Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное учреждение высшего образования

«Новгородский государственный университет имени Ярослава Мудрого»

Институт «Электронных и информационных систем»

Кафедра «Информационных технология и систем»

Лабораторная работа №4

**«ЛОКАЛЬНЫЕ ОПРЕДЕЛЕНИЯ»**

по дисциплине:

«Функциональное и логическое программирование»

**Отчёт**

Принял преподаватель:

\_\_\_\_\_\_\_\_\_\_ / Михайлов Д.В

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Выполнил студент группы 8091:

\_\_\_\_\_\_\_\_\_\_ / Лехновский А. Д.

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

**Великий Новгород**

**2021**

1. **Цель и задачи данной лабораторной работы**

Целью работы является практическое изучение различных видов локальных определений и особенностей их использования в рекурсивных программах.

Основные задачи:

* Изучить применение техники нисходящей и восходящей рекурсии при написании рекурсивных функций с использованием локальных определений;
* Сравнить возможности локальных определений LET и LAMBDA по организации вычислений в рекурсивных программах;

1. **Задания на лабораторную работу**

**Задача №1:**

Описать функцию вычисления факториала. Рассмотреть варианты решения задачи с применением локальных определений LAMBDA и LET.

**Задача №2:**

С использованием локальных определений LAMBDA или LET разработать программу символьного дифференцирования в соответствии с правилами, изложенными в книге Клоксин У. “Программирование на языке Пролог”.

**Задача №3:**

С применением локальных определений LAMBDA или LET реализовать функцию, возвращающую Т в том случае, если одинаковые атомы расположены в исходных списках в одном и том же порядке.

**Задача №4:**

Реализовать простейший интерпретатор лисповских программ. На вход интерпретатора подается текст, который может быть интерпретирован как вызов или суперпозиция функций.

Требования к программе:

* интерпретация базовых функций Лиспа и арифметических операций +, -, /, \*;
* в программе должны использоваться локальные определения;
* не допускается использование встроенной функции-интерпретатора EVAL;

**Задача №5:**

Дополнить интерпретатор из Задачи №4 функцией вычитания множеств.

1. **Решение поставленных задач**

**Задача №1:**

Факториал натурального числа n определяется как произведение всех натуральных чисел от 1 до n включительно:

Идея в реализованных рекурсивных функциях состоит в том, что перемножаются числа от заданного числа n до 1. Первым множителем будет число n, затем мы уменьшаем его на 1 и рекурсивно передаём наш новый множитель в функцию вычисления факториала, пока n не станет равно 1.

Были реализованы две функции с использованием локальных определений. Одна с использованием LET, а другая – LAMBDA. Основное отличие между ними только в том, что что формальные и фактические параметры LET-вызова помещены совместно в начале формы.

Код программы для решения задачи приведен в *Приложении 1*.

|  |  |
| --- | --- |
| *Входные данные* | *Результат, полученный с помощью программы* |
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |

*Таблица №1. Тестовые наборы данных.*

Частный случай 0! = 1;

**Задача №2:**

Символьным дифференцирование в математике называется операция преобразования одного арифметического выражения в другое, которое называется производной. Пусть U и V обозначает арифметическое выражение, которое может содержать переменную x, c – константа.

Правила преобразования при символьном дифференцировании представлены в Таблице №2:

|  |  |
| --- | --- |
| *Входные данные* | *Результат* |
| c | 0 |
| x | 1 |
| U + V | (U’ + V’) |
| U - V | (U’ - V’) |
| cU | (\* c U’) |
| U \* V | (+ (\* U’ V) (\* U V’)) |
| U / V | (/ (- (\* U’ V) (\* V’ U)) (^ V 2)) |
|  | (\* c (^ U (- c 1))) |
| lnU | (\* (^ U -1) U’) |

*Таблица №2. Правила преобразования.*

Код программы для решения задачи приведен в *Приложении 2*.

|  |  |
| --- | --- |
| *Входные данные* | *Результат, полученный с помощью программы* |
| '(15) | 0 |
| '(x) | 1 |
| '(y) | 0 |
| '(^ x 8) | (\* 8 (^ x 7)) |
| '(+ (^ x 5) (x)) | (+ (\* 5 (^ x 4)) 1) |
| '(- (\* 4 x) (^ x 4)) | (- 4 (\* 4 (^ x 3))) |
| '(\* 8 x) | 8 |
| '(\* 5 (^ x 3)) | (\* 5 (\* 3 (^ x 2))) |
| '(\* (\* 5 x) (^ x 3)) | (+ (\* 5 (^ x 3)) (\* (\* 3 (^ x 2)) (\* 5 x))) |
| '(\* (x) (^ x 2)) | (+ (\* 1 (^ x 2)) (\* (\* 2 (^ x 1)) (x))) |
| '(/ (1) (x)) | (/ (- (\* 0 (x)) (\* 1 (1))) (^ (x) 2)) |
| '(/ (\* 7 x) (^ x 7)) | (/ (- (\* 7 (^ x 7)) (\* (\* 7 (^ x 6)) (\* 7 x))) (^ (^ x 7) 2)) |
| '(ln (x)) | (\* (^ (x) -1) 1) |
| '(ln (^ x 2)) | (\* (^ (^ x 2) -1) (\* 2 (^ x 1))) |
| '(ln (\* 3 x)) | (\* (^ (\* 3 x) -1) 3) |

