Лабораторная работа №5
«Опять производные»

Предварительные сведения

В ходе выполнения данной работы необходимо написать решение 15 задач (68 функций) на языке Lisp. Все задания связаны с символьными вычислениями математических выражений, представленных в виде выражений на Lisp.

Прежде чем приступать к выполнению заданий, ознакомьтесь с замечаниями, изложенными в пункте 2 данного описания.

Предполагаем, что в виде списка (или атома) задано выражение на языке Lisp, представляющее запись математического выражения, например,

```
(+ (* x (sqrt x) (sqrt y)) (/ (tan x) (log (cos 1.3) x) (* 4 x)))
```

для выражения

$$x\sqrt{x}\sqrt{y} + \frac{\frac{\operatorname{tg} x}{\log_x \cos 1.3}}{4x}$$

В выражении могут встречаться числовые константы, вызовы функций `+`, `-`, `*`, `/`, `expt`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp` и `log`, а так же имена переменных (символьные атомы, отличные от чисел, не являющиеся вызовами функций). В настоящей лабораторной работе вам предстоит описать набор функций для преобразования таких выражений, а именно для вычисления производной выражения по заданной переменной и для упрощения выражений. Для этих целей понадобятся вспомогательные определения: предикаты для определения типа выражения, функции для конструирования выражений из аргументов и функции нормализации выражений — приведения выражения к форме, пригодной для взятия производной и упрощения.

1. Задания

1. Нам потребуется набор вспомогательных функций предикатов, выдающих `T`, если заданное выражение является выражением заданного типа (под типом здесь мы понимаем разновидность выражения).

   Опишите 16 функций: `0?`, `1?`, `+?`, `-?`, `*?`, `/?`, `expt?`, `sqrt?`, `sin?`, `cos?`, `tan?`, `asin?`, `acos?`, `atan?`, `exp?` и `log?`. Каждая из перечисленных функций должна принимать один аргумент.

   Первые две функции должны возвращать `T` тогда и только тогда, когда аргумент является числом со значением равным нулю или единице соответственно. Последующие функции должны возвращать `T` тогда и только тогда, когда аргумент является вызовом функции, упомянутой в наименовании, с корректным для языка Lisp количеством аргументов.

   Например, следующий вызов

   ```
   (+? '(+ 2 A (* 3 5) (* 4 8))) ; передается сумма четырех слагаемых
   ```

   должен вернуть `T`, а каждый из вызовов

   ```
   (sin? '(+ 2 A (* 3 5) (* 4 8))) ; переданный список - не вызов синуса
   (log? '(log 2 (5 6) 45 b))      ; у вызова логарифма некорректное число аргументов
   ```

   должен вернуть `NIL`.

   Стоит напомнить, что у сложения и умножения в Lisp — любое количество аргументов (возможно 0). У вычитания и деления — не менее одного аргумента. У функции `expt` — точно два аргумента. У функций `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan` и `exp` — по одному. У `log` — один или два.

   Определение каждой функции, кроме последней — одна-две строки кода. Определение функции `log?` в типовом решении — 4 строки кода.

2. Опишите 14 функций: `make+`, `make-`, `make*`, `make/`, `makeexpt`, `makesqrt`, `makesin`, `makecos`, `maketan`, `makeasin`, `makeacos`, `makeatan`, `makeexp` и `makelog`.

   Каждая из функций этого задания должна конструировать для заданного набора аргументов вызов функции, упомянутой в названии. Каждая из функций должна быть определена для корректного числа параметров.

   Например, вызов

   ```
   (make- 25 45 'A '(+ 3 9))
   ```

   должен вернуть следующий список

   ```
   (- 25 45 A (+ 3 9))
   ```

   представляющий выражение для разности. Вызовы подобные следующим

   ```
   (makesin 13 'B)       ; слишком много аргументов для вызова синуса
   (makelog '(+ 5 B) 'A) ; слишком много аргументов для вызова логарифма
   (make-)               ; слишком мало аргументов для разности
   ```

   должны заканчиваться системной ошибкой.

   Определение каждой функции, кроме последней — одна-две строки кода. Определение функции `makelog` в типовом решении — 2 строки кода.

