



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»

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

Тема Использование управляющих структур, работа со списками

Студент Зайцева А. А.

Группа ИУ7-62Б

Оценка (баллы) _____

Преподаватели Толпинская Н.Б., Строганов Ю. В.

Москва — 2022 г.

Теоретические вопросы

1. Синтаксическая форма и хранение программы в памяти

Программа на Lisp представляет собой вызов функции на верхнем уровне.

Синтаксически программа оформляется в виде S-выражения (обычно – списка – частного случая точечной пары). S-выражение, попавшее на вход системы, анализирует функция eval. S-выражение очень часто может быть структурированным. Наличие скобок является признаком структуры.

По определению:

- S-выражение ::= <атом> | <точечная пара>
- Атомы:
 - символы (идентификаторы) – синтаксически – набор литер (букв и цифр), начинающихся с буквы;
 - специальные символы – T, Nil (используются для обозначения логических констант);
 - самоопределимые атомы – натуральные числа, дробные числа, вещественные числа, строки – последовательность символов, заключенных в двойные апострофы (например, “abc”);
- Точечная пара ::= (<атом> . <атом>) | (<атом> . <точечная пара>) | (<точечная пара> . <атом>) | (<точечная пара> . <точечная пара>);
- Список ::= <пустой список> | <непустой список>, где
 - <пустой список> ::= () | Nil,
 - <непустой список> ::= (<первый элемент> . <хвост>),
 - <первый элемент> ::= <S-выражение>,
 - <хвост> ::= <список>.

Атомы представляются **в памяти** пятью указателями (name, value, function, property, package), а любая непустая структура – списковой ячейкой (бинарным узлом), хранящей два указателя: на голову (первый элемент) и хвост – все остальное.

Функцией называется правило, по которому каждому значению одного или нескольких аргументов ставится в соответствие конкретное значение результата.

Функционалом, или функцией высшего порядка называется функция, аргументом или результатом которой является другая функция.

Форма – функция, которая особым образом обрабатывает свои аргументы, т. е. требует специальной обработки.

2. Трактовка элементов списка.

3. Порядок реализации программы.

4. Способы определения функции

Определение функций пользователя в Lisp-е возможно двумя способами.

- Базисный способ определения функции - использование λ -выражения (λ -нотации). Так создаются функции без имени.

λ -выражение: (lambda λ -список форма), где λ -список – это формальные параметры функции (список аргументов), а форма – это тело функции.

Вызов такой функции осуществляется следующим способом: (λ -выражение последовательность_форм), где последовательность_форм – это список фактических параметров.

Вычисление функций без имени может быть выполнено с использованием функционала `apply`: (`apply` λ -выражение последовательность_форм), где последовательность_форм – это список фактических параметров, или с использованием функционала `funcall`: (`funcall` λ -выражение последовательность_форм), где последовательность_форм – это фактические параметры.

Функционал `apply` является обычной функцией с двумя вычисляемыми аргументами, обращение к ней имеет вид: (`apply` F L), где F – функциональный аргумент и L – список, рассматриваемый как список фактических параметров для F. Значение функционала – результат применения F к этим фактическим параметрам.

Функционал `funcall` – особая функция с вычисляемыми аргументами, обращение к ней: (`funcall` F e1 ... en), $n \geq 0$. Её действие аналогично `apply`, отличие состоит в том, что аргументы применяемой функции F задаются не списком, а по отдельности.

- Другой способ определения функции – использование макро-определения `defun`:

(`defun` имя_функции λ -выражение),

или в облегченной форме:

(`defun` имя_функции (x_1, x_2, \dots, x_k) (форма)), где (x_1, x_2, \dots, x_k) – это список аргументов.

В качестве имени функции выступает символьный атом. Вызов именованной функции осуществляется следующим образом: (имя_функции последовательность_форм), где последовательность_форм – это фактические параметры

λ -определение более эффективно, особенно при повторных вычислениях.

