Функционалы, замыкания, мемоизация.

Символьное дифференцирование.

Артамонов Ю.Н.

Университет "Дубна"
филиал Котельники

21 апреля 2017 г.

Программирование, управляемое данными

Известно, что Лисп не делает особых различий между программой и данными. Интерпретатор Лиспа работает по схеме Read (чтение) — Eval (вычисление) — Print (печать полученного значения). Интерпретатор корректно воспринимает S-выражения: атомы, числа и списки. Поскольку программирование в Лиспе - это запись реализованных пользователем функций, которые сами являются списками, Лисп без ограничений может воспринимать программу как данные, а полученные в ходе переработки данные выполнять как программу. Указанное свойство открывает массу возможностей для новых технологий программирования. Одна из таких возможностей — *программирование, управляемое данными* - это метод проектирования программ, в котором внешние к программе данные используются с целью управления работой программы или сами интерпретируются в качестве программы. Эта технология позволяет унифицировать одну и ту же программу для обработки данных разных типов. Например, можно разработать систему, которая умеет складывать, вычитать, умножать, делить числа, комплексные числа, многочлены. При этом программа сама определяет тип данных и последовательность их обработки, т.е. сама управляет обработкой данных.

Определение функционалов

Основу технологии программирования, управляемого данными, составляет способность Лисп перерабатывать данные в программы, которая отчасти реализуется с помощью функционалов.
Функционалом называют функцию, которая имеет в качестве аргумента некоторую другую функцию (в частном случае саму себя). Различие между обычным аргументом и так называемым функциональным аргументом состоит в том, что обычный аргумент используется лишь как объект, участвующий в вычислениях, а функциональный аргумент используется как средство, определяющее вычисления, которое применяется к другим аргументам.
Иногда функция не просто имеет в качестве своего аргумента другую функцию, но сама может возвращать в качестве своего значения некоторую функцию, такая функция называется функцией с функциональным значением. Функционалы и функции с функциональным аргументом образуют так называемые *функции высшего порядка*.
Принято различать два типа функционалов: *применяющие* и *отображающие функционалы*.

Применяющий функционал apply

Применяющие функционалы — это функционалы, применяющие функциональный аргумент к другим своим аргументам. Рассмотрим их реализацию в среде Лисп в виде двух разновидностей:

```
(defun My_apply (f L)
  ((lambda (fun x) (eval  (cons fun  x))) f L))
```

В теле данной функции реализуется лямбда-вызов, где в выражение (lambda (fun x) (eval (cons fun x))) вместо fun и x подставляются фактические f и L соответственно. После (cons fun x) формируется список (f L), который потом интерпретируется как функция f с аргументом L и вычисляется с помощью Eval. В качестве f мы можем подставлять любую функцию, интерпетатор сам определяет как обрабатывать данные, содержащиеся в списке L в зависимости от вида функции f. Приведем примеры использования этой функции

```
>(My_apply '+ '(1 2 3 4))
10
>(My_apply '* '(1 2 3 4))
24
>(My_apply 'eval '(+ 1 2))
EVAL: слишком много аргументов для EVAL: (EVAL + 1 2)
```

Применяющий функционал apply (продолжение)

В последнем случае нужно изменять нашу функцию My_apply, т.к. Eval должна работать со списком. Таким образом, мы сталкиваемся с проблемой изменения процесса вычисления в зависимости от типа аргумента. Мы можем так модифицировать нашу функцию для этого случая:

```
(defun My_apply (f L)
  ((lambda (fun x)
     (eval (list fun (list 'quote x)))) f L))
```

Теперь все заработает:

```
> (My_apply 'eval '(+ 1 2))
3
>(My_apply 'car '(+ 1 2))
+
>(My_apply 'cdr '(+ 1 2))
(1 2)
```

Но теперь не будут работать первоначальные примеры, поэтому, конечно, желательно унифицировать функцию My_apply для любых функциональных аргументов f, т.е. использовать технологию программирования, управляемую данными. Мы не будем здесь этим заниматься а лишь отметим, что существует и встроенный применяющий функционал apply.