3. В шаблоне решения уже представлена функция `normalize`. Следующий набор функций — 14 вспомогательных функций для `normalize`: `+-normalize`, `--normalize`, `*-normalize`, `/-normalize`, `expt-normalize`, `sqrt-normalize`, `sin-normalize`, `cos-normalize`, `tan-normalize`, `asin-normalize`, `acos-normalize`, `atan-normalize`, `exp-normalize` и `log-normalize`. Каждой функции передается выражение, представляющее корректный вызов соответствующей функции. Нормализация будет заключаться в унификации количества аргументов функции и нормализации аргументов:

   `(+-normalize expr)` : Если `expr` — вызов функции `+` без аргументов, то нормализованным выражением должен быть ноль — `0`.
   Если `expr` — вызов

   ```
   (+ e1)
   ```

   (вызов функции с одним аргументом), то нормализованным выражением должно являться `e1'` — нормализованное выражение, полученное из `e1`.
   Если `expr` — вызов

   ```
   (+ e1 e2 ... e3 e4 e5)
   ```

   то он должен быть преобразован к виду

   ```
   (+ e1' (+ e2' ... (+ e3' (+ e4' e5'))...))
   ```

   где `ei'` обозначает нормализованное выражение, полученное из `ei`. То есть, после нормализации любой вызов функции `+` должен иметь два аргумента.

   `(--normalize expr)` : Функцию разности в нормализованном выражении использовать не будем. Если `expr` — вызов функции `-` с одним аргументом (унарный минус), то результатом должен быть вызов функции `*`, которой передается значение `-1` и нормализованный аргумент. Если `expr` — вызов

   ```
   (- e1 e2 ... e3 e4 e5)
   ```

   то он должен быть преобразован к виду

   ```
   (+ e1' (* -1 (+ e2' ... (+ e3' (+ e4' e5'))...)))
   ```

   где `ei'` обозначает нормализованное выражение, полученное из `ei`.

`(*-normalize expr)` : Если `expr` — вызов функции `*` без аргументов, то нормализованным выражением должна быть единица — `1`.
Если `expr` — вызов

```
(* e1)
```

(вызов функции с одним аргументом), то нормализованным выражением должно являться `e1'` — нормализованное выражение, полученное из `e1`.
Если `expr` — вызов

```
(* e1 e2 ... e3 e4 e5)
```

то он должен быть преобразован к виду

```
(* e1' (* e2' ... (* e3' (* e4' e5'))...))
```

где `ei'` обозначает нормализованное выражение, полученное из `ei`. То есть, после нормализации любой вызов функции `*` должен иметь два аргумента.

`(/-normalize expr)` : Если `expr` — вызов функции `/` с одним аргументом (обратный элемент), то результатом должен быть вызов функции `/` с нормализованным аргументом. Если `expr` — вызов

```
(/ e1 e2 ... e3 e4 e5)
```

то он должен быть преобразован к виду

```
(* e1' (/ (* e2' ... (* e3' (* e4' e5'))...)))
```

где `ei'` обозначает нормализованное выражение, полученное из `ei`. То есть, после нормализации любой вызов функции `/` должен иметь один аргумент.

`(sqrt-normalize expr)` : Принимая во внимание, что

$$\sqrt{x} = x^{\frac{1}{2}},$$

вызовы функции `sqrt` выразим через функцию возведения в степень `expt` со вторым аргументом, равным `1/2`. То есть, если `expr` — вызов `(sqrt e)`, то он должен быть преобразован к виду

```
(expt e' 1/2)
```

`(tan-normalize expr)` : Вместо дальнейшего использования функции тангенса `tan` будем использовать его выражение через синус и косинус:

$$\operatorname{tg} x = \frac{\sin x}{\cos x}$$

То есть, если `expr` — вызов `(tan e)`, то он должен быть преобразован к виду

```
(* (sin e') (/ (cos e')))
```

`(log-normalize expr)` : Функцию `log` будем использовать только с одним аргументом (натуральный логарифм). Для этого используем формулу приведения логарифма к другому основанию:

$$\log_b a = \frac{\log_c a}{\log_c b}.$$

То есть, если `expr` — вызов `(log e1 e2)`, то он должен быть преобразован к виду

```
(* (log e1') (/ (log e2')))
```

Функции: `expt-normalize`, `sin-normalize`, `cos-normalize`, `asin-normalize`, `acos-normalize`, `atan-normalize` и `exp-normalize` должны оставлять вызов функции в выражении на месте, только проведя нормализацию аргументов.

