Soros language specification[1]

DRAFT (2009-08-21)

László Németh (nemeth@openoffice.org)

1 Introduction

The Soros programming language is a regex based language for conversion between self-similar character sequences. 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 the ECMA–376 and ISO/IEC 29 500:2008 Office Open XML format. Soros language is also the intended SPELLOUT replacement of the rule based number formatter (RBNF) of IBM ICU.[2]

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).

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.

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 RBNF is a complex but insufficient rule language only for numerical input. For example, the Swedish cardinal number rules of ICU are explicit bad, because the author didn't want to add 200 new rules to describe the special orthography of the numbers 1000, 11 000, 21 000 etc. Handling currencies and their special suffixation is not possible in RBNF for several languages.

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.

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`.

2.5 __numbertext__

The __numbertext__ directive sets the NUMBERTEXT mode: removing left zeros from the input (also in the recursive calls) and removing leading, trailing and double spaces from the output.

Note: The left zero deletion can be implemented by adding the following (first) program line to the programs:

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

2.6 Special characters

Back slash, ASCII quotion mark, dollar sign, left and right parentheses, 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
```

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    $1०
(\d*)1    $1१
(\d*)2    $1२
(\d*)3    $1३
(\d*)4    $1४
(\d*)5    $1५
(\d*)6    $1६
(\d*)7    $1७
(\d*)8    $1८
(\d*)9    $1९
```

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
20 twenty; 2(\d) twenty-$1; 30 thirty; 3(\d) thirty-$1; 40 forty; 4(\d) forty-$1
50 fifty; 5(\d) fifty-$1; 80 eighty; 8(\d) eighty-$1; (\d)0 $1ty
(\d)(\d) $1ty-$2
(\d)(00) $1 hundred

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

(\d)(\d\d) $1 hundred$(:\2) $2
(\d{1,2})([^0]\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)
```

4 Appendix

Python implementation of the Soros interpreter:

```
"Soros interpreter (see http://numbertext.org)"
import re

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

def compile(program):
    return _Soros(program)

# 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" # Unicode private area
_pipe = u"\uE003"

# 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:], _c[:4], "\\"), re.X)  # \$, \(, \), \| -> \uE000..\uE003

class _Soros:
    def __init__(self, prg):
       self.lines = []
       self.numbertext = False
       if prg.find("__numbertext__") > -1:
           self.numbertext = True
           prg = "0+(0|[^0]\\d*) $1\n" + prg.replace("__numbertext__", "")
       prg = _tr(prg, _m[:4], _c[:4], "\\") # \\, \", \;, \# -> \uE000..\uE003
       for s in re.sub("(#[^\n]*)?(\n|$)", ";", prg).split(";"):
           m = re.match("^\s*(\"[^\"]*\"|[^\s]*)\s*(.*[^\s])?\s*$", s)
           if m != None:
              s = _tr(m.group(1).strip("\""), _c[1:4], _m[1:4], "") \
                  .replace(_c[_m.find("\\")], "\\\\") # -> \\, ", ;, #
              if m.group(2) != None:
                  s2 = m.group(2).strip("\"")
              else:
                  s2 = ""
              s2 = _tr(s2, _m[4:], _c[4:], "\\") # \$, \(, \), \| -> \uE004..\uE007
              s2 = s2.replace(")|$", ")||$")     # $()|$() -> $()||$()
              s2 = _tr(s2, _c[:4], _m[:4], "")   # \uE000..\uE003-> \, ", ;, #
              s2 = _tr(s2, _m[4:], _c[:4], "")   # $, (, ), | -> \uE000..\uE003
              s2 = _tr(s2, _c[4:], _m[4:], "\\") # \uE004..\uE007 -> $, (, ), |
              s2 = re.sub(ur"\\(\d)", r"\\g<\1>",
                  re.sub(ur"\uE000(\d)", ur"\uE000\uE001\\g<\1>\uE002", s2))
              self.lines = self.lines + [[
                  re.compile("^" + s.lstrip("^").rstrip("$") + "$"),
                  s2, s[:1] == "^", s[-1:] == "$"]]

    def run(self, data):
       if self.numbertext:
           return re.sub("  +", " ", self._run(data, True, True).strip())
       return self._run(data, True, True)

    def _run(self, data, begin, end):
       for i in self.lines:
           if not ((begin == False and i[2]) or (end == False and i[3])):
              m = i[0].match(data)
              if m:
                  s = m.expand(i[1])
                  n = _func.search(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 ""
```