

# Soros language specification<sup>1</sup>

*Revised draft*<sup>2</sup>

László Németh (nemeth@numbertext.org)

## 1 Introduction

The Soros programming language is a regular expression (regex) based language for conversion between self-similar character sequences. Interpreters of the Soros language are based on the regex features of the different programming languages, resulting very small implementations and high portability.<sup>3</sup>

A typical conversion task is the number to number name conversion, and the language is originally developed for the generalization of the BAHTTEXT spreadsheet function, a function of Microsoft Excel for number to Thai number name and currency conversion, standardized by ECMA-376 and ISO/IEC 29 500:2008 Office Open XML formats<sup>4</sup>. Soros language is also the intended SPELLOUT replacement of the rule based number formatter (RBNF) of IBM ICU and Unicode CLDR database.<sup>5</sup>

## 2 Syntax

A Soros program implicitly defines a single string function named **\$** (dollar).

### 2.1 Program lines

A Soros program (the function **\$**) consists of lines of conditional commands. Lines can be separated by new line or semicolon characters. Empty lines, leading and trailing white space characters of the lines are removed by the interpreter before the program execution (see also comments in 2.7).

---

1 Released under the Creative Commons 2.0 Attribution-NoDerivs license

2 Revisions:

2009-09-28 Draft.

2010-04-19 New Java implementation and minor fixes in Python implementation.

2010-05-28 Comments about Unicode CLDR and DollarText; corrections for RuleBasedNumberFormat; BSD license for Python and Java interpreters of the appendix of the specification.

2018-04-06 Default NUMBERTEXT mode. Add conditional text and prefix macros, and related changes in left zero deletion and examples to describe ordinal, ordinal-number, year etc. variants of number to number name conversions more easily, for example, to implement Office Open XML's ordinal, cardinalText and ordinalText number format support in LibreOffice and other software.

3 Recent implementations: C++11, Java, JavaScript and Python.

4 This standard also contains an English spellout function with currency handling, see DollarText in ECMA-376, Part 4, section 2.16.4.

5 RBNF is a rule language for numerical input. Handling spelling exceptions, currencies and their special suffixation can be difficult and redundant in RBNF. RBNF database of Unicode CLDR misses currency handling.

## 2.2 Commands

A command has a *regex* part and a return *value* separated by the first (not quoted) white space or white space sequence:

*regex value*

The function `$` returns with the return value of the first matched command. Without any matching the return value is the empty string.

The default NUMBERTEXT mode of Soros programs uses a hidden command for left zero deletion, see in 2.5.

## 2.3 Regex part of the command

The *regex* part is the condition of the command. The *portable* Soros *regex* is an extended regular expression (described in ISO/IEC 9945-2:1993) with a single Perl regex extension, the possible usage of the `\d` notation for `[:digit:]` or `[0-9]` (digits) and the `\D` notation for non-digits (`[^0-9]`).

### 2.3.1 Matching

The *regex* matches the input string, if and only if the matching is full matching and the implicit input boundary values meet the global input boundary criteria (see next section).

### 2.3.2 Global input boundary criteria

The optional regex `^` and `$` boundary notations in REGEX part of the command define global input boundary criteria. The parameter of the first call (the global input) always matches both of these conditions, but the matching of the parameter of a recursive call depends on the position of the recursive call in the return value. The interpreter adds implicit boundary data to the recursive calls. When the recursive call is there in leading position (for example the `$1` in the `$1$2$3` or the nested `$( $1$2$3 )` return values), the input of the recursive call inherits the leading boundary data of the recent input. When the recursive call is there in trailing position (for example the `$3` in the `$1$2$3` or `$( $1$2$3 )` return values), the recursive call inherits the trailing boundary data of the recent input. See also global input boundary modifiers in 2.4.4.

### 2.3.3 Quoting

Regex part can contain white spaces and both parts of the Soros commands can contain leading or trailing white spaces by quoting with the optional ASCII quotation marks:

`"regex with spaces" "value "`

## 2.4 Return value part of the command

The return value is a character sequence with optional spaces, standard regex back references, recursive calls, abbreviated recursive calls and boundary modifiers for recursive calls.