*Таблица №3. Тестовые наборы данных.*

**Задача №3:**

Функция del-all имеет следующую логику:

Мы проверяем голову первого списка на атомарность (является ли он атомом?) и содержание (содержится ли во втором списке такой же атом?). Если оба условия соблюдены мы образуем список из головы первого списка и рекурсивно вызываем del-all, в которую передаём в качестве аргументов второй список без изменений и хвост первого списка. В противном случае мы рекурсивно вызываем del-all, в которую передадим хвост первого списка и второй список. Конец рекурсии будет, когда мы дойдём до конца списка, то есть список станет пустым. На выходе функция del-all даёт список, состоящий из атомов первого списка, которые также были во втором списке. Причём последовательность элементов этого списка строится на основе последовательности этих атомов в первом списке.

В главной функции мы для удобства использовали локальные определения, где локальным переменным s1 и s2 были присвоены результаты функции del-all при разных аргументах, то есть список. Затем мы просто проверяем на эквивалентность переменные s1 и s2.

Благодаря использованию локальных определений LET и LAMBDA сильно улучшилась читаемость кода данной программы.

Код программы для решения задачи приведен в *Приложении 3*.

|  |  |
| --- | --- |
| *Входные данные* | *Результат, полученный с помощью программы* |
| '(a b x c) '(h a x d) | true |
| '(a b x c) '(h x d a) | nil |
| '(1 (2 3 4) a (5 6 7) 2 () b) '((q w e r t y) 1 a (2 3 4) 2 b) | true |
| '(b (2 3 4) a (5 6 7) 2 () 1) '((q w e r t y) 1 a (2 3 4) 2 b) | nil |
| '(1 2 3 4) '(6 2 5 4) | true |
| '(1 2 3 4) '(6 2 5 4 8 9) | true |
| '(1 2 3 9 4) '(6 2 5 4 8 9) | nil |
| '() '() | true |
| '(1 a) '(a 1 a) | nil |
| '(1 b c d 2 5 6 7 9 0) '() | true |
| '((a b c) d) '( d c a b ) | true |

*Таблица №4. Тестовые наборы данных.*

**Задача №4-5:**

В созданном простейшем интерпретаторе были реализованы такие базовые функции Лиспа как:

* CONS (функция my\_cons);
* REST (функция my\_rest);
* FIRST (функция my\_first);

Также были реализованы такие арифметические операции как:

* + (функция plus);
* - (функция minus);
* \*(функция multiplication);
* / (функция division);

Функция lisp\_interpritator запускает работу нашего простенького интерпретатора. После запуска интерпретатора вызывается функция isFunc(в которой описаны доступные функции), если данная функция была реализована, то выполняется соответствующая ей функция, иначе мы выводим весь список. Также в качестве индивидуального задания была реализована функция my\_setsubtraction (функция вычитания множеств).

Код программы для решения задачи приведен в *Приложении 4*.

|  |  |
| --- | --- |
| *Входные данные* | *Результат, полученный с помощью программы* |
| '(+ (+ 23 7) (+ 10 56)) | 96 |
| '(- (+ 10 56) (- 25 13)) | 54 |
| '(\* (\* 2 5) (\* 8 9)) | 720 |
| '(/ (/ 100 4) (/ 10 2)) | 5 |
| '(- (+ (/ 50 5)(\* 3 2))(\* 7 2)) | 2 |
| '(first (A (B C) 2 1)) | A |
| '(rest (A (B C) 2 1)) | ((B C) 2 1) |
| '(first (rest (A B C))) | B |
| '(+ (- (\* (/ 10 2) 3) 9) 100) | 106 |
| '(first (rest (10 (+ 33 1) (- 5 1) 5 (+ 10 1))) 55) | 34 |
| '(cons a b (+ 1 2) (\* 2 7) (/ 64 8)) | (a b 3 14 8) |
| '(setsubtraction (first ((22 1 2 4 7 8 11) (8 9 19))) (1 2 4 11 22)) | (7 8) |
| '(setsubtraction ( 1 2 3 4 5 ) (4 5 6 7)) | (1 2 3) |

*Таблица №5. Тестовые наборы данных.*

1. **Вывод**

В данной лабораторной работе были практически изучены различные виды локальных определений и особенности их использования в рекурсивных программах.