Определение каждой функции — от 2-х до 5-ти строк кода.

Следующие 6 заданий связаны с реализацией упрощения выражения с помощью функции `simplify`, уже представленной в шаблоне. Необходимо реализовать функции: `simplify+`, `simplify*`, `simplify/`, `simplifyexpt`, `simplify-fun-1`, `simplifyexp`, `simplifylog`, а так же вспомогательные для них функции: `simplify+-aux1`, `simplify+-aux2`, `simplify*-through+`, `simplify*-aux1`, `simplify*-aux2`, `simplify/-through*`.

Будем использовать следующие основные принципы упрощения:

- функции суммы и произведения представляются как функции двух или более аргументов. В упрощенном выражении не может быть операции сложения, в которой аргумент представляет собой сумму. В упрощенном выражении не может быть операции умножения, в которой аргумент представляет собой произведение;
- упрощенное выражение есть одно слагаемое или сумма слагаемых;
- каждое слагаемое представляет собой один множитель или произведение нескольких множителей;
- каждый множитель — или число, или имя переменной, или вызов сложной функции;
- вызов сложной функции — вызов одной из функций: `/` (обратный элемент), `expt`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp` или `log`. Аргументом такого вызова является упрощенное выражение;
- в упрощенном выражении отсутствуют в качестве подвыражений вызовы функций только от числовых аргументов. То есть, если в выражении можно вычислить какое-то подвыражение, то оно должно быть вычислено;
- у суммы может быть только один числовой аргумент, стоящий на первой позиции и не равный нулю;
- у произведения может быть только один числовой аргумент, стоящий на первой позиции и не равный нулю или единице.

4. В этом задании должно быть реализовано упрощение суммы. Основной функцией является функция `simplify+`, которой подается на вход нормализованное выражение. У функции `simplify+` есть две вспомогательные функции.

   Функция `(simplify+-aux1 expr n)` получает на вход два слагаемых: упрощенное выражение `expr` и число `n`, и строит из них упрощенное выражение для их суммы.

   Функция `(simplify+-aux2 expr +expr)` получает на вход два упрощенных слагаемых `expr` и `+expr`, причем известно, что `+expr` является суммой, и строит из них упрощенное выражение для их суммы.

   Так как аргумент `simplify+` — нормализованное выражение, то это выражение — сумма точно двух аргументов. Функция должна сначала упростить свои аргументы, после чего проанализировать то, что из них получилось после упрощения, и построить из них упрощенное выражение:

   - если одно из слагаемых вырождено в ноль, то выдать второе слагаемое, как результат;
   - если одно из слагаемых является числом, то передать управление функции `simplify+-aux1`;
   - если одно из слагаемых является суммой, то передать управление функции `simplify+-aux2`;
   - в остальных случаях составить и выдать сумму полученных упрощенных аргументов.

   Вызов функции `(simplify+-aux1 expr n)` должен анализировать выражение `expr` и

   - если `expr` является числом, то вычислить сумму `expr` и `n` и выдать ее в качестве результата;
   - если `expr` является суммой, то из набора ее слагаемых и из числа `n` нужно составить новую сумму и выдать ее в качестве результата. Причем, если первое слагаемое в `expr` является числом, то в новой сумме первым слагаемым должно быть значение суммы этого числа с `n`;
   - в остальных случаях составить и выдать сумму полученных упрощенных аргументов.

   Вызов функции `(simplify+-aux2 expr +expr)` должен анализировать выражение `expr` и

   - если `expr` является числом, то передать управление функции `simplify+-aux1`;
   - если `expr` является суммой, в которой первое слагаемое — число, то нужно добавить остальные слагаемые в конец суммы `+expr`, и передать управление функции `simplify+-aux1`;
   - если `expr` является суммой, а первым слагаемым в `+expr` является число, то нужно добавить остальные слагаемые `+expr` в конец суммы `expr`, и передать управление функции `simplify+-aux1`;
   - если `expr` является суммой, то из набора ее слагаемых и слагаемых суммы `+expr` нужно составить новую сумму и выдать ее в качестве результата;

- если `expr` — не сумма, то поставить это выражение в конец суммы `+expr` и выдать получившееся выражение.

