



POLITECHNIKA WARSZAWSKA
WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH



Dokumentacja projektu

<i>Przedmiot:</i>	Lisp i Prolog
<i>Projekt:</i>	Lisp: Różniczkowanie symboliczne funkcji jednej zmiennej
<i>Autorzy:</i>	Maciej Jakubiak Miłosz Pączkiewicz Bartłomiej Śmietanka

Warszawa, dnia 21 listopada 2018

```

1 (ns symbolic-differentiation.core
2   (:gen-class)
3 )
4
5 "Odwołania do matematycznych funkcji z biblioteki standardowej Clojure,
6  liczba parametrów zależy od funkcji
7  Przykła użycia: (log 2 4)"
8 (defn sin [x] (Math/sin x))
9 (defn cos [x] (Math/cos x))
10 (defn tg [x] (Math/tan x))
11 (defn ctg [x] (/ 1 (Math/tan x)))
12 (defn sec [x] (/ 1 (Math/cos x)))
13 (defn csc [x] (/ 1 (Math/sin x)))
14 (defn exp [x] (Math/exp x))
15 (defn ln [x] (Math/log x))
16 (defn log [x, y] (/ (Math/log y) (Math/log x)))
17 (defn sqrt [x, y] (Math/pow x (/ 1 y)))
18 (defn arcsin [x] (Math/asin x))
19 (defn arccos [x] (Math/acos x))
20 (defn arctg [x] (Math/atan x))
21 (defn arcctg [x] (Math/atan (/ 1 x)))
22 (defn arcsec [x] (Math/acos (/ 1 x)))
23 (defn arccsc [x] (Math/asin (/ 1 x)))
24 (defn sinh [x] (Math/sinh x))
25 (defn cosh [x] (Math/cosh x))
26 (defn tgh [x] (Math/tanh x))
27 (defn ctgh [x] (/ (cosh x) (sinh x)))
28 (defn sech [x] (/ 1 (cosh x)))
29 (defn csch [x] (/ 1 (sinh x)))
30 (defn pow [x, y] (Math/pow x y))
31
32 "Funkcje sprawdzające czy odpowiadające wyrażenie jest prawidłowo sformułowane
33  Przykład użycia: (pow 2 3) => true
34                    (sin) => false - funkcja sin przyjmuje 1 argument"
35 (defn sin? [x] (and (= (count x) 2) (= (first x) 'sin)))
36 (defn cos? [x] (and (= (count x) 2) (= (first x) 'cos)))
37 (defn tg? [x] (and (= (count x) 2) (= (first x) 'tg)))
38 (defn ctg? [x] (and (= (count x) 2) (= (first x) 'ctg)))
39 (defn sec? [x] (and (= (count x) 2) (= (first x) 'sec)))
40 (defn csc? [x] (and (= (count x) 2) (= (first x) 'csc)))
41 (defn exp? [x] (and (= (count x) 2) (= (first x) 'exp)))
42 (defn ln? [x] (and (= (count x) 2) (= (first x) 'ln)))
43 (defn log? [x] (and (= (count x) 3) (= (first x) 'log)))
44 (defn sqrt? [x] (and (= (count x) 3) (= (first x) 'sqrt)))
45 (defn arcsin? [x] (and (= (count x) 2) (= (first x) 'arcsin)))
46 (defn arccos? [x] (and (= (count x) 2) (= (first x) 'arccos)))
47 (defn arctg? [x] (and (= (count x) 2) (= (first x) 'arctg)))
48 (defn arcctg? [x] (and (= (count x) 2) (= (first x) 'arcctg)))
49 (defn arcsec? [x] (and (= (count x) 2) (= (first x) 'arcsec)))
50 (defn arccsc? [x] (and (= (count x) 2) (= (first x) 'arccsc)))
51 (defn sinh? [x] (and (= (count x) 2) (= (first x) 'sinh)))
52 (defn cosh? [x] (and (= (count x) 2) (= (first x) 'cosh)))
53 (defn tgh? [x] (and (= (count x) 2) (= (first x) 'tgh)))
54 (defn ctgh? [x] (and (= (count x) 2) (= (first x) 'ctgh)))
55 (defn sech? [x] (and (= (count x) 2) (= (first x) 'sech)))
56 (defn csch? [x] (and (= (count x) 2) (= (first x) 'csch)))
57 (defn pow? [x] (and (= (count x) 3) (= (first x) 'pow)))
58 (defn addition? [x] (and (= (count x) 3) (= (first x) '+)))
59 (defn subtraction? [x] (and (= (count x) 3) (= (first x) '-)))
60 (defn multiplication? [x] (and (= (count x) 3) (= (first x) '*)))
61 (defn division? [x] (and (= (count x) 3) (= (first x) '/)))
62
63 "Główna funkcja wyznaczająca pochodną podanego wyrażenia po określonej zmiennej
64  Przykład użycia: (differentiation '(ln (tg x)) 'x)
65                    => (* (* 1 (/ 1 (pow (cos x) 2))) (/ 1 (tg x)))
66  Argumenty: expression - wyrażenie, z którego liczymy pochodną,
67             variable - nazwa zmiennej, po której liczymy pochodną"
68 (defn differentiation
69   [expression variable]
70   (cond
71     (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 wewnętrznego"
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             )
427           )
428       )
429   )