Параметры функции, переданные при вызове, будут связаны с переменными в списке параметров из объявления функции. Еще один способ связывания формальных параметров с фактическими – использование функции `let`:

(`let` ((x_1 p_1) (x_2 p_2) ... (x_k p_k)) e),

где x_i – формальные параметры, p_i – фактические параметры (могут быть формами), e – форма (что делать).

Из указаний к выполнению работы

cond

Общий вид условного выражения:

(`cond` (p_1 e_{11} e_{12} ... e_{1m_1}) (p_2 e_{21} e_{22} ... e_{2m_2}) ... (p_n e_{n1} e_{n2} ... e_{nm_n})), $m_i \geq 0, n \geq 1$

Вычисление условного выражения общего вида выполняется по следующим правилам:

1. последовательно вычисляются условия p_1, p_2, \dots, p_n ветвей выражения до тех пор, пока не встретится выражение p_i , значение которого отлично от `NIL`;
2. последовательно вычисляются выражения-формы $e_{i1} e_{i2} \dots e_{im_i}$ соответствующей ветви, и значение последнего выражения e_{im_i} возвращается в качестве значения функции `cond`;
3. если все условия p_i имеют значение `NIL`, то значением условного выражения становится `NIL`.

Ветвь условного выражения может иметь вид (p_i) , когда $m_i = 0$. Тогда если значение $p_i \neq \text{NIL}$, значением условного выражения `cond` становится значение p_i .

В случае, когда $p_i \neq \text{NIL}$ и $m_i \geq 2$, то есть ветвь `cond` содержит более одного выражения e_i , эти выражения вычисляются последовательно, и результатом `cond` служит значение последнего из них e_{im_i} . Таким образом, в дальнейших вычислениях может быть использовано только значение последнего выражения, и при строго функциональном программировании случай $m_i \geq 2$ обычно не возникает, т.к. значения предшествующих e_{im_i} выражений пропадают.

Использование более одного выражения e_i на ветви `cond` имеет смысл тогда, когда вычисление предшествующих e_{im_i} выражений даёт побочные эффекты, как при вызове функций ввода и вывода, изменении списка свойств атома, а также определении новой функции с помощью `defun`.

К примеру:

```
(cond ((< X 5)(print "Значение x меньше пяти") X) ((= X 10)(print "Значение x равно 10") X) (T(print "Значение x больше пяти, но не 10")X))
```

Значением этого условного выражения всегда будет значение переменной X , но при этом на печать будет выведена одна из трёх строк, в зависимости от текущего значения X .

if

Макрофункция (`If C E1 E2`), встроенная в `MuLisp` и `Common Lisp`, вычисляет значение выражения $E1$, если значение выражения C отлично от `NIL`, в ином случае она вычисляет значение $E2$:

```
(defmacro If (C E1 E2) (list 'cond (list C E1) (list T E2)))
```

Этот макрос строит и вычисляет условное выражение `cond`, в котором в качестве условия первой ветви берётся выражение C (первый аргумент `If`), а выражения $E1$ и $E2$ (второй и третий аргумент `If`) размещаются соответственно на первой и второй ветви `cond`.

К примеру, для макровывода (`If (numberp K) (+ K 10) K`) на этапе макро-расширения будет построена конструкция `(cond ((numberp K) (+ K 10) (T K)))`, а на этапе её вычисления в случае $K=5$ будет получено значение 15.

and/or

К логическим функциям-предикатам относят логическое отрицание `not`, конъюнкцию `and` и дизъюнкцию `or`. Первая из этих функций является обычной, а другие две – особыми, поскольку допускают произвольное количество аргументов, которые не всегда вычисляются.

Логическое отрицание `not` вырабатывает соответственно: $(\text{not NIL}) \Rightarrow T$ и $(\text{not } T) \Rightarrow \text{NIL}$, и может быть определено функцией `(defun not (x) (eq x NIL))`.

Фактически действие этой функции эквивалентно действию функции `null`, работающей не только с логическими значениями T и `NIL`, но и с произвольными

липовскими выражениями. Поэтому, например: $(\text{not } (B ())) \Rightarrow \text{NIL}$

Тем самым, определение функции `not` соответствует липовскому расширенному пониманию логического значения истина.