**Приложение 1**

Код программы:

;Описать функцию вычисления факториала. Рассмотреть варианты решения задачи с

; применением локальных определений LAMBDA и LET.

(define (fact\_let n)

(let (x (- n 1) )

(if (or (= n 1) (= n 0))

1

(\* n (fact\_let x))

)

)

)

(define (fact\_lambda n)

(

(lambda (x)

(if (or (= n 1) (= n 0))

1

(\* n (fact\_lambda x))

)

) (- n 1)

)

)

(fact\_let 5)

(fact\_lambda 5)

**Приложение 2**

Код программы:

;Разработать программу символьного дифференцирования с применением локальных определений LAMBDA и LET.

(define (differential lst)

(let (component (first lst))

(cond

(

(= component 'x) 1

)

(

(= component '^)

(list '\* (nth 2 lst) (list '^ (nth 1 lst) (- (nth 2 lst) 1)))

)

(

(= component '+)

(list '+ (differential (nth 1 lst)) (differential (nth 2 lst)))

)

(

(= component '-)

(list '- (differential (nth 1 lst)) (differential (nth 2 lst)))

)

(

(= component '\*)

(if (and (number? (nth 1 lst)) (= (nth 2 lst) 'x))

(nth 1 lst)

(if (number? (nth 1 lst))

(list '\* (nth 1 lst) (differential (nth 2 lst)))

(list '+ (list '\* (differential (nth 1 lst))(nth 2 lst)) (list '\* (differential (nth 2 lst)) (nth 1 lst)))))

)

(

(= component '/)

(list '/ (list '- (list '\* (differential (nth 1 lst)) (nth 2 lst)) (list '\*(differential (nth 2 lst)) (nth 1 lst))) (list '^ (nth 2 lst) 2))

)

(

(= component 'ln)

(list '\* (list '^ (nth 1 lst) '-1) (differential (nth 1 lst)))

)

(

(or (number? component) (and (symbol? component) (not (= component 'x)))) 0

)

)

)

)

(differential '(15))

(differential '(x))

(differential '(y))

(differential '(+ (x) (y)))

(differential '(^ x 8))

(differential '(+ (^ x 5) (x)))

(differential '(- (\* 4 x) (^ x 4)))

(differential '(\* 8 x))

(differential '(\* 5 (^ x 3)))

(differential '(\* (\* 5 x) (^ x 3)))

(differential '(\* (x) (^ x 2)))

(differential '(/ (1) (x)))

(differential '(/ (\* 7 x) (^ x 7)))

(differential '(ln (x)))

(differential '(ln (^ x 2)))

(differential '(ln (\* 3 x)))

**Приложение 3**

Код программы:

;Реализовать функцию, возвращающую Т в том случае, если одинаковые атомы расположены в исходных списках в одном и том же порядке.

; Реализация через let

(define (del-all lsta lst)

(cond

((null? lst) '())

( (and (atom? (first lst)) (member (first lst) lsta)) (cons (first lst) (del-all lsta (rest lst))))

(true (del-all lsta (rest lst)))

)

)

(define (task3\_let lst1 lst2)

(let

((s1 (del-all lst2 lst1))

(s2 (del-all lst1 lst2)))

(= s1 s2)

)

)

(task3\_let '(a b x c) '(h a x d))

(task3\_let '(a b x c) '(h x d a))

(task3\_let '(1 (2 3 4) a (5 6 7) 2 () b) '((q w e r t y) 1 a (2 3 4) 2 b))

(task3\_let '(b (2 3 4) a (5 6 7) 2 () 1) '((q w e r t y) 1 a (2 3 4) 2 b))

(task3\_let '(1 2 3 4) '(6 2 5 4))

(task3\_let '(1 2 3 4) '(6 2 5 4 8 9))

(task3\_let '(1 2 3 9 4) '(6 2 5 4 8 9))

(task3\_let '() '())

(task3\_let '(1 a) '(a 1 a))

(task3\_let '(1 b c d 2 5 6 7 9 0) '())

(task3\_let '((a b c) d) '( d c a b ))

;Реализация через lambda

(define (del-all lsta lst)

(cond

((null? lst) '())

( (and (atom? (first lst)) (member (first lst) lsta)) (cons (first lst) (del-all lsta (rest lst))))

(true (del-all lsta (rest lst)))

)

)

(define (task3\_lambda lst1 lst2)

(

(lambda (s1 s2)

(= s1 s2)

) (del-all lst2 lst1) (del-all lst1 lst2)

)

)

