Примеры использования рекурсивного программирования

Артамонов Ю.Н.

Университет "Дубна"
филиал Котельники

7 апреля 2017 г.

Булева алгебра - реализация булевых функций

Покажем насколько простым оказывается решение задачи составления таблицы истинности для произвольной булевой функции. Вначале реализуем сами логические функции. В виду своей простоты начальный код не требует комментарев.

```
;;Логическая конъюнкция
(defun and-m (x y)
  (cond
    ((and (eq x 1) (eq y 1)) 1)
    ((and (eq x 1) (eq y 0)) 0)
    ((and (eq x 0) (eq y 1)) 0)
    ((and (eq x 0) (eq y 0)) 0)
    (t 'Error)))
```

Булева алгебра - реализация булевых функций (продолжение)

```
;;Логическая дизъюнкция
(defun or-m (x y)
  (cond
    ((and (eq x 1) (eq y 1)) 1)
    ((and (eq x 1) (eq y 0)) 1)
    ((and (eq x 0) (eq y 1)) 1)
    ((and (eq x 0) (eq y 0)) 0)
    (t 'Error)))
;;Логическое отрицание
(defun not-m (X)
  (cond
    ((eq X 1) 0)
    ((eq X 0) 1)
    (t 'Error)))
```

Булева алгебра - реализация булевых функций (продолжение)

```
(defun n-and (x y)
  ((lambda (x y) (not-m (and-m x y))) x y))

(defun n-or (x y)
  ((lambda (x y) (not-m  (or-m x y))) x y))

(defun if-m (x y)
  (or-m (not-m x) y))

(defun xor-m (x y)
  (or-m (and-m (not-m x) y) (and-m x (not-m y))))

(defun eqv (x y)
  (or-m (and-m (not-m x) (not-m y)) (and-m x y)))
```

Булева алгебра - лямбды вызов

Обратите также внимание на немного необычную реализацию булевых функций n-and, n-or. Там тело функции представляет собой лямбда-вызов. Например, для n-and имеем ((lambda (x y) (not-m (and-m x y))) x y)) . Т.е. фактические значения переменных x, y передаются в тело выражения (not-m (and-m x y)). Если помните, это называется бета-редукция. На самом деле любая функция в Лиспе представляется в таком виде, а то, как мы пишем - это принятое упрощение, по сути синтаксический сахар.
(defun f (x) ((lambda (y) (+ y 1)) x)) <=> (defun f (x) (+ x 1))
>(f 4)
>5
В Common Lisp сопоставление функции лямбда выражения не дает значения функции, т.е. определение (defun f (x) (lambda (x) (+ x 1))) не вычисляет потом, например, (f 4). Это другое определение, которое задает так называемое *лексическое замыкание*, о котором мы будем говорить позже.

Булева алгебра - форма let

Лямбда выражения также широко используются для присваивания значений локальным переменным.
Рассмотрим лямбда выражение ((lambda (x y) (+ x y) (* x y)) 3 4). В самом лямбда выражении две формы (+ x y) и (* x y), вычисляются обе формы и значение последней возвращается лямбда вызову (лямбда вызов возникает, когда в лямбда выражение подставляют значения, т.е. проводят бета-редукцию). Т.е. если в интерпретаторе ввести
> ((lambda (x y) (+ x y) (* x y)) 3 4)
то получится
>12
Такая конструкция
((lambda (x1 x2 x3 ... xn) форма1 форма2...) a1 a2 a3 ...an)
оказалась настолько полезной, что для нее придумали синтаксический сахар:
(let ((x1 a1) (x2 a2) ...(xn an)) форма1 форма2 ...)

Булева алгебра - let*

Предложение let вычисляется так, что сначала локальным переменным x1,x2... одновременно присваиваются a1,a2... соответственно. Затем вычисляются значения форм: форма1, форма2... Значение последней формы возвращается в качестве значения всей формы let.
Также иногда используется разновидность let — это форма let*. Отличие от let в том, что переменным x1,x2,... значения присваиваются не одновременно а последовательно, т.е. внутри присваивания значения переменной можно использовать ранее определенные значения переменных. Например,
>(let ((x 2) (y (+ x 1))) (cons x y))
выдаст сообщение об ошибке, в то время как
>(let* ((x 2) (y (+ x 1))) (cons x y))
получит так называемый точечный список
>(2 . 3)