Примеры использования встроенного apply

```
>(setq r '+)
+
>(apply r '(1 2))
3
>(apply '* '(1 2 3 4))
24
>(apply 'apply '(+ (1 2)))
3
>(apply 'funcall '(+ 1 2 3))
6
```

В последних двух примерах apply применяется к самому себе, а также используется другой применяющий функционал funcall

Применяющий функционал funcall

Следующая разновидность применяющего функционала:

```
(defun My_funcall (f &rest L)
  ((lambda (fun x) (eval (cons fun  x))) f L))
```

отличается от My_apply тем, что позволяет применять функцию не к списку, а к набору аргументов:

```
>(My_funcall '+ 1 2 3 4)
10
>(My_fucall '* 1 2 3 4)
24
```

Этот эффект достигается использованием специального ключа &rest - переменное количество аргументов, который внутренне представляет все, что идет после первого аргумента списком. Применяющий функционал с аналогичным действием встроен в среду Лисп - это funcall. Рассмотрим примеры его использования:

```
>(funcall 'cons 2 3)
(2 . 3)
>(setq cons '+)
+
>(funcall cons 2 3)
5
```

Отображающие функционалы

Следующий класс функционалов, которые широко используются в практическом программировании в Лисп — это отображающие функционалы. Например, нам нужно из списка чисел получить список квадратов этих чисел. Конечно, очень легко построить рекурсивное определение этой функции:

```
(defun square (L)
  (cond
    ((Null L) nil)
    (t (cons (* (car L) (car L)) (square (cdr L))))))
```

Но если нам потребуется затем построить список из квадратных корней, то потребуется модифицировать исходную функцию. В общем же случае может потребоваться применить заранее неизвестную функцию к списку аргументов и сформировать список из результатов. В этом случае как раз помогает отображающий функционал. Реализуем данный функционал:

```
(defun my_mapcar (f L)
  (cond
    ((Null L) nil)
    (t (cons (funcall f (car L)) (my_mapcar f (cdr L))))))
```

Отображающие функционалы (примеры использования mapcar)

Рассмотрим примеры использования:

```
>(my_mapcar 'sqrt '(1 2 3 4 5))
(1 1.4142135 1.7320508 2 2.236068)
>(my_mapcar 'atom '(1 (2 3) 4 5))
(T NIL T T)
>(my_mapcar 'list '(1 2 3 4))
((1) (2) (3) (4))
>(mapcar 'cons '(1 2 3) '(1 2 3))
((1 . 1) (2 . 2) (3 . 3))
>(my_mapcar 'random '(20 30 40 70))
(2 19 9 9)
>(my_mapcar (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
```

Последний пример показывает, что функцию, которая будет применяться к списку аргументов можно определить прямо в теле вызова функционала в виде лямбда выражения. Действительно, нет смысла определять функцию взятия квадрата числа в программе, если она требуется однократно.

Отображающие функционалы (maplist)

Следующий отображающий функционал My_maplist действует подобно My_mapcar, но действия осуществляются не над элементами списка, а над последовательными хвостами списка. Реализация может иметь следующий вид:

```
(defun My_maplist (f L)
  (cond
    ((Null L) nil)
    (t (cons (funcall f L) (My_maplist f (cdr L))))))
```

Примеры использования:

```
>(My_maplist 'reverse '(1 2 3 4 5 6))
((6 5 4 3 2 1) (6 5 4 3 2) (6 5 4 3) (6 5 4) (6 5) (6))
>(My_maplist 'length '(5 6 7 8 9))
(5 4 3 2 1)
>(My_maplist 'cdr '(5 6 7 8 9))
((6 7 8 9) (7 8 9) (8 9) (9) NIL)
```

Отображающие функционалы (примеры использования maplist)

```
>(maplist 'cons '(1 2 3) '(1 2 3))
(((1 2 3) 1 2 3) ((2 3) 2 3) ((3) 3))
>(My_maplist (lambda (x) (eval (cons 'gcd x))) '(10 6 42 66 18))
(2 6 6 6 18)
```

В последнем примере формируется список из наибольших общих делителей последовательных хвостов списка (10 6 42 66 18).