Две другие встроенные логические функции также используют расширенное понимание истинного значения.

Вызов функции `and`, реализующей конъюнкцию, имеет вид $(\text{and } e_1 e_2 \dots e_n)$, $n \geq 0$.

При вычислении этого функционального обращения последовательно слева направо вычисляются аргументы функции e_i – до тех пор, пока не встретится значение, равное `NIL`. В этом случае вычисление прерывается и значение функции равно `NIL`. Если же были вычислены все значения e_i и оказалось, что все они отличны от `NIL`, то результирующим значением функции `and` будет значение последнего выражения e_n .

Вызов функции-дизъюнкции имеет вид $(\text{or } e_1 e_2 \dots e_n)$, $n \geq 0$.

При выполнении вызова последовательно вычисляются аргументы e_i (слева направо) – до тех пор, пока не встретится значение e_i , отличное от `NIL`. В этом случае вычисление прерывается и значение функции равно значению этого e_i . Если же вычислены значения всех аргументов e_i , и оказалось, что они равны `NIL`, то результирующее значение функции равно `NIL`.

При $n=0$ значения функций: $(\text{and}) \Rightarrow T$, $(\text{or}) \Rightarrow \text{NIL}$.

Таким образом, значение функции `and` и `or` не обязательно равно `T` или `NIL`, а может быть произвольным атомом или списочным выражением.

remove

sabstitute

Остальные функции будут рассмотрены по ходу выполнения работы.

Практические задания

1. Чем принципиально отличаются функции `cons`, `list`, `append`?

`cons` принимает 2 указателя на любые S-выражения и возвращает новую `cons`-ячейку (списковую ячейку), содержащую 2 значения. Если второе значение не `NIL` и не другая `cons`-ячейка, то ячейка печатается как два значения в скобках, разделённые точкой (так называемая точечная пара). Иначе, по сути, эта функция включает значение первого аргумента в начало списка, являющегося значением второго аргумента.

Функция `list`, составляющая список из значений своих аргументов (у которого голова – это первый аргумент, хвост – все остальные аргументы), создает столько списковых ячеек, сколько аргументов ей было передано. Эта функция относится к особым, поскольку у неё может быть произвольное число аргументов, но при этом все аргументы вычисляются.

`append` принимает произвольное количество аргументов-списков и соединяет (сливает) элементы верхнего уровня всех списков в один список. Действие `append` иногда называют конкатенацией списков. В результате должен быть построен новый список.

Например: `(append (list 1 2) (list 3 4)) ==> (1 2 3 4)`.

С точки зрения функционального подхода, задача функции `append` - вернуть список `(1 2 3 4)` не изменяя ни одну из `cons`-ячеек в списках-аргументах `(1 2)` и `(3 4)`. `append` на самом деле создаёт только две новые `cons`-ячейки, чтобы хранить значения 1 и 2, соединяя их вместе и делая ссылку из `CDR` второй ячейки на первый элемент последнего аргумента - списка `(3 4)`. После этого функция возвращает `cons`-ячейку содержащую 1. Ни одна из входных `cons`-ячеек не была изменена, и результатом, как и требовалось, является список `(1 2 3 4)`. Единственная хитрость в том, что результат, возвращаемый функцией `append` имеет общие `cons`-ячейки со списком `(3 4)`. Таким образом, если последний переданный список будет модифицирован, то итоговый список будет также изменен.

Итак, отличия: `cons` является базисной, `list` и `append` – нет; `list` и `append` принимают произвольное количество аргументов (причем аргументами `append` могут быть только списки), `cons` – фиксированное (два); `cons` создает точечную пару или список (в зависимости от второго аргумента), `list` и `append` – список; `cons` и `list` создают новые списковые ячейки (все), а `append` имеет общие списковые ячейки с последним списком.

`list` и `append` определяются с помощью `cons`.