Объем каждого решения — 10–15 строк.

5. В этом задании должно быть реализовано упрощение произведения. Основной функцией является функция `simplify*`, которой подается на вход нормализованное выражение. У функции `simplify*` есть три вспомогательные функции.

   Функция `(simplify*-through+ exprs n)` — вспомогательная функция, «раскрытие скобок» — получает список слагаемых `exprs` и множитель `n` и возвращает список слагаемых, домноженных на `n`.

   Функция `(simplify*-aux1 expr n)` получает на вход два множителя: упрощенное выражение `expr` и число `n`, и строит из них упрощенное выражение для их произведения.

   Функция `(simplify*-aux2 expr *expr)` получает на вход два упрощенных слагаемых `expr` и `*expr`, причем известно, что `*expr` является произведением, и строит из них упрощенное выражение для их произведения.

   Так как аргумент `simplify*` — нормализованное выражение, то это выражение — произведение точно двух множителей. Функция должна сначала упростить свои аргументы, после чего проанализировать то, что из них получилось после упрощения, и построить из них упрощенное выражение:

   - если один из множителей вырожден в ноль, то выдать ноль;
   - если один из множителей равен единице, то выдать второй множитель как результат;
   - если один из множителей является числом, то передать управление функции `simplify*-aux1`;
   - если один из множителей является произведением, то передать управление функции `simplify*-aux2`;
   - если один из множителей — вызов функции `/` (взятие обратного элемента), а аргумент этого вызова равен второму множителю (достаточно внешнего сравнения без дополнительного анализа), то выдать единицу.
   - если один из множителей является суммой, то результат получается в результате упрощения суммы, составленной из слагаемых, домноженных на второй множитель с помощью `simplify*-through+`.
   - в остальных случаях составить и выдать произведение полученных упрощенных аргументов.

   Вызов функции `(simplify*-through+ exprs n)` перебирает элементы списка `exprs`, составляя из каждого элемента и `n` произведение с помощью `make*`. Результат — список новых слагаемых.

   Например, вызов

```
(simplify*-through+ '(A B C D) 'E)
```

   должен вернуть список (порядок элементов может отличаться)

```
'((* A E) (* B E) (* C E) (* D E))
```

   Вызов функции `(simplify*-aux1 expr n)` должен анализировать выражение `expr`:

   - если `expr` является числом, то вычислить произведение `expr` и `n` и выдать его в качестве результата;
   - если `expr` является произведением, то из набора его множителей и из числа `n` нужно составить новое произведение и выдать его в качестве результата. Причем, если первый множитель в `expr` является числом, то в новом произведении первым множителем должно быть значение произведения этого числа с `n`;
   - если `expr` является суммой, то результат получается в результате упрощения суммы, составленной из слагаемых, домноженных на `n` с помощью `simplify*-through+`.
   - в остальных случаях составить и выдать произведение полученных упрощенных аргументов.

   Вызов функции `(simplify*-aux2 expr *expr)` должен анализировать выражение `expr`:

   - если `expr` является числом, то передать управление функции `simplify*-aux1`;
   - если `expr` является произведением, в котором первый множитель — число, то нужно добавить остальные множители в конец произведения `*expr` и передать управление функции `simplify*-aux1`;
   - если `expr` является произведением, а первым множителем в `*expr` является число, то нужно добавить остальные множители `*expr` в конец произведения `expr` и передать управление функции `simplify*-aux1`;

- если `expr` является произведением, то из набора его множителей и множителей произведения `*expr` нужно составить новое произведение и выдать его в качестве результата;
- если `expr` является суммой, то результат получается в результате упрощения суммы, составленной из слагаемых, домноженных на `*expr` с помощью `simplify*-through+`.
- если `expr` — не сумма и не произведение, то поставить это выражение последним множителем в `*expr` и выдать получившееся выражение.

Объем функции `simplify*-through+` в типовом решении — 4 строки. Объем остальных функций — 13–20 строк каждая.