Аналогично funcall и apply в Лиспе есть встроенные отображающие функционалы mapcar, maplist, которые выполняют те же действия, что и рассмотренные нами. С помощью данных функционалов можно сократить повторяющиеся вычисления, поэтому они часто используются для программирования циклов специального вида. Например, в следующей реализации из строки «It's work» берутся четыре случайных символа и формируется список из результатов:

```
>(mapcar (lambda (n) (char "It's work!"n)) (mapcar 'random '(9 9 9 9)))
(\#\r   \#\o  \#\'   \#\r )
```

Отображающие функционалы mapcan, mapcon

Следующие встроенные отображающие функционалы — mapcan, mapcon являются аналогами функций mapcar и maplist соответственно, но в отличие от последних не строят новый список из результатов, а объединяют списки, являющиеся результатами в один список:

```
>(mapcan 'list '(1 2 3 4))
(1 2 3 4)
>(mapcon 'reverse '(1 2 3 4 5 6))
(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6)
```

Замыкание

Иногда бывает важно отметить функциональный аргумент, отличив его от обычного уже на этапе вызова функционала. Такая возможность есть в Лиспе — функциональный аргумент можно пометить с помощью специальной формы Function, предотвращающей его вычисление. Аналогично тому, как у обычной блокировки quote есть синтаксический сахар ’ так и у формы function, которую называют функциональная блокировка, есть сокращение в виде #’. В обычном случае если функционально блокируется какая либо форма, не содержащая свободных переменных, то такая блокировка ничем не отличается от обычной блокировки Quote. Например, формы

```
>(funcall '+ 1 2)
3
>(funcall #'+ 1 2)
3
```

дают одинаковый результат. Однако назначение функциональной блокировки совсем в другом — в создании объектов с внутренним состоянием.

Замыкание (продолжение)

Например, рассмотрим реализацию банковского счета. Клиент должен иметь возможность положить на счет деньги и снять со счета деньги. Может быть несколько разных клиентов, каждый со своим счетом. Очевидно, для реализации банковского счета нам необходимо создать объект с внутренним состоянием, который помнит, сколько денег на счету. Конечно, для каждого счета можно определить свою глобальную переменную, но тогда любое неосторожное действие с глобальными переменными может изменить счет, это неудобно. В этом случае как раз помогает так называемое *замыкание или лексическое замыкание*. Чтобы лучше понять, как оно работает используем функциональную блокировку для выражения со свободной переменной:

```
>(setq add (let ((y 10)) #'(lambda (x) (+ x y))))
#<FUNCTION :LAMBDA (X) (+ X Y)>
```

Переменной add присваивается лямбда выражение, в котором X — связанная переменная, Y — свободная переменная, которая инициализируется начальным значением 10.

Замыкание (продолжение)

Проведем расчеты:

```
>(funcall add 7)
17
```

Теперь изменим значение переменной y:

```
>(setq y 1)
1
```

Повторим вычисление add:

```
>(funcall add 7)
17
```

Результат не изменился, переменная y стала внутренней переменной и сохраняет свое первоначальное значение 10, в тоже время глобальной переменной y присвоено 1. Таким образом, мы научились сохранять контекст вычислений и создавать объекты с внутренним состоянием. Это сохранение контекста и называется замыканием. Оно не позволяет изменять свободные переменные замыкания внешним по отношению к данному замыканию функциям.

Замыкание (банковкий счет)

Реализуем для примера объект — банковский счет:

```
(defun make_count (N)
  #'(lambda (x)
    (if (>= N x) (setq N (- N x)) "No money")))
```

Данная функция фактически делает замыкание на свободной по отношению к лямбда выражению переменной N, сохраняя для N значение на момент вызова. Создадим счет, на который положим 100 y.e.

```
>(setq a1 (make_count 100))
#<FUNCTION :LAMBDA (X) (IF (>= N X) (SETQ N (- N X)) "No money")>
```

Теперь можно снимать деньги:

```
>(funcall a1 9)
91
>(funcall a1 100)
"No money"
```

Замыкание (банковкий счет)

Можно создать еще один счет, который будет совершенно независимым объектом от первого:

```
(setq a2 (make_count 100))
#<FUNCTION :LAMBDA (X) (IF (>= N X) (SETQ N (- N X)) "No money")>
>(funcall a2 10)
90
>(funcall a1 2)
89
```

Замыкание можно трактовать как функцию, вычисление которой осуществлено лишь частично, окончательное вычисление функции отложено на момент ее вызова. При создании замыкания из определения функции формируется частично вычисленный контекстный объект и, чтобы осуществить окончательное вычисление, ему следует передать значения недостающих параметров.

