```
(ns symbolic-differentiation.core
  (:gen-class)
)

"Odwołania do matematycznych funkcji z biblioteki standardowej Clojure,
 liczba parametrów zależy od funkcji
 Przykła użycia: (log 2 4)"
(defn sin [x] (Math/sin x))
(defn cos [x] (Math/cos x))
(defn tg [x] (Math/tan x))
(defn ctg [x] (/ 1 (Math/tan x)))
(defn sec [x] (/ 1 (Math/cos x)))
(defn csc [x] (/ 1 (Math/sin x)))
(defn exp [x] (Math/exp x))
(defn ln [x] (Math/log x))
(defn log [x, y] (/ (Math/log y) (Math/log x)))
(defn sqrt [x, y] (Math/pow x (/ 1 y)))
(defn arcsin [x] (Math/asin x))
(defn arccos [x] (Math/acos x))
(defn arctg [x] (Math/atan x))
(defn arcctg [x] (Math/atan (/ 1 x)))
(defn arcsec [x] (Math/acos (/ 1 x)))
(defn arccsc [x] (Math/asin (/ 1 x)))
(defn sinh [x] (Math/sinh x))
(defn cosh [x] (Math/cosh x))
(defn tgh [x] (Math/tanh x))
(defn ctgh [x] (/ (cosh x) (sinh x)))
(defn sech [x] (/ 1 (cosh x)))
(defn csch [x] (/ 1 (sinh x)))
(defn pow [x, y] (Math/pow x y))

"Funkcje sprawdzające czy odpowiadające wyrażenie jest prawidłowo sformułowane
 Przykład użycia: (pow 2 3) => true
                  (sin) => false - funkcja sin przyjmuje 1 argument"
(defn sin? [x] (and (= (count x) 2) (= (first x) 'sin)))
(defn cos? [x] (and (= (count x) 2) (= (first x) 'cos)))
(defn tg? [x] (and (= (count x) 2) (= (first x) 'tg)))
(defn ctg? [x] (and (= (count x) 2) (= (first x) 'ctg)))
(defn sec? [x] (and (= (count x) 2) (= (first x) 'sec)))
(defn csc? [x] (and (= (count x) 2) (= (first x) 'csc)))
(defn exp? [x] (and (= (count x) 2) (= (first x) 'exp)))
(defn ln? [x] (and (= (count x) 2) (= (first x) 'ln)))
(defn log? [x] (and (= (count x) 3) (= (first x) 'log)))
(defn sqrt? [x] (and (= (count x) 3) (= (first x) 'sqrt)))
(defn arcsin? [x] (and (= (count x) 2) (= (first x) 'arcsin)))
(defn arccos? [x] (and (= (count x) 2) (= (first x) 'arccos)))
(defn arctg? [x] (and (= (count x) 2) (= (first x) 'arctg)))
(defn arcctg? [x] (and (= (count x) 2) (= (first x) 'arcctg)))
(defn arcsec? [x] (and (= (count x) 2) (= (first x) 'arcsec)))
(defn arccsc? [x] (and (= (count x) 2) (= (first x) 'arccsc)))
(defn sinh? [x] (and (= (count x) 2) (= (first x) 'sinh)))
(defn cosh? [x] (and (= (count x) 2) (= (first x) 'cosh)))
(defn tgh? [x] (and (= (count x) 2) (= (first x) 'tgh)))
(defn ctgh? [x] (and (= (count x) 2) (= (first x) 'ctgh)))
(defn sech? [x] (and (= (count x) 2) (= (first x) 'sech)))
(defn csch? [x] (and (= (count x) 2) (= (first x) 'csch)))
(defn pow? [x] (and (= (count x) 3) (= (first x) 'pow)))
(defn addition? [x] (and (=(count x) 3) (= (first x) '+)))
(defn subtraction? [x] (and (=(count x) 3) (= (first x) '-)))
(defn multiplication? [x] (and (=(count x) 3) (= (first x) '*)))
(defn division? [x] (and (=(count x) 3) (= (first x) '/)))

"Główna funkcja wyznaczająca pochodną podanego wyrażenia po określonej zmiennej
 Przykład użycia: (differentiation '(ln (tg x)) 'x)
                                => (* (* 1 (/ 1 (pow (cos x) 2))) (/ 1 (tg x)))
 Argumenty: expression - wyrażenie, z którego liczymy pochodną,
            variable - nazwa zmiennej, po której liczymy pochodną"
(defn differentiation
  [expression variable]
  (cond
    (number? expression)
```

```
72        0
73      (symbol? expression)
74        (if (= expression variable)
75          1
76          0)
77      (sin? expression)
78        (list '*
79          (differentiation (second expression) variable)
80          (list 'cos (second expression)))
81      (cos? expression)
82        (list '*
83          (differentiation (second expression) variable)
84          (list '- 0 (list 'sin (second expression))))
85      (tg? expression)
86        (list '*
87          (differentiation (second expression) variable)
88          (list '/
89            1
90            (list 'pow
91              (list 'cos (second expression))
92              2)))
93      (ctg? expression)
94        (list '*
95          (differentiation (second expression) variable)
96          (list '/
97            -1
98            (list 'pow
99              (list 'sin (second expression))
100             2)))
101     (sec? expression)
102       (list '*
103         (differentiation (second expression) variable)
104         (list '*
105           (list 'tg (second expression))