6. Нужно реализовать функцию `simplify/` упрощения выражения для обратного элемента. У этой функции есть вспомогательная: `simplify/-through*`, получающая список множителей и возвращающая список обратных элементов для этих множителей.

   Функция `simplify/` должна сначала упрощать параметр вызова `/`, после чего:

   - если упрощенный параметр число, то выдать в результате обратное значение для этого числа;
   - если параметр оказался вызовом функции `/`, то выдать параметр этого вызова как результат;
   - если параметр оказался произведением, то результат получается упрощением произведения, составленного из обратных элементов множителей, полученных с помощью `simplify/-through*`;
   - иначе составить вызов функции `/` от упрощенного параметра и выдать его в качестве результата.

   Вызов функции `(simplify/-through* exprs)` перебирает элементы списка `exprs`, составляя из каждого элемента вызов функции `/` с помощью `make/`. Результат — список новых множителей.

   Например, вызов

   ```
   (simplify/-through* '(A B C D))
   ```

   должен вернуть список (порядок элементов может отличаться)

   ```
   '((/ A) (/ B) (/ C) (/ D))
   ```

   Количество строк в типовом решении — 7 и 4, соответственно.

7. Необходимо реализовать функцию упрощения возведения в степень `simplifyexpt`. Аргументом функции является выражение от двух параметров, которые нужно упростить, а затем проанализировать то, что из них получилось:

   - если второй упрощенный параметр равен нулю, то результат — единица;
   - если первый параметр равен нулю, то результат — ноль;
   - если первый параметр равен единице, то результат — единица;
   - если второй параметр равен единице, то результат — первый параметр;
   - если и первый и второй параметр являются числами, то результат — результат возведения первого числа в степень, заданную вторым числом;
   - если первый параметр является вызовом функции возведения в степень `(expt a b)`, то нужно составить произведение из `b` и второго параметра, составить с помощью `makeexpt` вызов возведения в степень `a` с полученным произведением в качестве показателя степени, упростить получившееся выражение и выдать в качестве результата;
   - если первый параметр является вызовом функции экспоненты `(exp b)`, то нужно составить произведение из `b` и второго параметра, составить с помощью `makeexp` вызов экспоненты с полученным произведением в качестве параметра, упростить получившееся выражение и выдать в качестве результата;
   - в противном случае составить вызов функции возведения в степень от упрощенных параметров.

   Объем типового решения — 14 строк.

8. В этом задании необходимо описать три оставшиеся функции для упрощения: `simplify-fun-1` — для упрощения тригонометрических функций, `simplifyexp` — для упрощения экспоненты и `simplifylog` — для упрощения логарифма.

   Функция `(simplify-fun-1 expr)` упрощает параметр полученного выражения, конструирует из него новое выражение, после чего, если параметр выражения выродился в число — выдает значение получившегося

выражения, а иначе выдает само выражение. Никаких дополнительных упрощений тригонометрических функций проводить не нужно.

Функция `(simplifyexp expr)` работает таким же образом, но делает дополнительную проверку получившегося после упрощения параметра. Если в результате упрощения параметром является логарифм, то аргумент логарифма выдается в качестве результата.

Функция `(symplifylog expr)` упрощает параметр полученного выражения, после чего:

- если параметр оказался числом, то выдается значение натурального логарифма от этого числа;
- если параметр — вызов функции экспоненты, то выдается аргумент этого вызова;
- если параметр — вызов функции возведения в степень, то конструируется произведение показателя степени и логарифма от основания и выдается результат упрощения получившегося выражения;
- в противном случае конструируется вызов логарифма от упрощенного параметра.

Каждая функция — 6–9 строк кода.

Остальные задания связаны с реализацией операции взятия производной нормализованного выражения с помощью функции `deriv`, уже представленной в шаблоне. Необходимо реализовать 11 функций: `+-deriv`, `*-deriv`, `/-deriv`, `expt-deriv`, `sin-deriv`, `cos-deriv`, `asin-deriv`, `acos-deriv`, `atan-deriv`, `exp-deriv`, `log-deriv`. Каждая из этих функций получает нормализованное выражение (представляющее собой вызов функции, упомянутой в наименовании) и имя переменной и должна возвращать выражение (без упрощений и нормализации) для производной выражения по заданной переменной.

