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 №3 по дисциплине «Функциональное и логическое программирование»

Тема Работа интерпретатора Lisp

Студент Золотухин А. В.

Группа ИУ7-64б

Оценка (баллы) _____

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

Москва — 2023 г.

Теоретические вопросы

1. Базис Lisp

Базис – это минимальный набор инструментов языка и стркутур данных, который позволяет решить любые задачи.

Базис Lisp :

- атомы (представляются в памяти пятью указателями – name, value, function, property, package) и структуры (представляющиеся бинарными узлами);
- базовые (несколько) функций, функционалов и форм: встроенные — примитивные функции (atom, eq, cons, car, cdr); формы (quote, cond, lambda, eval); функционалы (apply, funcall).

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”);

Более сложные данные – списки и точечные пары (структуры), которые строятся с помощью унифицированных структур – блоков памяти – бинарных узлов.

Определения:

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>,

S-выражение ::= <атом> | <точечная пара>,

<хвост> ::= <список>.

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.Классификация функций

1. Чистые математические функции (имеют фиксированное количество аргументов, сначала выяисляются все аргументы, а только потом к ним применяется функция);
2. Рекурсивные функции (основной способ выполнения повторных вычислений);
3. Специальные функции, или формы (могут принимать произвольное количество аргументов, или аргументы могут обрабатываться по-разному);
4. Псевдофункции (создают «эффект», например, вывод на экран);
5. Функции с вариантами значений, из которых выбирается одно;
6. Функции высших порядков, или функционалы – функции, аргументом или результатом которых является другая функция (используются для построения синтаксически управляемых программ);

Классификаци базисных функций и функций ядра.

1. Селекторы: car и cdr (будут подробнее расссмотрены ниже).
2. Конструкторы: cons и list (будут подробнее расссмотрены ниже).
3. Предикаты – «логические» функции, позволяющие определить структуру элемента:
   - atom возвращает T, если значением её единственного аргумента является атом, иначе – NIL;
   - null возвращает T, если значение его аргумента – NIL (пустой список), иначе – NIL; listp возвращает T, если значением её аргумента является список, иначе – NIL; consp возвращает T, если значением её аргумента является структура, представленная в виде списковой ячейки, иначе – NIL.

4. Функции сравнения (принимают два аргумента, перечислены по мере роста «тщательности» проверки):

   - eq корректно сравнивает два символьных атома. Так как атомы не дублирутюся для данного сеанса работы, то фактически сравниваются соответсвующие указатели. Возвращает T, когда: 1) значением одного из аргументов является атом, и одновременно 2) значения аргументов равны (идентичны). В ином случае значением функции eq является NIL.
   - eql корректно сравнивает атомы и числа одинакового типа (синтетической формы записи). Например, (eql 1 1) вернет T, а (eql 1 1.0) – Nil, так как целое значение 1 и значение с плавающей точкой 1.0 являются представителями различных классов;
   - = корректно сравнивает только числа, причем числа могут быть разных типов. Например, и (= 1 1), и (= 1 1.0) вернет T;
   - equal работает идентично eql, но в дополнение умеет корректно сравнивать списки (считая списки эквивалентными, если они рекурсивно, согласно тому же equal, имеют одинаковую структуру и содержимое; считая строки эквивалентными, если они содержат одинаковые знаки);
   - equalp корректно сравнивает любые S-выражения.

3. Способы создания функций

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 фактические параметры.

  Вычисление функций без имени может быть также 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 $\geqslant$ 0. Её действие аналогично apply, отличие состоит в том, что аргументы применяемой функции F задаются не списком, а по отдельности.

funcall используется тогда, когда во время написания кода количество аргументов известно, apply – когда неизвестно.

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

  (defun имя_функции λ-выражение),

  или в облегченной форме:

  (defun имя_функции $(x_1, x_2, ..., x_k)$ форма), где $(x_1, x_2, ..., x_k)$ – 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. Также для ее вызова можно воспользоваться рассмотренными выше функционалами funcall (например, (foo 1 2 3) === (funcall #'foo 1 2 3)) и apply (например, (apply #'plot plot-data), где plot-data - список, хранящий аргументы).

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 ((x1 p1) (x2 p2) ... (xk pk)) e),

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

4. Работа функций cond, if, and/or

cond

Общий вид условного выражения:

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

Вычисление условного выражения общего вида выполняется по следующим правилам:

1. последовательно вычисляются условия $p_1, p_2, \ldots p_n$ ветвей выражения до тех пор, пока не встретится выражение p_i, значение которого отлично от NIL;

2. последовательно вычисляются выражения-формы $e_{i1}\ e_{i2}\ \ldots\ e_{im_i}$ 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 pi $\neq$ NIL, значением условного выражения cond становится значение pi.

В случае, когда pi $\neq$ NIL и $m_i \geqslant 2$, 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 $m_i \geqslant 2$ 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 макровызова (If (numberp K) (+ K 10) K) на этапе макрорасширения 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: (not NIL) => T и (not T) => 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 лисповскими выражениями. Поэтому, например: (not '(B ())) => NIL

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

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

Вызов функции and, реализующей конъюнкцию, имеет вид (and e1 e2 ... en), $n \geqslant 0$.

При вычислении этого функционального обращения последовательно слева направо вычисляются аргументы функции ei – 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 ei и оказалось, что все они отличны от NIL, то результирующим значением функции and будет значение последнего выражения en .

Вызов функции-дизъюнкции имеет вид (or e1 e2 ... en), $n \geqslant 0$.

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

При n=0 значения функций: (and)=>T, (or)=>NIL.

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