### 2.4.1 Back references – \1 ... \9

\1 ... \9 are back references to the parenthesized subexpressions of the *regex* part, like in the POSIX standard Unix tools, *sed* and *awk*.

### 2.4.2 Recursive calls – \$(param)

The expression \$(*param*) will be replaced by the result of the recursive call of the main \$ function with the parameter *param*. Nested function calls are interpreted in adequate order.

### 2.4.3 Abbreviated recursive calls – \$1 ... \$9

\$1 ... \$9 are abbreviated forms of the \$(\1) ... \$(\9) recursive calls.

### 2.4.4 Global input boundary modifiers

Recursive calls without boundary position in the return value lose their global input boundary values. Pipe signs before or after recursive calls are boundary modifiers and declare global input boundaries within the value.

The call with empty string can differentiate the right and left boundary modifiers. The \$1 | \$2 is equivalent form of the \$1 | | \$2. The following form sets global input boundary modifier only to the second non-empty call: \$1\$() | \$2.

### 2.4.5 Conditional text

Brackets around a function call defines conditional text. If the return value of the function call is not empty, then the text within the brackets is added to the return value:

```
2(\d)                twenty[-$1]
(\d)(\d\d)           $1 hundred[ and $2]
"EUR (\d+)[.](\d\d?)" $1 euro[ $2 cents]
```

are equivalents of the following program lines:

```
20                twenty
2(\d)             twenty-$1
(\d)00            $1 hundred
(\d)(\d\d)        $1 hundred and $2
"EUR (\d+)"        $1 euro
"EUR (\d+)[.](\d\d?)" $1 euro $2 cents
```

## 2.5 NUMBERTEXT mode and \_\_numbertext\_\_

Soros programs have default NUMBERTEXT mode: removing left zeros from the input (also in the recursive calls).

The \_\_numbertext\_\_ directive removes the start of the NUMBERTEXT mode to the place of the directive, so it's possible to modify or overwrite the effect of left zero deletion.

*Example:* left zero deletion can be implemented by adding the following program line to the

programs:

```
0+(0|[1-9]\d*) $1
```

Numbertext reference implementation uses similar prefixes, as in RBNF rules, for ordinal etc. numbers (see also prefix macros in 2.8):

```
"ordinal 1"           first
"ordinal 1(\d{3})"     thousand $(ordinal \1)
```

In NUMBERTEXT mode, left zero deletion works after prefixes, too, for example, by using the following Soros program line at the start of the program (default NUMBERTEXT mode) or at the place of the `__numbertext__` directive:

```
"([a-z][-a-z]* )?0+(0|[1-9]\d*)" $(\1\2)
```

## 2.6 Special characters

Back slash, ASCII quotation mark, dollar sign, left and right parentheses and brackets, pipe sign, hash mark, semicolon and new line characters can be added by their quoted forms and by the `\n` new line notation:

```
\\, \", \$, \(\, \), \[, \], \|, \#, \; \n
```

## 2.7 Comments

Hash mark signs comments (terminated by the next new line character or the end of the program):

```
# full-line comment
1 one # in-line comment
2 two # semicolons (;) are parts of the comments, too
```

### 2.7.1 Language-dependent program lines

Using `[ :lang-code: ]` in a comment defines a language-dependent program line.

For example, German number to number name rules

```
3(\d) [$1und]dreissig # [ :de-CH: ]
3(\d) [$1und]dreißig
```

result “dreissig” for the input “30”, if the language of the conversion is *de-CH* (Swiss Standard German), otherwise “dreißig”.

It's possible to use any number of language codes in the comment of the same program line:

```
(\d)(\d\d) $1 hundred[ and $2] # [ :en-AU: ] [ :en-GB: ] [ :en-IE: ]
(\d)(\d\d) $1 hundred[ $2]
```

## 2.8 Prefix macros

The line `== prefix-literal ==` defines a prefix macro: the arbitrary string „`prefix-literal`” is concatenated with the regex part of the following lines before compilation. Prefix literals and non-empty regex parts separated by a space, handling also boundary notation `^` and

quotation marks as follows:

```
== ordinal ==  
1    first  
^2   second  
"3"  third  
""   Ordinal numbers...
```

equivalent of the following expansion:

```
"ordinal 1"    first  
"^ordinal 2"  second  
"ordinal 3"    third  
"ordinal"      Ordinal numbers...
```

A new macro definition overwrites the previous one.