9. Опишите функцию `(+-deriv expr x)`, реализующую правило взятия производной

$$(u+v)'_x = u'_x + v'_x.$$

Объем решения — три строки.

10. Опишите функцию `(*-deriv expr x)`, реализующую правило взятия производной

$$(uv)'_x = u'_x v + v'_x u.$$

Объем решения — 5–8 строк.

11. Опишите функцию `(/-deriv expr x)`, реализующую правило взятия производной

$$\left(\frac{1}{u}\right)'_x = -\frac{1}{u^2}u'_x.$$

Объем решения — 3–4 строки.

12. Опишите функцию `(expt-deriv expr x)`, реализующую правило взятия производной

$$(u^v)'_x = vu^{v-1}u'_x + u^v v'_x \log u.$$

Объем типового решения — 10 строк.

13. Опишите функции `(sin-deriv expr x)` и `(cos-deriv expr x)`, реализующие правила взятия производной

$$(\sin u)'_x = u'_x \cos u, \qquad (\cos u)'_x = -u'_x \sin u.$$

Объем каждой функции — 4–5 строк.

14. Опишите функции `(asin-deriv expr x)`, `(acos-deriv expr x)` и `(atan-deriv expr x)`, реализующие правила взятия производной

$$(\arcsin u)'_x = \frac{u'_x}{\sqrt{1-u^2}}, \qquad (\arccos u)'_x = -\frac{u'_x}{\sqrt{1-u^2}}, \qquad (\operatorname{arctg} u)'_x = \frac{u'_x}{1+u^2}.$$

Объем каждой функции — 3–4 строки.

15. Опишите функции `(exp-deriv expr x)` и `(log-deriv expr x)`, реализующие правила взятия производной

$$(e^u)'_x = e^u u'_x, \qquad (\log u)'_x = \frac{1}{u}u'_x.$$

Объем каждой функции — 3–4 строки.

Общий объем решения, включая строки, предварительно заданные в файле, должен составить порядка 500 строк.

2. Замечания по выполнению заданий

2.1. Необходимый минимум

Для выполнения работы потребуются сведения о следующих функциях, операциях и конструкциях:

- конструкция `defun` для определения функции;
- условные конструкции `if` и `cond`;
- конструкция `let`;
- функции `cons`, `car` и `cdr`, `list`, `append` для работы со списками;
- логические конструкции `and`, `or`, `not`;
- предикаты проверки типа аргумента `null`, `numberp`, `zerop`, `listp`;
- функции сравнения на равенство `eq`, `equal`, `=`;
- функции сравнения `>`, `>=`, `<`, `<=`, `/=`;
- функция вычисления длины списка `length`;
- функции для выполнения вычислений математических выражений `+`, `-`, `*`, `/`, `expt`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`;
- функция `eval` — интерпретатор выражения языка Lisp.

Информацию об этих и других конструкциях можно найти в конспекте, приведенном на портале.

2.2. Ограничения

При выполнении данной лабораторной работы нужно соблюдать следующие ограничения:

1. При описании функций нельзя использовать функции и конструкции кроме перечисленных в разделе 2.1 или предоставленных в шаблоне решения. Если вы считаете, что для выполнения какого-то из заданий необходима функция/конструкция, не описанная в разделе 2.1 — задайте вопрос на форуме «Лабораторная работа №5»;

2. Установки на обязательное использование хвостовой рекурсии в этой лабораторной работе нет. Напротив, в целях выразительности решения при реализации циклических процессов следует отдавать предпочтение реализации функций с обычной рекурсией.

3. НЕЛЬЗЯ описывать какие-либо ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ, за исключением тех, о которых явно говорится в задании;

4. все ограничения снимаются при описании тестов к программному коду.

2.3. Предостережения насчет решения

Решением каждой задачи должна быть функция с указанным именем и количеством параметров. Пример вызова для каждой функции можно найти в шаблоне тестов к программному коду.

Не следует делать предположений насчет задания, не сформулированных явно в условии. Если возникают сомнения — задайте вопрос на форуме «Лабораторная работа №5».