106           (list 'sec (second expression))))
107     (csc? expression)
108       (list '*
109         (differentiation (second expression) variable)
110         (list '*
111           (list '- 0 (list 'ctg (second expression)))
112           (list 'csc (second expression))))
113     (exp? expression)
114       (list '*
115         (differentiation (second expression) variable)
116         (list 'exp (second expression)))
117     (ln? expression)
118       (list '*
119         (differentiation (second expression) variable)
120         (list '/ 1 (second expression))
121       )
122     (log? expression)
123       (list '*
124         (differentiation (second (next expression)) variable)
125         (list '/
126           1
127           (list '*
128             (second (next expression))
129             (list 'ln (second expression)))))
130     (sqrt? expression)
131       (list '*
132         (differentiation (second (next expression)) variable)
133         (list '*
134           (list '/ 1 (second expression))
135           (list 'pow (second (next expression))
136             (list '-
137               (list '/ 1 (second expression))
138               1))))
139     (pow? expression)
140       (list '*
141         (differentiation (second expression) variable)
142         (list '*
```

```
143           (second (next expression))
144           (list 'pow  (second expression)
145             (list '-
146               (second (next expression))
147               1))))
148     (arcsin? expression)
149       (list '*
150         (differentiation (second expression) variable)
151         (list '/
152           1
153           (list 'pow
154             (list '- 1 (list 'pow (second expression) 2))
155             0.5
156           )))
157     (arccos? expression)
158       (list '*
159         (differentiation (second expression) variable)
160         (list '/
161           -1
162           (list 'pow
163             (list '- 1 (list 'pow (second expression) 2))
164             0.5
165           )))
166     (arctg? expression)
167       (list '*
168         (differentiation (second expression) variable)
169         (list '/
170           1
171           (list '+ (list 'pow (second expression) 2) 1)
172         ))
173     (arcctg? expression)
174       (list '*
175         (differentiation (second expression) variable)
176         (list '/
177           -1
178           (list '+ (list 'pow (second expression) 2) 1)
179         ))
180     (arcsec? expression)
181       (list '*
182         (differentiation (second expression) variable)
183         (list '/
184           1
185           (list '*
186             (list 'pow
187               (list '-
188                 1
189                 (list '/
190                   1
191                   (list 'pow (second expression) 2)))
192               0.5)
193             (list 'pow (second expression) 2))
194         ))
195     (arccsc? expression)
196       (list '*
197         (differentiation (second expression) variable)
198         (list '/
199           -1
200           (list '*
201             (list 'pow
202               (list '-
203                 1
204                 (list '/
205                   1
206                   (list 'pow (second expression) 2)))
207               0.5)
208             (list 'pow (second expression) 2))
209         ))
210     (sinh? expression)
211       (list '*
212         (differentiation (second expression) variable)
213         (list 'cosh (second expression)))
```

```
214     (cosh? expression)
215       (list '*
216         (differentiation (second expression) variable)
217         (list 'sinh (second expression)))
218     (tgh? expression)
219       (list '*
220         (differentiation (second expression) variable)
221         (list 'pow (list 'sech (second expression)) 2))
222     (ctgh? expression)
223       (list '*
224         (differentiation (second expression) variable)
225         (list '- 0 (list 'pow (list 'csch (second expression)) 2)))
226     (sech? expression)
227       (list '*
228         (differentiation (second expression) variable)
229         (list '* (list 'tgh (second expression)) (list '- 0 (list 'sech (second expression)))))
230     (csch? expression)
231       (list '*