Пусть `(setf lst1 '(a b)); (setf lst2 '(c d))`

Каковы результаты вычисления следующих выражений?

```

1 (cons lst1 lst2) => ((A B) C D)
2 (list lst1 lst2) => ((A B) (C D))
3 (append lst1 lst2) => (A B C D)

```

2. Каковы результаты вычисления следующих выражений, и почему?

reverse переворачивает свой список-аргумент, т.е. меняет порядок его элементов верхнего уровня на противоположный, например: `(reverse '(A (B D) C)) => (C (B D) A)`. **reverse** является примером рекурсии, определение может быть следующим [1]:

```

1 (defun Reverse (L)
2   (cond ((null L) NIL)
3         (T (append (Reverse (cdr L))
4                     (cons (car L) NIL) ))))

```

В этом определении реализована следующая идея рекурсивного построения требуемого списка: он получается из перевернутого хвоста исходного списка присоединением к нему справа первого элемента.

last возвращает последнюю cons-ячейку в списке. Если вызывается с целочисленным аргументом *n*, возвращает *n* ячеек (то есть по умолчанию *n*=1). Если *n* больше или равно количеству cons-ячеек в списке, результатом будет исходный список.

```

1 (reverse ()) => Nil
2 (last ()) => Nil
3 (reverse '(a)) => (A)
4 (last '(a)) => (A)
5 (reverse '((a b c))) => ((A B C))
6 (last '((a b c))) => ((A B C))

```

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

Первый элемент перевернутого списка:

```

1 (defun f3_reverse (lst) (
2   car (reverse lst)
3 ))
4
5 (f3_reverse '(1 2 (3 4)) => (3 4))

```


Рекурсивно:

```
1 (defun f3_recursive (lst) (  
2   if (null (cdr lst))  
3     (car lst)  
4     (f3_recursive (cdr lst))  
5 ))  
6  
7 (f3_recursive '(1 2 (3 4))) => (3 4)
```

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

Перевернутый хвост перевернутого списка:

```
1 (defun f4_reverse (lst) (  
2   reverse (cdr (reverse lst))  
3 ))  
4  
5 (f4_reverse '((0 1) 2 (3 4))) => ((0 1) 2)
```

Рекурсивно:

```
1 (defun f4_recursive (lst) (  
2   if (null (cdr lst))  
3     Nil  
4     (cons (car lst) (f4_recursive (cdr lst)))  
5 ))  
6  
7 (f4_recursive '((0 1) 2 (3 4))) => ((0 1) 2)
```

5. 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 получает 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.

```
1 (defun roll_dice () (+ (random 6) 1))  
2
```

```

3 (defun check_continue_game (result) (not (or (= result 7) (= result 11)
4   )))
5 (defun make_a_move (player_i)
6   (let ((dice1 (roll_dice)) (dice2 (roll_dice)))
7     (if (and (print (list 'player_i '
8       ) (= dice1 dice2) (or (= dice1 1) (= dice1 6)))
9       (and
10        (print (list 'dice1 dice2 '
11          ))
12        (make_a_move player_i)
13        )
14        (and
15         (print (list 'dice1 dice2))
16         (+ dice1 dice2)
17         )
18         )
19       )
20   )
21 (defun compare_results (res1 res2)
22   (if (check_continue_game res2)
23     (and
24       (print (list 'rrrr))
25       (print (list '
26         1 ' res1))
27       (print (list '
28         2 ' res2))
29       (cond
30         ((< res1 res2) (and (print (list '
31           2 '
32           )) 2))
33         ((> res1 res2) (and (print (list '
34           1 '
35           )) 1))
36         ((and (print '(
37           )) 0))
38       )
39       (and (print (list '
40         2 ' res2 '
41         )) 2)
42       )
43   )
44 (defun play_game ()
45   (let ((res1 (make_a_move 1)))
46     (if (check_continue_game res1)
47       (compare_results res1 (make_a_move 2))
48       (and (print (list '
49         1 ' res1 '
50         )) 1)
51     )
52 )

```

44)	
45)	

Литература

- [1] Большакова Елена Игоревна Груздева Надежда Валерьевна. Основы программирования на языке Лисп. М.: Издательский отдел факультета ВМК МГУ имени М.В.Ломоносова (лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2010. с. 112.