Empty macro definition switch off macro expansion for the following lines: `== ==.`

### 3 Examples

Reverse input string (Example 1):

```
(.*) (.)      \2$1
```

Add thousand separators (Example 2):

```
(\d+) (\d{3})  $1,\2  
(\d+)         \1
```

Number to Devanagari numeral conversion (Example 3):

```
(\d*)0    $1 ०  
(\d*)1    $1 १  
(\d*)2    $1 २  
(\d*)3    $1 ३  
(\d*)4    $1 ४  
(\d*)5    $1 ५  
(\d*)6    $1 ६  
(\d*)7    $1 ७  
(\d*)8    $1 ८  
(\d*)9    $1 ९
```

Number to English number name conversion program (Example 4):

```
__numbertext__  
^0 zero; 1 one; 2 two; 3 three; 4 four; 5 five; 6 six;  
7 seven; 8 eight; 9 nine; 10 ten; 11 eleven; 12 twelve;  
13 thirteen; 15 fifteen; 18 eighteen; 1(\d) $1teen  
2(\d) twenty[-$1]; 3(\d) thirty[-$1]; 4(\d) forty[-$1]  
5(\d) fifty[-$1]; 8(\d) eighty[-$1]  
(\d)(\d) $1ty[-$2]  
(\d)(\d\d) $1 hundred[ and $2]
```

```

# separator function
:0+                # one million
:0*\d?\d " and "   # one million and twenty-two
:\d+ ", "          # one million, one thousand

(\d{1,2})([1-9]\d\d) $1 thousand[ $2] # ten thousand two hundred
(\d{1,3})(\d{3}) $1 thousand$(:\2)$2 # one hundred thousand, two
hundred
(\d{1,3})(\d{6}) $1 million$(:\2)$2
(\d{1,3})(\d{9}) $1 billion$(:\2)$2
(\d{1,3})(\d{12}) $1 trillion$(:\2)$2
(\d{1,3})(\d{15}) $1 quadrillion$(:\2)$2
(\d{1,3})(\d{18}) $1 quintillion$(:\2)$2
(\d{1,3})(\d{21}) $1 sextillion$(:\2)$2
(\d{1,3})(\d{24}) $1 septillion$(:\2)$2

# negative number
[--](\d+) negative |$1

# decimals
([--]?\d+)[.,] $1| point
([--]?\d+[.,]\d*)(\d) $1| |$2

# currency unit/subunit singular/plural
us:([^\,]*) ,([^\,]*) ,([^\,]*) ,([^\,]*) \1
up:([^\,]*) ,([^\,]*) ,([^\,]*) ,([^\,]*) \2
ss:([^\,]*) ,([^\,]*) ,([^\,]*) ,([^\,]*) \3
sp:([^\,]*) ,([^\,]*) ,([^\,]*) ,([^\,]*) \4

AUD:(\D+) $(\1: Australian dollar, Australian dollars, cent,
cents)
CAD:(\D+) $(\1: Canadian dollar, Canadian dollars, cent, cents)
CHF:(\D+) $(\1: Swiss franc, Swiss francs, centime, centimes)
CNY:(\D+) $(\1: Chinese yuan, Chinese yuan, fen, fen)
EUR:(\D+) $(\1: euro, euro, cent, cents)
GBP:(\D+) $(\1: pound sterling, pounds sterling, penny, pence)
HKD:(\D+) $(\1: Hong Kong dollar, Hong Kong dollars, cent, cents)
INR:(\D+) $(\1: Indian rupee, Indian rupees, paisa, paise)
JPY:(\D+) $(\1: Japanese yen, Japanese yen, sen, sen)
MXN:(\D+) $(\1: Mexican peso, Mexican pesos, centavo, centavos)
NZD:(\D+) $(\1: New Zealand dollar, New Zealand dollars, cent,
cents)
SGD:(\D+) $(\1: Singapore dollar, Singapore dollars, cent, cents)
USD:(\D+) $(\1: U.S. dollar, U.S. dollars, cent, cents)
ZAR:(\D+) $(\1: South African rand, South African rand, cent,
cents)