(task3\_lambda '(a b x c) '(h a x d))

(task3\_lambda '(a b x c) '(h x d a))

(task3\_lambda '(1 (2 3 4) a (5 6 7) 2 () b) '((q w e r t y) 1 a (2 3 4) 2 b))

(task3\_lambda '(b (2 3 4) a (5 6 7) 2 () 1) '((q w e r t y) 1 a (2 3 4) 2 b))

(task3\_lambda '(1 2 3 4) '(6 2 5 4))

(task3\_lambda '(1 2 3 4) '(6 2 5 4 8 9))

(task3\_lambda '(1 2 3 9 4) '(6 2 5 4 8 9))

(task3\_lambda '() '())

(task3\_lambda '(1 a) '(a 1 a))

(task3\_lambda '(1 b c d 2 5 6 7 9 0) '())

(task3\_lambda '((a b c) d) '( d c a b ))

**Приложение 4**

Код программы:

(define (lisp\_interpritator lst)

(if (atom? (nth 0 lst))

(isFunc lst 0)

"MISTAKE first func"

)

)

(define (isFunc lst i)

(let (component (nth i lst))

(case component

(+ (plus lst i))

(- (minus lst i))

(\* (multiplication lst i))

(/ (division lst i))

(first (my\_first lst i))

(rest (my\_rest lst i))

(cons (my\_cons lst i))

(setsubtraction (my\_setsubtraction lst i))

(true lst)

)

)

)

(define (plus lst i)

(let (n (+ i 1))

(if (< n (length lst))

(if (number? (nth n lst))

(+ (nth n lst) (plus lst n))

(+ (isFunc (nth n lst) 0) (plus lst n))

)

0

)

)

)

(define (minus lst i)

(let (n (+ i 1))

(if (< n (length lst))

(if (number? (nth n lst))

(- (nth n lst) (minus lst n))

(- (isFunc (nth n lst) 0) (minus lst n))

)

0

)

)

)

(define (multiplication lst i)

(let (n (+ i 1))

(if (< n (length lst))

(if (number? (nth n lst))

(\* (nth n lst) (multiplication lst n))

(\* (isFunc (nth n lst) 0) (multiplication lst n))

)

1

)

)

)

(define (division lst i)

(let (n (+ i 1))

(if (< n (length lst))

(if (number? (nth n lst))

(/ (nth n lst) (division lst n))

(/ (isFunc (nth n lst) 0) (division lst n))

)

1

)

)

)

(define (my\_first lst i)

(let (n (+ i 1))

(if (list? (first(nth n lst)) )

(isFunc (first (nth n lst)) 0)

(first (isFunc (nth n lst) 0))

)

)

)

(define (check\_lst lst i)

(if (< i (length lst))

(if (list? (nth i lst))

(cons (isFunc (nth i lst) 0)(check\_lst lst (+ i 1)))

(cons (nth i lst) (check\_lst lst (+ i 1)))

)

)

)

(define (my\_rest lst i)

(let (n (+ i 1) )

(chop (check\_lst (rest (nth n lst)) 0))

)

)

(define (my\_cons lst i)

(let (n (+ i 1) )

(chop(check\_lst lst n))

)

)

(define (my\_setsubtraction lst i)

(let (

set1 (isFunc (nth 1 lst) 0)

set2 (isFunc (nth 2 lst) 0)

)

(if (and (isset? set1) (isset? set2))

(difference set1 set2)

"Mistake. These are not sets!"

)

)

)

(define (isset? lst)

(if (= (unique lst) lst)

true

nil

)

)

(lisp\_interpritator '(+ (+ 23 7) (+ 10 56)))

(lisp\_interpritator '(- (+ 10 56) (- 25 13)))

(lisp\_interpritator '(\* (\* 2 5) (\* 8 9)))

(lisp\_interpritator '(/ (/ 100 4) (/ 10 2)))

(lisp\_interpritator '(- (+ (/ 50 5)(\* 3 2))(\* 7 2)))

(lisp\_interpritator '(first (A (B C) 2 1)))

(lisp\_interpritator '(rest (A (B C) 2 1)))

(lisp\_interpritator '(first (rest (A B C))))

(lisp\_interpritator '(+ (- (\* (/ 10 2) 3) 9) 100))

(lisp\_interpritator '(first (rest (10 (+ 33 1) (- 5 1) 5 (+ 10 1))) 55))

(lisp\_interpritator '(cons a b (+ 1 2) (\* 2 7) (/ 64 8)))

(lisp\_interpritator '(setsubtraction (first ((22 1 2 4 7 8 11) (8 9 19))) (1 2 4 11 22)));

(lisp\_interpritator '(setsubtraction ( 1 2 3 4 5 ) (4 5 6 7)));