Programming Paradigms Fall 2021
Homework Assignment №1

Innopolis University

October 26, 2021

About this assignment

You are encouraged to use homework as a playground: don't just make some code that solves the problem, but instead try to produce a readable, concise and well-documented solution.

Try to divide a problem until you arrive at simple tasks. Create small functions for every small logical task and combine those functions to build a complex solution. Try not to repeat yourself: for every logical task try to have just one small function and reuse it in multiple places.

The assignment is split into mutliple exercises that have complexity specified in terms of stars ($\star$). The more stars an exercise has the more difficult it is. Exercises with three or more stars ($\star\star\star$) might be really challenging, so please make sure you are done with simple exercises before trying out the more difficult ones.

The assignment provides clear instructions and some examples. However, you are welcome to make the result extra pretty or add more functionality as long as it does not change significantly the original problem and does not make the code *too complex*.

Submit a homework with all solutions as a single file. The file should work by loading it in DrRacket development environment and simply running it.

1 Symbolic differentiation

1.1 Polynomial expressions

In this assignment you will need to implement the tools for symbolic differentiation of expressions. Symbolic differentiation computes derivative of a given expression symbolically (i.e. as an expression), for example, given $(1+2x)(x+y)$ the computed partial derivative with respect to variable y is $1+2x$.

For start we will consider only expressions involving numeric constants, variables, binary addition and multiplication. The expressions will be given as valid Racket terms, for example:

```
'(+ 1 x)                        ;  1 + x
'(* 2 y)                        ;  2y
'(* (+ 1 (* 2 x)) (+ x y))      ;  (1 + 2x)(x + y)
```

To work with these expressions, you need to be able to traverse its structure. You can use `number?` predicate to check whether an expression is a number.

Exercise 1.1 ($\star$)**.** Implement the following predicates and functions:

```
(define (variable? expr) ...)     ; check whether a given expression is a variable
(define (sum? expr) ...)          ; check whether a given expression is a sum
(define (summand-1 expr) ...)     ; extract first summand from a sum
(define (summand-2 expr) ...)     ; extract second summand from a sum
(define (product? expr) ...)      ; check whether a given expression is a product
(define (multiplier-1 expr) ...)  ; extract first multiplier from a product
(define (multiplier-2 expr) ...)  ; extract second multipler from a product
```

Now, whenever we have an expression we can check what kind of expression we have and decompose it into its constituent subexpressions.

Exercise 1.2 ($\star$)**.** Implement a recursive function `derivative` that computes a symbolic derivative of a given expression with respect to a given variable. At this point you are not expected to simplify expressions after differentiation:

```
(derivative '(+ 1 x) 'x)
; '(+ 0 1)

(derivative '(* 2 y) 'y)
; '(+ (* 0 y) (* 2 1))

(derivative '(* (+ x y) (+ x (+ x x))) 'x)
; '(+ (* (+ 1 0) (+ x (+ x x))) (* (+ x y) (+ 1 (+ 1 1))))
```

Exercise 1.3 ($\star\star$)**.** Implement a recursive function `simplify` that simplifies an expression using the following rules:

1. $0 + e = e$ for all expressions e,
2. $e + 0 = e$ for all expressions e,
3. $1 * e = e$ for all expressions e,
4. $e * 1 = e$ for all expressions e,
5. $0 * e = 0$ for all expressions e,
6. $e * 0 = 0$ for all expressions e,
7. sums and products of numeric constants should be computed.

Examples:

```
(simplify '(+ 0 1))
; 1

(simplify '(+ (* 0 y) (* 2 1)))
; 2

(simplify '(+ (* (+ 1 0) (+ x (+ x x))) (* (+ x y) (+ 1 (+ 1 1)))))
; '(+ (+ x (+ x x)) (* (+ x y) 3))
```

Hint: it might be easier to implement a non-recursive function `simplify-at-root` *that only simplifies expression if it matches exactly left-hand side of one of the rules. Then use* `simplify-at-root` *to implement a recursive function* `simplify`*.*

Exercise 1.4 ($\star\star\star$)**.** Implement function `normalize` that simplifies any expression down to a polynomial of its variables. To achieve that you need to open parentheses using distributive property of multiplication over addition: $x(y + z) = xy + xz$. You will also need to simplify similar terms: $xy + 2yz + 3xy = 4xy + 2yz$.

Exercise 1.5 ($\star$)**.** Implement a recursive function `to-infix` that converts an expression into an infix form:

```
(to-infix '(+ (+ x (+ x x)) (* (+ x y) 3))
; '((x + (x + x)) + ((x + y) * 3)
```

1.2 More functions

Exercise 1.6 ($\star$)**.** Update functions `derivative` and `simplify` to support the following functions (here e, e_1 and e_2 denote arbitary expressions):

1. exponentiation: $e_1^{e_2}$
2. trigonometric functions: $\sin e$, $\cos e$, $\tan e$
3. natural logarithm: $\log e$

Exercise 1.7 ($\star\star$)**.** Update functions `derivative` and `simplify` to support the following polyvariadic[1] sums and products:

```
(derivative '(+ 1 x y (* x y z)) 'x)
; '(+ 0 1 0 (+ (* 1 y z) (* x 0 z) (* x y 0)))

(simplify '(+ 0 1 0 (+ (* 1 y z) (* x 0 z) (* x y 0))))
; '(+ 1 (* y z))
```

1.3 Gradient

Exercise 1.8 ($\star\star$)**.** Implement a function `variables-of` that returns a (sorted) list of distinct variables used in a given expression:

```
(variables-of '(+ 1 x y (* x y z)))
; '(x y z)
```

Exercise 1.9 ($\star\star$)**.** Implement a function `gradient` that returns a gradient of a multivariable expression (given explicitly the list of variables). Recall that the gradient ∇f is a vector consisting of partial derivatives:

$$\nabla e = \left[\frac{\partial e}{\partial x_1} \; \frac{\partial e}{\partial x_2} \; \cdots \; \frac{\partial e}{\partial x_n}\right]^\top$$

Represent gradient in Racket as a list of expressions:

```
(gradient '(+ 1 x y (* x y z)) '(x y z))
; '((+ 1 (* y z)) (+ 1 (* x z)) (* x y))
```

[1] "polyvariadic" means that a function or operator can support arbitrary number of arguments; for example, `+` in Racket can accept as many arguments, as user wants: `(+ 1 2 3 4)` computes to $1 + 2 + 3 + 4 = 10$.