```

```

"(JPY [--]?\\d+)[.,](\\d\\d)0" $1
"(JPY [--]?\\d+[.,]\\d\\d)(\\d)" $1 $2 rin

"([A-Z]{3}) ([--]?1)" $2 $(\\1:us)
"([A-Z]{3}) ([--]?\\d+)" $2 $(\\1:up)

"(CNY [--]?\\d+)[.,](\\d)0?" $1 $2 jiao
"(CNY [--]?\\d+[.,]\\d)(\\d)" $1 $2 fen

"(([A-Z]{3}) [--]?\\d+)[.,](01)" $1 and |$(1) $(\\2:ss)
"(([A-Z]{3}) [--]?\\d+)[.,](\\d)" $1 and |$(\\30) $(\\2:sp)
"(([A-Z]{3}) [--]?\\d+)[.,](\\d\\d)" $1 and |$3 $(\\2:sp)

== ordinal ==

# convert to text, and recall to convert
# cardinal names to ordinal ones

([--]?\\d+) $(ordinal |$1)

(\\.*)one    \\1first
(\\.*)two    \\1second
(\\.*)three  \\1third
(\\.*)five   \\1fifth
(\\.*)eight  \\1eighth
(\\.*)nine   \\1ninth
(\\.*)twelve  \\1twelfth
(\\.*)y       \\1ieth
"(\\.*[^ ])*" \\1th

== ordinal-number ==

(\\.*1\\d)    \\1th
(\\.*1)      \\1st
(\\.*2)      \\1nd
(\\.*3)      \\1rd
(\\.*)       \\1th

```

## 4 Appendix

### 4.1 Python implementation of the Soros interpreter<sup>6</sup>

```

"Soros interpreter (see http://numbertext.org)"
from __future__ import unicode_literals

```

---

<sup>6</sup> This program is released under the free BSD license.

```

from __future__ import print_function
import re, sys

def run(program, data, lang):
    return compile(program, lang).run(data)

def compile(program, lang):
    return _Soros(program, lang)

# conversion function
def _tr(text, chars, chars2, delim):
    for i in range(0, len(chars)):
        text = text.replace(delim + chars[i], chars2[i])
    return text

# string literals for metacharacter encoding
_m = "\\\";#$( )|[]"
_c = u"\uE000\uE001\uE002\uE003\uE004\uE005\uE006\uE007\uE008\uE009" # Unicode
private area
_pipe = u"\uE003"
# separator prefix = \uE00A

# pattern to recognize function calls in the replacement string
_func = re.compile(_tr(r""""(?:\|?(?:\$\\( )+)? # optional nested calls
    (\|?\$\\(((^\\(\\))*\\)|?) # inner call (2 subgroups)
    (?:\\)+\\|?)?""", # optional nested calls
    _m[4:8], _c[:4], "\\\"), re.X) # \$, \(\, \), \| -> \uE000..\uE003

class _Soros:
    def __init__(self, prg, lang):
        self.lines = []
        if prg.find("__numbertext__") == -1:
            prg = "__numbertext__;" + prg
        # default left zero deletion
        # and separator function (no separation, if subcall returns with empty
string)
        prg = prg.replace("__numbertext__", u""""0+(0|[1-9]\\d*) $1
\"([a-z][-a-z]* )0+(0|[1-9]\\d*)\" $\\1\\2)
\"\\uE00A(.*)\\uE00A(.+)\\uE00A(.*)\" \\1\\2\\3
\"\\uE00A.*\\uE00A\\uE00A.*\"
""")
        prg = _tr(prg, _m[:4], _c[:4], "\\\"") # \\, \", \;, \# -> \uE000..\uE003
        # switch off all country-dependent lines, and switch on the requested ones
        prg = re.sub(r"^(^[\n;])(^[\n;#]*#^[^\\n]*[[ ]:[^\\n:]]*:[^\\n]*)", r"\1#\2",
prg)
        prg = re.sub(r"^(^[\n;])(^[\n;#]*#^[^\\n]*[[ ]:" + lang.replace("_", "-") +
r":)[^\\n]*)", r"\1\2", prg)
        matchline = re.compile("^s*(\"[^\"]*\"|^[^s]*)s*(.*[^\s])?s*$")
        prefix = ""
        for s in re.sub("#([^\n]*)?(\\n|$)", ";", prg).split(";"):
            macro = re.match("== *(.*[^\n ]?) ==", s)
            if macro != None:
                prefix = macro.group(1)
                continue
            m = matchline.match(s)