Булева алгебра - главная функция

Вернемся к задаче построения таблицы истинности. Необходимо реализовать функцию, которая вычисляет значение заданного булева выражения, когда вместо переменных стоят 1 или 0. Данная реализация может иметь следующий вид:

;;Функция вычисляет выражение от 1-0 аргумента

```
(defun bool-fun (expr)
  (cond
    ((atom expr)
     (if (or (= expr 0) (= expr 1)) expr 'Error))
    ((equal (car expr) 'not-m)
     (funcall (car expr) (bool-fun (eval (second expr)))))
    (t (funcall (car expr) (bool-fun (eval (second expr)))
(bool-fun (eval (third expr))))))))
```

В первой ветви проверяется, если выражение expr является атомом, то если этот атом 1 или 0, то соответствующее значение возвращается в качестве результата. В противном случае печатается сообщение об ошибке.

Булева алгебра - пояснение главной функции

В коде присутствуют две специальные функции:
funcall — это так называемый применяющий функционал. Его смысл состоит в том, что вызов (funcall 'f a b) применяет функцию f к аргументам a и b, т.е. вычисляет (f a b). Особенность в том, что функция может быть заранее не задана, а вычислена как результат какого-то выражения. Например,
>(funcall (car '(+ - * /)) 2 3)
>5
eval — противоположна блокировке Quote (синтаксический сахар '), т.е. выражение вычисляется, а не используется как совокупность символов, из которых оно записано.
Во второй ветви анализируется, если голова выражения expr унарная операция, то эта операция выполняется над результатом булевой функции, которая представлена хвостом выражения expr (вторым аргументом).
И наконец, если голова expr бинарная операция, то она выполняется над результатами булевых функций от второго и третьего аргументов.

Булева алгебра - завершающая стадия

В принципе теперь осталось только получить список всех возможных наборов нулей и единиц для всех переменных заданного выражения, подставить каждый набор вместо переменных и вызвать функцию bool-fun, при этом каждый раз печатается набор и значение bool-fun для этого набора. Искомая реализация может иметь следующий вид:

;;Построение таблицы истинности

```
(defun true-table (expression list-arg)
  (defun make_list (k N)
    (cond
      ((= n 1) (list k))
      (t (cons k (make_list k (- N 1))))))
  (defun make_next_str (L)
    (cond
      ((null L) nil)
      ((= (car L) 0) (cons 1 (cdr L)))
      (t (cons 0 (make_next_str (cdr L))))))
```

Булева алгебра - завершающая стадия

```
(defun set_arg (Y L)
  (cond
    ((Null L) nil)
    (t (let () (set (car Y) (car L))
        (set_arg (cdr Y) (cdr L))))))
(let
    ((N (expt 2  (length list-arg)))
     (L (make_list 0 (length list-arg))))
  (let ()
    (print (reverse (cons 'Itog (reverse list-arg))))
    (loop
(set_arg list-arg L)
(print (reverse (cons (bool-fun expression) (reverse L))))
(setq L (make_next_str L))
(if (= N 1) (return 'ok) (setq N (- N 1)))))))
```

Булева алгебра - пояснение завершающей стадии

Функция true-table имеет два аргумента — булево выражение и список аргументов этого булева выражения. Например, в интерпретаторе задаем
>(true-table '(eqv (and-m (not-m a) (if-m b c)) (n-or a (not-m b))) '(a b c))
Получаем следующий результат:
(A B C ITOG)
(0 0 0 0)
(1 0 0 1)
(0 1 0 0)
(1 1 0 1)
(0 0 1 0)
(1 0 1 1)
(0 1 1 1)
(1 1 1 1)
OK