```

```

427         (recur str open true (+ index 1))
428     )
429 )
430 )
431 )
432 )
433 )
434 )
435
436 "Funkcja rekurencyjnie usuwająca niepotrzebne nawiasy z wyrażenia.
437 Funkcja do użytku wewnętrznego"
438 (defn remove_doubled_brackets
439   [str]
440   (loop [orig str res (remove_doubled_brackets_priv orig)]
441     (if (= orig res)
442       res
443       (recur res (remove_doubled_brackets_priv res)))
444   )
445 )
446 )
447
448 "Główna funkcja służąca do uproszczenia wzoru otrzymanej pochodnej.
449 Wykorzystuje powyższe funkcje: optimize-expression_priv, remove_doubled_brackets
450 Przykład użycia: (optimize-expression (differentiation '(ln (tg x)) 'x))
451 Argumenty: expression - wyrażenie, z którego liczymy pochodną"
452 (defn optimize-expression
453   [expression]
454   (def __optimized__ [])
455   (loop [expression expression result (optimize-expression_priv expression)]
456     (do
457       (def __optimized__ [])
458       (let
459         [res (read-string (clojure.string/replace (clojure.string/join " " (remove_doubled_brackets result))
460 (re-pattern "\\(\\s0\\s\\)") "0"))]
461         (do
462           (if (= res expression)
463             res
464             (recur res (optimize-expression_priv res)))
465         )
466       )
467     )
468   )
469 )
470 )
471
472 "Funkcja rekurencyjnie licząca n-tą pochodną.
473 Po każdej operacji różniczkowania próbuje uprościć wyrażenie.
474 Przykład użycia: (nth-differentiation '(sin (tg (ln (sqrt 2 x)))) 'x 2)
475 Argumenty: expression - wyrażenie, z którego liczymy pochodną,
476           variable - nazwa zmiennej, po której liczymy pochodną
477           degree - stopień wyliczonej pochodnej"
478 (defn nth-differentiation
479   [expression variable degree]
480   (loop [expression expression variable variable degree degree]
481     (if (< degree 1)
482       (optimize-expression expression)
483       (recur (optimize-expression(differentiation expression variable)) variable (- degree 1))))
484 )
485
486 "Funkcja liczy n-tą pochodną podanej w parametrze funkcji i oblicza wartości pochodnej w zadanych punktach
487 Przykład użycia: (function-multiple-nth-differentiation-values '(sin (tg (ln (sqrt 2 x)))) 'x (1 2 3) 2)
488 Argumenty: expression - wyrażenie, z którego liczymy pochodną,
489           variable - nazwa zmiennej, po której liczymy pochodną
490           argument-values - zadane wartości zmiennej
491           degree - stopień wyliczonej pochodnej"
492 (defn function-multiple-nth-differentiation-values
493   [expression variable argument-values degree]
494   (loop [expression (nth-differentiation expression variable degree) variable variable argument-values
495 argument-values counted []]
496     (do
497       (if (= (count argument-values) 0)