```





```

while n:
    b = False
    e = False
    if n.group(1)[0:1] == _pipe or n.group()[0:1] == _pipe:
        b = True
    elif n.start() == 0:
        b = begin
    if n.group(1)[-1:] == _pipe or n.group()[-1:] == _pipe:
        e = True
    elif n.end() == len(s):
        e = end
    s = s[:n.start(1)] + self._run(n.group(2), b, e) + s[n.end(1):]
    n = _func.search(s)
return s
return ""

```

## 4.2 Java implementation of the Soros interpreter<sup>7</sup>

```

package org.numbertext;

import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.ArrayList;

public class Soros {
    private ArrayList<Pattern> patterns = new ArrayList<Pattern>();
    private ArrayList<String> values = new ArrayList<String>();
    private ArrayList<Boolean> begins = new ArrayList<Boolean>();
    private ArrayList<Boolean> ends = new ArrayList<Boolean>();

    private static String m = "\\\";#";
    private static String m2 = "$()|[]";
    private static String c = "\uE000\uE001\uE002\uE003";
    private static String c2 = "\uE004\uE005\uE006\uE007\uE008\uE009";
    private static String slash = "\uE000";
    private static String pipe = "\uE003";

    // pattern to recognize function calls in the replacement string

    private static Pattern func = Pattern.compile(translate(
        "(?:\\|?(?:\\$\\(\\)+)?" +           // optional nested calls
        "(\\|?\\$\\(((\\^\\(\\|\\|)*)\\|\\|\\|?)" + // inner call (2 subgroups)
        "(?:\\|)+\\|?)?",                 // optional nested calls
        m2.substring(0, c.length()), c, "\\"); // \$, \(), \), \| ->
    \uE000..\uE003

    private boolean numbertext = false;

    public Soros(String source, String lang) {
        source = translate(source, m, c, "\\"); // \, \", \;, \# ->
    \uE000..\uE003
        // switch off all country-dependent lines, and switch on the requested ones
        source = source.replaceAll("(^|[\n;])([^\n;#]*#[^\n]*\\[:[^\n:\\]]*:]