232         (differentiation (second expression) variable)
233         (list '- 0 (list '* (list 'ctgh (second expression)) (list 'csch (second expression)))))
234     (addition? expression)
235       (list '+
236         (differentiation (second expression) variable)
237         (differentiation (second (rest expression)) variable))
238     (subtraction? expression)
239       (list '-
240         (differentiation (second expression) variable)
241         (differentiation (second (rest expression)) variable))
242     (multiplication? expression)
243       (list '+
244         (list '*
245           (differentiation (second expression) variable)
246           (second (rest expression)))
247         (list '*
248           (second expression)
249           (differentiation (second (rest expression)) variable)))
250     (division? expression)
251       (list '/
252           (list '-
253             (list '*
254               (differentiation (second expression) variable)
255               (second (rest expression)))
256             (list '*
257               (second expression)
258               (differentiation (second (rest expression)) variable)))
259           (list 'pow
260             (second (rest expression))
261             2))
262   )
263 )
264
265 "Funkcja liczy wartość wyrażenia dla zadanej wartości zmiennej.
266  Oryginalne wartości zmiennych pozostają bez zmian
267  Przykład użycia: (diff-eval expression variable (first values))
268  Argumenty: expression - wyrażenie, z którego liczymy pochodną,
269             variable - nazwa zmiennej, po której liczymy pochodną
270             argument-values - zadane wartości zmiennej"
271 (defn diff-eval
272   [diff variable argument-value]
273   (def diff-string diff)
274   (let
275     [pom (resolve variable)]
276     (do
277       (if (not= pom nil)
278         (let [val (eval variable)]
279           (do
280             (eval(read-string (clojure.string/join " " ["(" "def" variable argument-value ")"])))
281             (let [result (eval(read-string (pr-str diff-string)))]
282               (do
283                 (eval(read-string (clojure.string/join " " ["(" "def" variable val ")"])))
284                 result
```

```
285               )
286             )
287           ))
288         (do ;else
289           (eval(read-string (clojure.string/join " " ["(" "def" variable argument-value ")"])))
290           (eval(read-string (pr-str diff-string)))
291         )
292       )
293     )
294   )
295 )
296
297 "Funkcja liczy pochodną podanej w parametrze funkcji i oblicza wartości pochodnej w zadanych punktach
298  Przykład użycia: (function-multiple-differentiation-values '(ln (tg x)) 'x '(1 2 3 4))
299  Argumenty: expression - wyrażenie, z którego liczymy pochodną,
300             variable - nazwa zmiennej, po której liczymy pochodną
301             argument-values - zadane wartości zmiennej"
302 (defn function-multiple-differentiation-values
303   [expression variable argument-values]
304     (loop [expression (differentiation expression variable) variable variable argument-values argument-values
305 counted []]
306       (do
307         (if (= (count argument-values) 0)
308           counted
309           (recur expression variable (next argument-values) (conj counted (diff-eval expression variable (first
310 argument-values))))
311         )
312       )
313     )
314 )
315
316 "Funkcja oblicza wartości funkcji w zadanych punktach
317  Przykład użycia: (function-multiple-values '(* y (ln x)) 'x '(1 2 3 4))
318  Argumenty: expression - wyrażenie, z którego liczymy pochodną,
319             variable - nazwa zmiennej, po której liczymy pochodną
320             argument-values - zadane wartości zmiennej"
321 (defn function-multiple-values
322   [expression variable argument-values]
323     (loop [expression expression variable variable argument-values argument-values counted []]
324       (do
325         (if (= (count argument-values) 0)
326           counted
327           (recur expression variable (next argument-values) (conj counted (diff-eval expression variable (first
328 argument-values))))
329         )
330       )
331     )
332 )
333
334 "Funkcja usuwająca elementy neutralne dla danego wyrażenia np. (+ 0 x) lub (* 1 x).