```



```

498         counted
499         (recur expression variable (next argument-values) (conj counted (diff-eval expression variable (first
500 argument-values))))))
501     )
502 )
503 )
504 )
505
506 "Makro rekurencyjne drukujące informacje o użyciu zdefiniowanych w programi funkcji"
507 (defmacro info []
508   (loop [
509     func-names ["(differentiation function x) - function - funkcja, x - zmienna po której funkcja będzie
510 rozniczkowana"
511       "(optimize-expression function) - funkcja optymalizuje wyrażenie - odpowiednio zamienia mnożenie przez 1
512 i przez 0"
513       "(function-multiple-differentiation-values function x values) - funkcja oblicza pochodną danej funkcji, a
514 następnie oblicza wartości obliczonej pochodnej w zadanych punktach"
515       "(function-multiple-values function x values) - funkcja oblicza wartości funkcji w zadanych punktach"
516       "(function-multiple-values-macro function x values) - makro oblicza wartości funkcji w zadanych punktach"
517       "(nth-differentiation function x degree) - function - funkcja, x - zmienna po której funkcja będzie
518 rozniczkowana, degree - stopień pochodnej"
519       "(function-multiple-nth-differentiation-values function x values degree) - funkcja oblicza n-tą pochodną
520 danej funkcji, a następnie oblicza wartości obliczonej pochodnej w zadanych punktach"
521       "(function-multiple-differentiation-values-macro function x values) - makro oblicza pochodną danej
522 funkcji, a następnie oblicza wartości obliczonej pochodnej w zadanych punktach"
523     ]
524     result ['do]]
525     (if (= 0 (count func-names))
526       (eval (apply list result))
527       (recur (next func-names) (conj (conj result (list 'println (first func-names))) (list 'println ""))))
528     )
529   )
530 )
531
532 "Makro rekurencyjne wyznaczające wartość funkcji dla wszystkich zadanych wartości zmiennej
533 Przykład użycia: (function-multiple-values-macro '(ln x) 'x (1 2 3 4))
534 Argumenty: expression - wyrażenie do ewaluacji,
535           variable - zmienna, pod którą będziemy podstawiać wartości,
536           values - lista wartości"
537 (defmacro function-multiple-values-macro
538   [expression variable values]
539   ;(println expression variable values)
540   (loop [expression expression variable variable values values result []]
541     (if (= 0 (count values))
542       result
543       (do
544         ;(println result)
545         (recur expression variable (next values) (conj result (list diff-eval expression variable (first
546 values))))))
547     )
548   )
549 )
550 )
551
552 "Makro rekurencyjne wyliczające pochodną funkcji oraz wyznaczające wartości pochodnej dla kilku zadanych
553 wartości
554 Przykład użycia: (function-multiple-differentiation-values-macro '(ln x) 'x (1 2 3 4))
555 Argumenty: expression - wyrażenie do ewaluacji,
556           variable - zmienna, pod którą będziemy podstawiać wartości,
557           values - lista wartości"
558 (defmacro function-multiple-differentiation-values-macro
559   [expression variable values]
560   ;(println expression variable values)
561   (loop [expression (list differentiation expression variable) variable variable values values result []]
562     (if (= 0 (count values))
563       result
564       (do
565         ;(println result)
566         (recur expression variable (next values) (conj result (list diff-eval expression variable (first
567 values))))))
568     )

```

```
569 )
570 )
571 )
572
573 (defn -main
574   [& args]
575   (ns symbolic-differentiation.core)
576   (def y 3)
577   (def x 5)
578   ;(macroexpand info)
579   (println "Informacje o użytych funkcjach:")
580   (info)
581   (println "2-ga pochodna '(sin (tg (ln (sqrt 2 x)))):")
582   (println (nth-differentiation '(sin (tg (ln (sqrt 2 x)))) 'x 2))
583   (println "Wartości funkcji '(ln x) dla x równego kolejno (1 2 3 4):")
584   (println (function-multiple-values-macro '(ln x) 'x (1 2 3 4)))
585   (println "Wartości pochodnej funkcji '(ln x) dla x równego kolejno (1 2 3 4):")
586   (println (function-multiple-differentiation-values-macro '(ln x) 'x (1 2 3 4)))
587   (println "Pochodna wyrażenia '(ln (sin (tg (* 3 x)))) po uproszczeniu:")
588   (println (optimize-expression (differentiation '(ln (sin (tg (* 3 x)))) 'x)))
589   (println "Wartość '(ln (sin (tg (* 3 x)))) dla x = 5 (zdefiniowany powyżej):")
590   (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)))
)
```