Генераторы

Замыкания хорошо подходят для программирования так называемых генераторов — при каждом вызове порождается значение в некотором смысле следующее по порядку. Например, построим генератор, который при каждом вызове выдает квадрат следующего по отношению к предыдущему вызову число:

```
(defun next (n)
  #'(lambda () (prog1 (* n n) (setq n (+ n 1)))))
```

Теперь создаем два генератора

```
>(setq u1 (next 1))
#<FUNCTION :LAMBDA NIL (PROG1 (* N N) (SETQ N (+ N 1)))>
>(setq u2 (next 2))
#<FUNCTION :LAMBDA NIL (PROG1 (* N N) (SETQ N (+ N 1)))>
```

Генераторы - пробуем

```
>(funcall u1)
1
>(funcall u1)
4
>(funcall u1)
9
>(funcall u1)
16
>(funcall u2)
4
```

С помощью генераторов потенциально можно работать с неограниченными последовательностями и структурами. Такие структуры и последовательности вычисляются лишь до необходимой в текущий момент глубины.

Мемоизация

Сама по себе идея мемоизации очень проста - сохранение результатов вычисления функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ. Перед вызовом функции проверяется, вызывалась ли функция ранее:

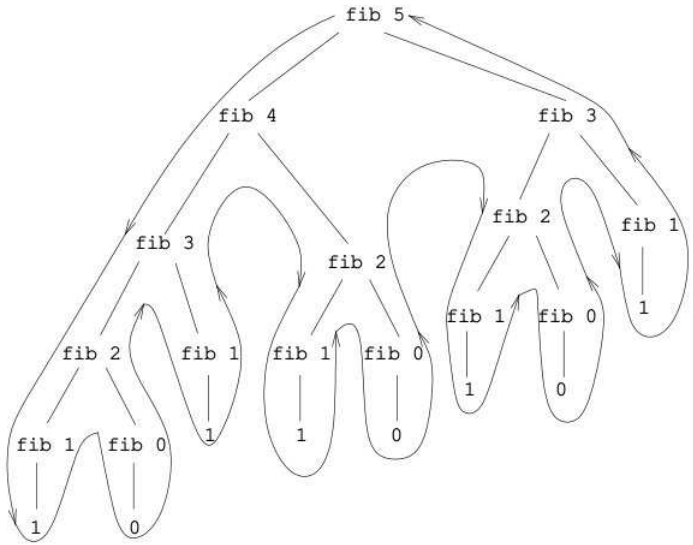
- если не вызывалась, функция вызывается и результат её выполнения сохраняется;
- если вызывалась, используется сохранённый результат.

Рассмотрим вначале на примере чисел Фибоначчи. Как мы показывали ранее использование вот такой простой реализации:

```
(defun fib (n)
  (cond
    ((<= n 1) n)
    (t (+ (fib (- n 1)) (fib (- n 2))))))
```

Приводит к древовидному рекурсивному процессу с вычислительной сложностью $O(2^n)$.

Демонстрация древовидного рекурсивного процесса при вычислении чисел Фибоначчи



Как видно из рисунка, вычисления на каждом шаге раздваиваются и требуют вычисления чисел Фибоначчи от чисел, для которых они уже были подсчитаны на предыдущих узлах дерева. В итоге возникает многократный повторный счет и рекурсивный вызов. Для небольших n с этим еще можно мириться, но увеличение n быстро выявляет неэффективность.

Демонстрация неэффективности вычисления чисел Фибоначчи в простейшей реализации

```
>(time (fib 10))
0.000 seconds of real time
0.000000 seconds of total run time (0.000000 user, 0.000000 system)
100.00% CPU
17,962 processor cycles
0 bytes consed
55
>(time (fib 40))
Evaluation took:
4.127 seconds of real time
4.124000 seconds of total run time (4.124000 user, 0.000000 system)
99.93% CPU
9,887,697,905 processor cycles
27,504 bytes consed
102334155
```

Вспомогательные средства - хэш таблицы

Бороться с возникшими сложностями можно по-разному. Один способ - использование накапливающего параметра уже был рассмотрен. Второй способ - использовать мемоизацию. Но прежде, чем мы ее реализуем, рассмотрим два дополнительных механизма.

