Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт по лабораторной работе №4 по дисциплине «Функциональное и логическое программирование»

Тема: Использование управляющих структур, работа со списками
Студент: Княжев А. В.
Группа: ИУ7-62Б
Оценка (баллы): _____
Преподаватели: Толпинская Н. Б., Строганов Ю. В.

Москва — 2023 г.

Оглавление

1. Теоретическая часть

1.1. Синтаксическая форма и хранение программы в памяти

В Lisp программа синтаксически представлена в форме S-выражений. Реализована единая форма фиксации, то есть отсутствие разделения на программу и данные. И программа, и данные представляются списочной структурой. Благодаря такому подходу возможно изменение кода программы при обработке данных. Программа будто может "изменять саму себя". Так как программа имеет вид S-выражения, в памяти она представлена либо как атом (5 указателей, которыми представляется атом в памяти), либо как списковая ячейка (2 указателя, бинарный узел).

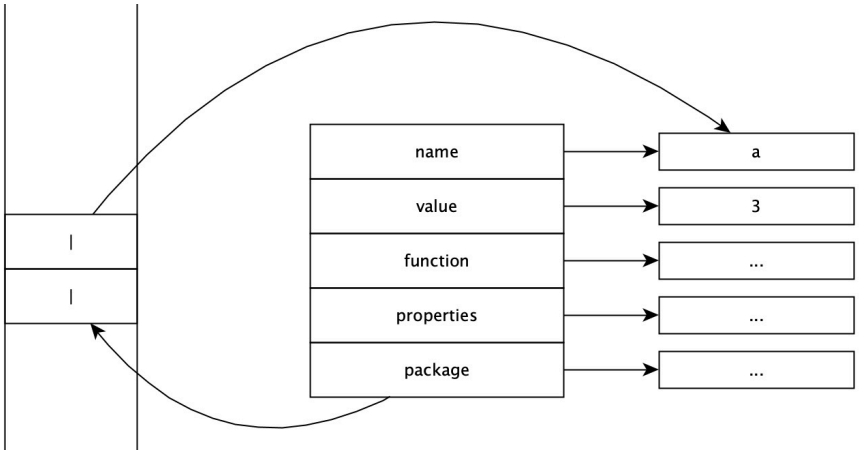

Рисунок 1.1 — Атом в памяти

Рисунок 1.2 — Списковая ячейка

1.2. Трактовка элементов списка

При обработке списков первый элемент воспринимается интерпретатором как название функции, все остальные — ее аргументы (список трактуется как вычислимая форма). Количество элементов, не считая первого — названия функции, должно совпадать с количеством входных аргументов указанной функции.

В случае если перед скобкой применяется блокировка вычислений (‘ или `quota`), результатом является все, что стоит после блокировки.

```
> (cons 1 2)
(1 . 2)
```

```
> (cons 1 2 3)
EVAL: too many arguments given to CONS: (CONS 1 2 3)
```

```
> `(cons 1 2)
(CONS 1 2)
```

1.3. Порядок реализации программы

1. Ожидает ввода S-выражения.
2. Передает введенное S-выражение функции `eval`.
3. Выводит полученный результат.

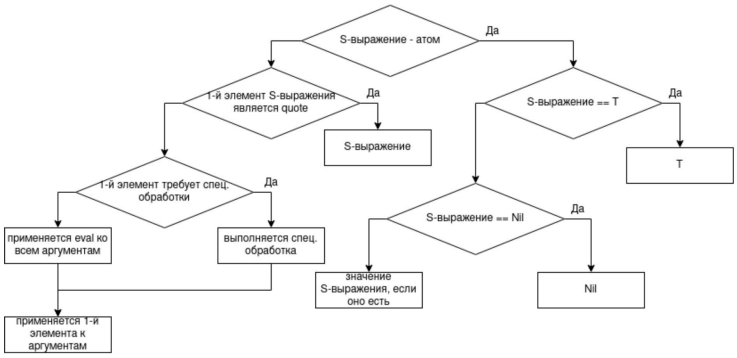

Рисунок 1.3 — Диаграмма работы `eval`

1.4. Способы определения функции

Функцию можно определить двумя способами: неименованную с помощью `lambda` и именованную с помощью `defun`.

```
(lambda (x_1 x_2 ... x_n) f)
```