335  Służy do wyznaczenia uproszczonej postaci pochodnej funkcji.
336  Wykorzystywana wewnętrznie przez funkcję optimize-expression"
337 (defn optimize-expression_priv
338   [expression]
339   (if (list? expression)
340     (let [expr expression]
341       (if (list? expr)
342         (do
343           (cond
344             (and (= (first expr) '*) (= (second expr) 1))
345               (do
346                 (def __optimized__ (conj __optimized__ "("))
347                 (optimize-expression_priv (second (next expr)))
348                 (def __optimized__ (conj __optimized__ ")"))
349               )
350             (and (= (first expr) '*) (= (second expr) 0))
351               (optimize-expression_priv '(0))
352             (and (= (first expr) '*) (= (second (next expr)) 0))
353               (optimize-expression_priv '(0))
354             (and (= (first expr) '/) (= (second expr) 0))
355               (optimize-expression_priv '(0))
```

```
356             (and (= (first expr) '+) (= (second expr) 0))
357               (do
358                 (def __optimized__ (conj __optimized__ "("))
359                 (optimize-expression_priv (second (next expr)))
360                 (def __optimized__ (conj __optimized__ ")"))
361               )
362             (and (= (first expr) '-) (= (second (next expr)) 0))
363               (do
364                 (def __optimized__ (conj __optimized__ "("))
365                 (optimize-expression_priv (second expr))
366                 (def __optimized__ (conj __optimized__ ")"))
367               )
368             (and (= (first expr) '/) (= (second (next expr)) 1))
369               (do
370                 (def __optimized__ (conj __optimized__ "("))
371                 (optimize-expression_priv (second expr))
372                 (def __optimized__ (conj __optimized__ ")"))
373               )
374             (= 1 1)
375               (do
376               (def __optimized__ (conj __optimized__ "("))
377               (doseq [x expr]  (optimize-expression_priv x))
378               (def __optimized__ (conj __optimized__ ")"))
379               )
380           )
381         )
382         (do ;else
383           (if (not= nil expr)
384             (def __optimized__ (conj __optimized__ expr))
385           )
386         )
387       )
388     )
389     (if (not= nil expression) ;else
390       (def __optimized__ (conj __optimized__ expression))
391     )
392   )
393
394   __optimized__
395 )
396
397 "Funkcja usuwająca z listy element o indeksie podanym w parametrze funkcji.
398  Część funkcji służącej do uproszczenia otrzymanych pochodnych, do użytku wewnętrzengo"
399 (defn remove_index
400   [vec index]
401   (let [coll vec
402     i index]
403     (into [] (concat (subvec coll 0 i)
404         (subvec coll (inc i)))))
405 )
406
407 "Funkcja usuwająca zbędne nawiasy z wyrażenia.
408  Funkcja do użytku wewnętrznego"
409 (defn remove_doubled_brackets_priv
410   [str]
411   (loop [str str open 0 delete false index 0]
412     (if (>= index (count str))
413       str
414       (do
415         (if (= false delete)
416           (if (and (= (nth str index) "(") (= (nth str (+ index 1)) "(")) ;))
417             (recur (remove_index str index) 1 true (+ index 1))
418             (recur str 0 false (+ index 1))
419           )
420           (if (= (nth str index) "(") ;)
421             (recur str (+ open 1) true (+ index 1))
422             (if (= (nth str index) ")") ;(
423               (if (= open 1)
424                 (recur (remove_index str index) 0 false (+ index 1))
425                 (recur str (- open 1) true (+ index 1))
426               )
```

```
              (recur str open true (+ index 1))
            )
          )
        )
      )
    )
  )
)

"Funkcja rekurencyjnie usuwająca niepotrzebne nawiasy z wyrażenia.