Хэш-таблицы - как и ассоциированный список, это способ связать ключ со значением. В Common Lisp создать хэш-таблицу можно так:

```
>(setf my-table (make-hash-table))
#<HASH-TABLE :TEST EQL :COUNT 0 1007D11343>
```

Чтобы получить значение ключа из хэш-таблицы, можно использовать:

```
>(gethash 'key my-table)
NIL
NIL
```

Функция gethash возвращает два результата: по-первых, это значение ключа key (в нашем случае, поскольку мы ничего в таблицу не записали, то значение любого ключа пусто), во-вторых, это признак, что у ключа есть значение (если значение есть, то возвращается T, в противном случае NIL)

Хэш таблицы (продолжение)

Чтобы записать ключ-значение в хэш-таблицу, следует действовать так:
>(setf (gethash 'key my-table) 'value)
Теперь посмотрим, что записалось:
>(gethash 'key my-table)
VALUE
T
Таким образом, хэш-таблицы удобно использовать в нашей мемоизации. Чтобы получить сразу два значения от (gethash 'key my-table), можно использовать специальную форму:
>(multiple-value-bind (value foundp) (gethash 'key my-table))
В итоге переменной value присвоится значение ключа key, а переменной foundp признак того, что это значение действительно найдено в хэш-таблице.

Работа с символами функций, или программируемый язык программирования

В Common Lisp не закрыта возможность доступа к внутренней реализации всех функций (даже встроенных). Например, можно получить имя глобальной функции и присвоить ей что-то другое. Это возможно с помощью формы symbol-function. Например, присвоим функции sin функцию cos:
>(sin (/ pi 2))
1.0d0
>(cos (/pi 2))
6.123233995736766d-17 (ну ноль короче ;-))
>(setf (symbol-function 'sin) (lambda (x) (cos x)))
Немного ругаемся:
[Condition of type SYMBOL-PACKAGE-LOCKED-ERROR]
Выбираем 2: [UNLOCK-PACKAGE] Unlock the package.
>(sin (/ pi 2))
6.123233995736766d-17
Навредили (отменили встроенную функцию синуса), но в дальнейшем обязуемся направлять эту магию только на добрые дела!))

Реализация мемоизации (или ох уж этот русский язык)

```
(defun memoize  (my-function)
  (let ((my-table (make-hash-table)))
    #'(lambda (my-key)
      (multiple-value-bind (value foundp)
          (gethash my-key my-table)
  (if foundp
    value
    (setf (gethash my-key my-table)
              (funcall my-function my-key)))))))

(defun make-memoize (function-name)
  (setf (symbol-function function-name)
(memoize (symbol-function function-name))))
```

Реализация мемоизации (продолжение)

Итак, функция memoize формирует замыкание - недовычесленный контекст, требующий на вход значения my-key. Далее проверяется, если от my-key функция уже вычислялась, то результат ищется в хэш-таблице, иначе функция вычисляется от этого значения и результат записывается в хэш-таблицу.
Функция make-memoize делает тот самый трюк - заменяет функцию на ее мемоизированный вариант. Если этого не сделать, то повторной рекурсии не избежать:

```
>(setf fib-test (memoize 'fib))
#<CLOSURE (LAMBDA (MY-KEY) :IN MEMOIZE) 10069088AB>
> (time (funcall fib-test 40))
Evaluation took:
4.375 seconds of real time
4.372000 seconds of total run time (4.372000 user, 0.000000 system)
99.93% CPU
10,480,494,718 processor cycles
33,936 bytes consed
102334155
```

Не помогло!))

Реализация мемоизации (продолжение)

Мемоизируем функцию fib, так, чтобы и при рекурсивных вызовах вызывался ее мемоизированный вариант:

```
>(make-memoize 'fib)
#<CLOSURE (LAMBDA (MY-KEY) :IN MEMOIZE) 1006EE23BB>
```

Вычисляем:

```
>(time (funcall fib-test 40))
Evaluation took: 0.000 seconds of real time
0.000000 seconds of total run time (0.000000 user, 0.000000 system)
100.00% CPU
5,830 processor cycles
0 bytes consed
102334155
```

Ух ты!