— `f` — тело функции;

— `x_i` — формальные параметры.

```
(defun <имя> [lambda] (x_1 x_2 ... x_n) f)
```

— `f` — тело функции;

— `x_i` — формальные параметры.

Тогда имя будет ссылкой на описание функции.

1.5. Работа со списками

Ниже перечислены наиболее часто используемые для работы со списками функции.

1.5.1. Создание

Список можно создать несколькими способами.

1. Базисная функция `cons` может создавать список, если её второй аргумент является списком.

```
> (cons 1 `(2 3 4))
(1 2 3 4)
```

2. Функция `list` также создаёт список, принимая на вход неопределённое количество аргументов.

```
> (list 1 `(2 3 4) 5 6)
(1 (2 3 4) 5 6)
```

3. Функция `last` возвращает список, содержащий последний элемент в списке.

```
> (last `(1 2 3))
(3)
```

4. `append` принимает произвольное количество аргументов-списков и соединяет элементы верхнего уровня всех списков в один список. В результате должен быть построен новый список.

```
> (append (list 1 2) (list 3 4))
(1 2 3 4)
```

1.5.2. Изменение

1. Конкатенация — `nconc`. Похожа на `append`, но в отличие от неё «ломает» свои аргументы, не создавая копии списковых ячеек.

```
> (setf x (list 1 2))
(1 2)
> (nconc x (list 3 4))
(1 2 3 4)
> x
(1 2 3 4)
```

2. `reverse` — выполняет разворот списка по верхнему уровню списковых ячеек. Создает копию, не «ломая» аргумент.

```
> (setf x (list 1 2))
(1 2)
> (reverse x)
(4 3 2 1)
> x
(1 2 3 4)
```

3. `nreverse` — работает аналогично, но без создания копий.

```
> (setf x (list 1 2))
(1 2)
> (nreverse x)
(4 3 2 1)
> x
(4 3 2 1)
```

1.5.3. Селекторы

Функция `car` используется для получения `car`-указателя — указателя на голову списка. Функция `cdr` используется для получения `cdr`-указателя — указателя на хвост списка.

```
(car `(1 2 3 4))
1
```

```
> (cdr `(1 2 3 4))
(2 3 4)
```

Также, можно использовать композицию функций.

```
> (caddr `(1 2 3 4))
3
```

```
> (caadr `(1 (2 5) 3 4))
2
```

2. Практическая часть

2.1. Задание 1

Задание

Чем принципиально отличаются функции cons, list, append? Пусть

```
(setf lst1 `( a b c))
(setf lst2 `( d e))
```

Каковы результаты вычисления следующих выражений?

1. `(cons lst1 lst2)`
2. `(list lst1 lst2)`
3. `(append lst1 lst2)`

Решение

1. `((A B C) D E)`
2. `((A B C) (D E))`
3. `(A B C D E)`

2.2. Задание 2

Задание

Каковы результаты вычисления следующих выражений, и почему?

1. `(reverse ‘(a b c))`
2. `(reverse ‘(a b (c (d))))`
3. `(reverse ‘(a))`
4. `(last ‘(a b c))`

5. `(last ‘(a))`
6. `(last ‘((a b c)))`
7. `(reverse ())`
8. `(reverse ‘((a b c)))`
9. `(last ‘(a b (c)))`
10. `(last ())`

Решение

1. `(C B A)`
2. `((C (D)) B A)`
3. `(A)`
4. `(C)`
5. `(A)`
6. `((A B C))`
7. `NIL`
8. `((A B C))`
9. `((C))`
10. `NIL`

2.3. Задание 3

Задание

Написать, по крайней мере, два варианта функции, которая возвращает последний элемент своего списка-аргумента.