 Funkcja do użytku wewnętrznego"
(defn remove_doubled_brackets
  [str]
  (loop [orig str res (remove_doubled_brackets_priv orig)]
    (if (= orig res)
      res
      (recur res (remove_doubled_brackets_priv res))
    )
  )
)

"Główna funkcja służąca do uproszczenia wzoru otrzymanej pochodnej.
 Wykorzystuje powyższe funkcje: optimize-expression_priv, remove_doubled_brackets
 Przykład użycia: (optimize-expression (differentiation '(ln (tg x)) 'x))
 Argumenty: expression - wyrażenie, z którego liczymy pochodną"
(defn optimize-expression
  [expression]
  (def __optimized__ [])
  (loop [expression expression result (optimize-expression_priv expression)]
    (do
      (def __optimized__ [])
      (let
        [res (read-string (clojure.string/replace (clojure.string/join " " (remove_doubled_brackets result))
(re-pattern "\\(\\s0\\s\\)") "0"))]
        (do
            (if (= res expression)
              res
              (recur res (optimize-expression_priv res))
            )
        )
      )
    )
  )
)

"Funkcja rekurencyjnie licząca n-tą pochodną.
 Po każdej operacji różniczkowania próbuje uprości wyrażenie.
 Przykład użycia: (nth-differentiation '(sin (tg (ln (sqrt 2 x)))) 'x 2)
 Argumenty: expression - wyrażenie, z którego liczymy pochodną,
            variable - nazwa zmiennej, po której liczymy pochodną
            degree - stopień wyliczonej pochodnej"
(defn nth-differentiation
  [expression variable degree]
  (loop [expression expression variable variable degree degree]
    (if (< degree 1)
      (optimize-expression expression)
      (recur (optimize-expression(differentiation expression variable)) variable (- degree 1))))
)

"Funkcja liczy n-tą pochodną podanej w parametrze funkcji i oblicza wartości pochodnej w zadanych punktach
 Przykład użycia: (function-multiple-nth-differentiation-values '(sin (tg (ln (sqrt 2 x)))) 'x (1 2 3) 2)
 Argumenty: expression - wyrażenie, z którego liczymy pochodną,
            variable - nazwa zmiennej, po której liczymy pochodną
            argument-values - zadane wartości zmiennej
            degree - stopień wyliczonej pochodnej"
(defn function-multiple-nth-differentiation-values
  [expression variable argument-values degree]
    (loop [expression (nth-differentiation expression variable degree) variable variable argument-values
argument-values counted []]
      (do
        (if (= (count argument-values) 0)
```

```
          counted
          (recur expression variable (next argument-values) (conj counted (diff-eval expression variable (first
argument-values))))
        )
      )
    )
)

"Makro rekurencyjne drukujące informacje o użyciu zdefiniowanych w programi funkcji"
(defmacro info []
  (loop [
    func-names ["(differentiation function x) - function - funkcja, x - zmienna po ktorej funkcja bedzie
rozniczkowana"
      "(optimize-expression function) - funkcja optymalizuje wyrazenie - odpowiednio zamienia mnozenie przez 1
i przez 0"
      "(function-multiple-differentiation-values function x values) - funkcja oblicza pochodna danej funkcji, a
nastepnie oblicza wartosci obliczonej pochodnej w zadanych punktach"
      "(function-multiple-values function x values) - funkcja oblicza wartosci funkcji w zadanych punktach"
      "(function-multiple-values-macro function x values) - makro oblicza wartosci funkcji w zadanych punktach"
      "(nth-differentiation function x degree) - function - funkcja, x - zmienna po ktorej funkcja bedzie
rozniczkowana, degree - stopien pochodnej"
      "(function-multiple-nth-differentiation-values function x values degree) - funkcja oblicza n-ta pochodna
danej funckji, a nastepnie oblicza wartosci obliczonej pochodnej w zadanych punktach"