Булева алгебра - пояснение завершающей стадии

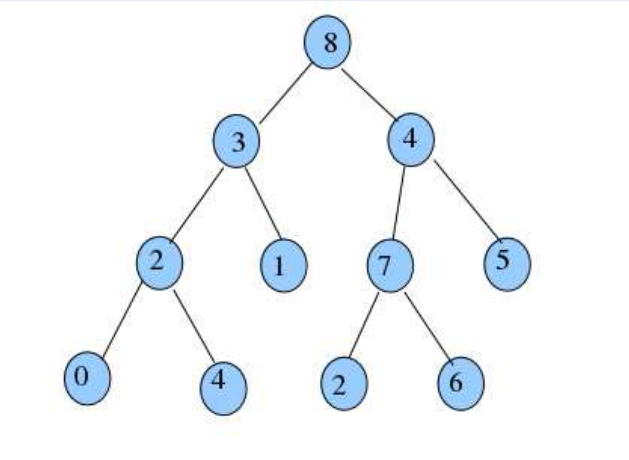
В функции true-table используется три вспомогательных функции:
make_list — строит список из элементов k длины N. Например, (make_list 0 3) даст (0 0 0)
make_next_str — для строки L — набора значений переменных получаем следующую строку. Например, (make_next_str '(0 0 0)) даст (1 0 0), а (make_next_str '(1 1 0)) даст (0 0 1)
В реализации этих функций нет ничего нового, поэтому подробно мы их рассматривать не будем.
set_arg — функция присваивает списку аргументов булева выражения текущие значения. Она имеет два аргумента Y — это список переменных, которым будут присваиваться значения, L — это список самих значений. В реализации функции встречается новая синтаксическая конструкция set. В ветви let последовательно вычисляются две формы - (set (car Y) (car L)) и (set_arg (cdr Y) (cdr L)). В первой форме при помощи функции set одной из переменных списка Y, а именно той, которая является головой этого списка присваивается значение из списка L. Во второй форме функция вызывает себя от «уменьшенного аргумента».

Булева алгебра - пояснение завершающей стадии

Таким образом, мы ввели в наше программирование операцию присваивания set. Выражение (set a b) — присваивает значению выражения a значение выражения в. У этой функции есть разновидность — setq. В отличие от set в функции setq вычисляется только второй аргумент, поэтому в записи (setq a b) самому символу a присваивается значение выражения b.
В основном теле функции в первой ветви let инициализируются локальные переменные $N = 2^k$, где k — это количество аргументов, т.е. эта переменная хранит количество строк таблицы истинности, L - начальная строка таблицы истинности из одних нулей. Во второй ветви let вычисляются две формы (print (reverse (cons 'Вых (reverse list-arg)))) - печатает на экран заголовок таблицы - (A B C ВЫХ), в следующей форме встречается новая конструкция - цикл loop, который выполняет последовательность форм, пока не встретится оператор выхода из цикла Return. В отличие от императивых языков эти конструкции не являются обязательными — они лишь синтаксический сахар. Лисп позволяет расширять свой синтаксис так, чтобы такие конструкции появились без специальных механизмов и зарезервированных слов, но об этом позже!)))

Самостоятельное задание - бинарные деревья

Договоримся о форме представления бинарного дерева в системе Лисп. Пусть дано бинарное дерево вида:



Будем представлять это дерево в виде следующего списка:
(8 (3 (2 (0 nil nil) (4 nil nil)) (1 nil nil)) (4 (7 ((2 nil nil) (6 nil nil)) (5 nil nil)))
Таким образом, от вершины 8 левая половина представлена списком (3 (2 (0 nil nil) (4 nil nil)) (1 nil nil)), а правая половина списком (4 (7 ((2 nil nil) (6 nil nil)) (5 nil nil)). Внутри каждого из этих списков представление аналогично. Например, от вершины 3 левая половина - (2 (0 nil nil) (4 nil nil)), а правая половина - (1 nil nil). Правая и левая половина для листьев дерева — пустые списки.

Самостоятельное задание - задание 1

Реализовать процедуру tree-make, которая будет генерировать случайные бинарные деревья.
Процедура будет иметь два параметра: n — характеризует глубину дерева, m — диапазон чисел, из которого будут случайно выбираться имена вершин.
Идея процедуры следующая:

- если n=0 (нулевая глубина), то формируется список вида (случайное число из диапазона до m nil nil) — это база рекурсии;
- в противном случае формируется список из случайного числа и бинарного дерева, образующего левую половину и бинарного дерева, образующего правую половину, глубина каждой из половин уменьшается случайным образом.

Самостоятельное задание - задание 2

Реализовать функцию peak-count подсчета количества узлов дерева.

Самостоятельное задание - задание 3

Реализовать функцию leaves-count подсчета количества листьев дерева.

Самостоятельное задание - задание 4

Реализовать функцию leaves-list получения списка листьев дерева.

Самостоятельное задание - задание 5

Реализовать функцию story-tree получения n-го яруса дерева.

Самостоятельное задание - задание 6

Реализовать функцию deep-tree нахождения глубины дерева.

Самостоятельное задание - задание 7

Реализовать функцию member-tree проверки принадлежности элемента дереву.

Самостоятельное задание - задание 8

Реализовать функцию replace-knot, которая заменяем все узлы с данным элементом на заданный элемент.