Решение

```
(defun f1 (x) (car (last x)))
```

```
(defun f2 (x) (car (reverse x)))
```

2.4. Задание 4

Задание

Написать, по крайней мере, два варианта функции, которая возвращает свой список аргумент без последнего элемента.

Решение

```
(defun f3 (x) (reverse (cdr (reverse x))))
```

```
(defun f4 (x)
        (if (<= (length x) 1)
                nil
                (cons (car x) (f4 (cdr x)))
        )
)
```

2.5. Задание 5

Задание

Напишите функцию `swap-first-last`, которая переставляет в списке-аргументе первый и последний элементы.

Решение

```
(defun f5 (x)
        (append (last x) (cdr (f4 x)) (list (first x)))
)
```

2.6. Задание 6

Задание

Написать простой вариант игры в кости, в котором бросаются две правильные кости. Если сумма выпавших очков равна 7 или 11 — выигрыш, если выпало (1,1) или (6,6) — игрок имеет право снова бросить кости, во всех остальных случаях ход переходит ко второму игроку, но запоминается сумма выпавших очков. Если второй игрок не выигрывает абсолютно, то выигрывает тот игрок, у которого больше очков. Результат игры и значения выпавших костей выводить на экран с помощью функции `print`.

Решение

```
(defun dice ()
        (+ (random 6) 1)
)

(defun is_retry (p)
        (or (equal p (cons 6 6)) (equal p (cons 1 1)))
)
```

```
(defun player_dice (n)
        (let ((pair (cons (dice) (dice))))
                (print (list n `played pair))
                (cond
                        ((is_retry pair) (player_dice n))
                        (t (+ (car pair) (cdr pair)))
                )
        )
)

(defun is_win (s)
        (or (= s 7) (= s 11))
)

(defun play ()
        (let ((s1 (player_dice 1)) (s2 (player_dice 2)))
                (cond
                        ((is_win s1) (print "1 absolute win"))
                        ((is_win s2) (print "2 absolute win"))
                        ((> s1 s2) (print "1 win"))
                        ((< s1 s2) (print "2 win"))
                        (t (print "nobody win"))
                )
                nil
        )
)
```

2.7. Задание 7

Задание

Написать функцию, которая по своему списку-аргументу `lst` определяет является ли он палиндромом (то есть равны ли `lst` и `(reverse lst)`).

Решение

```
(defun f6 (x) (equalp x (reverse x)))
```

2.8. Задание 8

Задание

Напишите свои необходимые функции, которые обрабатывают таблицу из 4-х точечных пар: (страна . столица), и возвращают по стране — столицу, а по столице — страну.

Решение

```
(defun capital (table c)
        (cdr (assoc c table :test #'equalp))
)
```

```
(defun country (table c)
        (car (rassoc c table :test #'equalp))
)
```

2.9. Задание 9

Задание

Напишите функцию, которая умножает на заданное число-аргумент первый числовой элемент списка из заданного 3-х элементного списка-аргумента, когда

1. все элементы списка — числа;
2. элементы списка — любые объекты.

Решение

```
(defun f7 (l n)
        (cond
                ((not (listp l)) nil)
                ((not (numberp n)) nil)
                ((not (= (length l) 3)) nil)
                (t (list (* (first l) n) (second l) (third l)))
        )
)
```

```
(defun f8 (l n)
        (cond
                ((not (listp l)) nil)
                ((not (numberp n)) nil)
                ((not (= (length l) 3)) nil)
                ((numberp (first l)) (list (* (first l) n) (second l)
                (third l)))
                ((numberp (second l)) (list (first l) (* (second l) n)
                (third l)))
                ((numberp (third l)) (list (first l) (second l)
                (* (third l) n)))
                (t nil)
        )
)
```