```

<sup>7</sup> This program is released under the free BSD license.

```

[^\n]*)", "$1#$2")
    .replaceAll("(^|[\n;])#([^\n;]*#[^\n]*\\[:\" + lang.replace('_', '-') +
":][^\n]*)", "$1$2")
    .replaceAll("(#[^\n]*)?(\\n|$)", ";"); // remove comments
if (source.indexOf("__numbertext__") == -1)
    source = "__numbertext__;" + source;
source = source.replace("__numbertext__",
    // default left zero deletion
    "\"([a-z][-a-z]* )?0+(0|[1-9]\\d*)\" \"$\\1\\2\";" +
    // separator function
    "\"\\uE00A(.)\\uE00A(.)\\uE00A(.)\" \"\\1\\2\\3\";" +
    // no separation, if subcall returns with empty string
    "\"\\uE00A.*\\uE00A\\uE00A.*\"");

Pattern p = Pattern.compile("^\\s*(\"[^\"]*\")\\s*"
[^\s]*)\\s*(.*[^\s])?\\s*$");
Pattern macro = Pattern.compile("== *(.*[ ]?) ==");
String prefix = "";
for (String s : source.split(";")) {
    Matcher matchmacro = macro.matcher(s);
    if (matchmacro.matches()) {
        prefix = matchmacro.group(1);
        continue;
    }
    Matcher sp = p.matcher(s);
    if (!prefix.equals("") && !s.equals("") && sp.matches()) {
        s = sp.group(1).replaceFirst("^\"", "").replaceFirst("\"$", "");
        s = "\"" + (s.startsWith("^") ? "^" : "") + prefix + (s.equals("") ?
"" : " ") +
            s.replaceFirst("^\\^", "") + "\" " + sp.group(2);
        sp = p.matcher(s);
    }
    if (!s.equals("") && sp.matches()) {
        s = translate(sp.group(1).replaceFirst("^\"",
"").replaceFirst("\"$", ""),
            c.substring(1), m.substring(1), "");
        s = s.replace(slash, "\\"); // -> \, ", ;, #
        String s2 = "";
        if (sp.group(2) != null) s2 = sp.group(2).replaceFirst("^\"",
"").replaceFirst("\"$", "");
        s2 = translate(s2, m2, c2, "\\"); // \$, \(\, \), \|, \[, \] ->
\uE004..\uE009
        // call inner separator: [ ... $1 ... ] -> $(\uE00A ...
\uE00A$1\uE00A ... )
        s2 = s2.replaceAll("^\\[[\$](\\d\\d?|\\([^\"]\\)+\\))\"", "\\$
(\\uE00A\\uE00A|\\$\\$1\\uE00A)" // add "|"
            .replaceAll("\\\\([^$\\\\\\\\\\*)\\$](\\d\\d?|\\([^\"]\\)+\\))\"", "\\$
(\\uE00A$1\\uE00A\\$\\$2\\uE00A)"
            .replaceAll("\\uE00A\\|\\$", "\\uE00A)" // add "|" in terminating
position
            .replaceAll("\\\\]", ")")
            .replaceAll("(\\$\\d|\\))\\\\\\\\\\$", "$1|\\\\\\$"); // $()|$( ) -> $
()| |$( )
        s2 = translate(s2, c, m, ""); // \uE000..\uE003-> \, ", ;, #
        s2 = translate(s2, m2.substring(0, c.length()), c, ""); // $,

```

```

(, ), | -> \uE000..\uE003
    s2 = translate(s2, c2, m2, "");    // \uE004..\uE009 -> $, (, ), |,
[, ]
    s2 = s2.replaceAll("[\$]", "\\\$")    // $ -> \$
    .replaceAll("\uE000(\\d)", "\uE000\uE001\\$$1\uE002") // $n -> $
(\n)
    .replaceAll("\\\\(\\d)", "\\$1") // \[n] -> $[n]
    .replace("\\n", "\n");    // \n -> [new line]
patterns.add(Pattern.compile("^" + s.replaceFirst("^\\^", ""))
    .replaceFirst("\\$", "")) + "$");
begins.add(s.startsWith("^"));
ends.add(s.endsWith("$"));
values.add(s2);
    }
}
}

public String run(String input) {
    return run(input, true, true);
}

private String run(String input, boolean begin, boolean end) {
    for (int i = 0; i < patterns.size(); i++) {
        if ((!begin && begins.get(i)) || (!end && ends.get(i))) continue;
        Matcher m = patterns.get(i).matcher(input);
        if (!m.matches()) continue;
        String s = m.replaceAll(values.get(i));
        Matcher n = func.matcher(s);
        while (n.find()) {
            boolean b = false;
            boolean e = false;
            if (n.group(1).startsWith(pipe) || n.group().startsWith(pipe)) b =
true;
            else if (n.start() == 0) b = begin;
            if (n.group(1).endsWith(pipe) || n.group().endsWith(pipe)) e = true;
            else if (n.end() == s.length()) e = end;
            s = s.substring(0, n.start(1)) + run(n.group(2), b, e) +
s.substring(n.end(1));
            n = func.matcher(s);
        }
        return s;
    }
    return "";
}

private static String translate(String s, String chars, String chars2, String
delim) {
    for (int i = 0; i < chars.length(); i++) {
        s = s.replace(delim + chars.charAt(i), "" + chars2.charAt(i));
    }
    return s;
}
}

```