      "(function-multiple-differentiation-values-macro function x values) - makro oblicza pochodna danej
funkcji, a nastepnie oblicza wartosci obliczonej pochodnej w zadanych punktach"
      ]
    result ['do]]
    (if (= 0 (count func-names))
      (eval (apply list result))
      (recur (next func-names) (conj (conj result (list 'println (first func-names))) (list 'println ""))))
    )
  )
)

"Makro rekurencyjne wyznaczające wartość funkcji dla wszystkich zadanych wartości zmiennej
 Przykład użycia: (function-multiple-values-macro '(ln x) 'x (1 2 3 4))
 Argumenty: expression - wyrażenie do ewaluacji,
            variable - zmienna, pod którą będziemy podstawiać wartości,
            values - lista wartości"
(defmacro function-multiple-values-macro
  [expression variable values]
  ;(println expression variable values)
  (loop [expression expression variable variable values values result []]
    (if (= 0 (count values))
      result
      (do
        ;(println result)
        (recur expression variable (next values) (conj result (list diff-eval expression variable (first
values))))
      )
    )
  )
)

"Makro rekurencyjne wyliczające pochodną funkcji oraz wyznaczające wartości pochodnej dla kilku zadanych
wartości
 Przykład użycia: (function-multiple-differentiation-values-macro '(ln x) 'x (1 2 3 4))
 Argumenty: expression - wyrażenie do ewaluacji,
            variable - zmienna, pod którą będziemy podstawiać wartości,
            values - lista wartości"
(defmacro function-multiple-differentiation-values-macro
  [expression variable values]
  ;(println expression variable values)
  (loop [expression (list differentiation expression variable) variable variable values values result []]
    (if (= 0 (count values))
      result
      (do
        ;(println result)
        (recur expression variable (next values) (conj result (list diff-eval expression variable (first
values))))
      )
```

```
    )
  )
)

(defn -main
  [& args]
  (ns symbolic-differentiation.core)
  (def y 3)
  (def x 5)
  ;(macroexpand info)
  (println "Informacje o użytych funkcjach:")
  (info)
  (println "2-ga pochodna '(sin (tg (ln (sqrt 2 x)))):")
  (println (nth-differentiation '(sin (tg (ln (sqrt 2 x)))) 'x 2))
  (println "Wartości funkcji '(ln x) dla x równego kolejno (1 2 3 4):")
  (println (function-multiple-values-macro '(ln x) 'x (1 2 3 4)))
  (println "Wartości pochodnej funkcji '(ln x) dla x równego kolejno (1 2 3 4):")
  (println (function-multiple-differentiation-values-macro '(ln x) 'x (1 2 3 4)))
  (println "Pochodna wyrażenia '(ln (sin (tg (* 3 x)))) po uproszczeniu:")
  (println (optimize-expression (differentiation '(ln (sin (tg (* 3 x)))) 'x)))
  (println "Wartość '(ln (sin (tg (* 3 x)))) dla x = 5 (zdefiniowany powyżej):")
  (println (eval (optimize-expression (differentiation '(ln (sin (tg (* 3 x)))) 'x))))
  (println "Pochodna '(ln (tg x)):")
  (println (differentiation '(ln (tg x)) 'x))
  (println "Wartość pochodnej z  '(ln (tg x)) dla x = 5:")
  (println (eval (differentiation '(ln (tg x)) 'x)))
  (println "Pochodna '(ln (tg x)) po uproszczeniu:")
  (println (optimize-expression (differentiation '(ln (tg x)) 'x)))
  (println "Wartość pochodnej z '(ln (tg x)) dla x = 5:")
  (println (eval (optimize-expression (differentiation '(ln (tg x)) 'x))))
  (println "Wartości pochodnej z '(ln (tg x)) dla x równych kolejno (1 2 3 4):")
  (println (function-multiple-differentiation-values '(ln (tg x)) 'x '(1 2 3 4)))
  (println "Wartości funkcji '(* y (ln x)) dla x = (1 2 3 4) oraz y = 3 (zdefiniowany powyżej):")
  (println (function-multiple-values '(* y (ln x)) 'x '(1 2 3 4)))
  (println "Wartość pochodnej '(sin (* y (tg x))) dla x = 5:")
  (println (eval (differentiation '(sin (* y (tg x))) 'x)))
)
```