokens (names of arguments) |
| body | executable list |

Function will be defined in current scope.

(defun meno (argumenty …)

( … vyhodnotenie argumentov … )

)

definicia funkcie

### exec

(exec …list-of-executable-items…)

List of executable items will be executed in sequence and result of last items will be returned as result of this list.

### for

cyklicke vykonavanie akcie na zozname hodnot

(for meno\_premennej

zoznam\_hodnot

akcie

)

prechadza cez zoznam hodnot, priradi hodnotu do premennej meno\_premennej a vykona vsetky akcie

### if

Podmienene vykonavanie prikazov.

(if podmienka

akcia\_ak\_pravda

akcia\_ak\_nepravda

)

### new

(new datatype optional\_arguments ...)

vytvori novy instanciu objektu zadanej triedy

### return

(return hodnota)

Ukoncuje aktualny blok (funkciu) a vracia zadanu hodnotu ako vysledok

### while

(while podmnienka

akcia

)

vykonava zadanu akciu dovtedy, dokym plati podmienka

## Vytvorenie noveho udajoveho typu

### Kroky potrebne k vytvoreniu noveho udajoveho typu

Kroky ktore su tu uvedene su urcene pre implementaciu noveho udajoveho typu spolu s novym delegatom pre script. Moze ist napriklad o customizovanu implementaciu interpretera pre nejaky novy modul alebo novu aplikaciu.

1. vytvorit podtriedu triedy SValue

2. implementovat v novej triede metodu CreateInstance

Prikladom na implementaciu moze byt:

public override SValue CreateInstance(List<SValue> args)

{

if (args.Count > 0)

{

if (args[0] is SVNumber)

return new SVDouble(args[0].getDoubleValue());

double d = 0.0;

double.TryParse(args[0].getStringValue(), out d);

return new SVDouble(d);

}

else

{

return new SVDouble();

}

}

3. implementovat metody getStringValue, getIntValue, getLongValue a getBoolValue

Tento krok je volitelny

4. v customizovanej implementacii metody RegisterDataTypes v novej podtriede ScriptingDelegate implementujeme pridavanie noveho udajoveho typu pocas inicializacie

Priklad:

space.AddDatatype("short", new SVInt16());

space.AddDatatype("Int16", new SVInt16());