Мемоизация - еще один пример

```
(defun HV_To_Number (L)
  (cond
     ((and (Null (cdr L)) (eq (car L) 'H)) 2)
     ((and (Null (cdr L)) (eq (car L) 'V)) (/ 1 2))
     ((eq (car L) 'H) (+ 1 (HV_To_Number(cdr L))))
     ((eq (car L) 'V) (/ (HV_To_Number (cdr L))
                         (+ 1 (HV_To_Number(cdr L)))))))
```

Мемоизация - еще один пример

```
> (time (HV_To_Number '(V V V H V V H H H V V V V H V V H H H V V V V H
V V H H H V V V V H V V H H H V ))) Evaluation took:
2.159 seconds of real time
2.168000 seconds of total run time (2.156000 user, 0.012000 system)
[ Run times consist of 0.200 seconds GC time, and 1.968 seconds non-GC time. ]
100.42% CPU
5,171,272,384 processor cycles
1,423,113,808 bytes consed
2870201/10614001
Мемоизируем
> (make-memoize 'HV_To_Number)
#<CLOSURE (LAMBDA (MY-KEY) :IN MEMOIZE) 1007468E9B>
> (time (HV_To_Number '(V V V H V V H H H V V V V H V V H H H V V V V H
V V H H H V V V V H V V H H H V )))
Evaluation took:
0.000 seconds of real time
0.000000 seconds of total run time (0.000000 user, 0.000000 system)
100.00% CPU
113,706 processor cycles
0 bytes consed
2870201/10614001
```

Задание

Мемоизируйте задачу о размене монет. Всегда ли мемоизация эффективна?

Символьное дифференцирование

```
;;;Присваивание свойству proizvod символов значений
;;;свойству proizvod символа + присваивается значение diff+ и т.д.
(setf (get '+ 'proizvod) 'diff+)
(setf (get '* 'proizvod) 'diff*)
(setf (get '/ 'proizvod) 'diff/)
(setf (get '^ 'proizvod) 'diff^)
(setf (get '- 'proizvod) 'diff-)
(setf (get 'sin 'proizvod) 'diffsin)
(setf (get 'cos 'proizvod) 'diffcos)
(setf (get 'tan 'proizvod) 'difftan)
(setf (get 'log 'proizvod) 'difflog)
```

Символьное дифференцирование

```
;;;Общая функция дифференцирования
(defun diff (L x)
(cond
((atom L) (if (eq L x) 1 0))
(t (funcall (get (first L) 'proizvod) (cdr L) x))))
;;;Дифференцирование суммы
(defun diff+ (L x)
(List '+ (diff (first L) x)
(diff (second L) x)))
;;;Дифференцирование разности
(defun diff- (L x)
(List '- (diff (first L) x)
 (diff (second L) x)))
```

Символьное дифференцирование (продолжение)

```
;;;Дифференцирование произведения
(defun diff* (L x)
(List '+ (List '* (diff (first L) x) (second L))
(List '* (diff (second L) X) (first L))))
;;;Дифференцирование частного
(defun diff/ (L x)
(list '/
(List '- (List '* (diff (first L) x) (second L))
(List '* (diff (second L) X) (first L)))
(list '* (second L) (second L))))
```

Символьное дифференцирование - задание 1

Реализовать дифференцированиие логарифма (функция difflog)

Символьное дифференцирование - задание 2

Реализовать дифференцированиие тригонометрических функций (функции diffsin, diffcos, difftan)

Символьное дифференцирование - задание 3

Реализовать дифференцированиие степени.

```
(defun ^ (X N) (expt X N))
```

Символьное дифференцирование - задание 4

Реализовать упрощатель:

```
(defun simple (L)
(cond
((atom L) L)
((and (eq (first L) '+) (numberp (second L)) (numberp (third L)))
                (+ (second L) (third L)))
((and (eq (first L) '+) (eq (second L) 0) ) (third L))
((and (eq (first L) '+) (eq (third L) 0)) (second L))
((and (eq (first L) ' ) (numberp (second L)) (numberp (third L)))
              (  (second L) (third L)))
((and (eq (first L) ' ) (eq (second L) 0) ) (list '  1 (third L)))
((and (eq (first L) ' ) (eq (third L) 0)) (second L))
```

и т.д.

Символьное дифференцирование - задание 5

Реализовать дифференцированиие с упрощением выражения.

Под капотом системы Maxima

Рассмотрим в будущем!