Вычисление в Лиспе.

Лекция 12.

Специальности : 230105, 010501

Понятие формы.

<u>Определение.</u> Под формой понимается такое символьное выражение, значение которого может быть найдено интерпретатором Лиспа.

Вычислимые выражения можно разделить на три группы :

1). Самоопределенные (self-evaluating) формы. К ним относятся все лисповские объекты, которые представляют только сами себя : числа, константы T/NIL, знаки, строки, битовые векторы. К самоопределенным формам относятся также ключи, которые начинаются с двоеточия и определяемые в лямбда-списке через ключевое слово &KEY.

2). Символы, которые используются в качестве переменных.

3). Формы в виде списочной структуры :

- **Вызовы функций и лямбда-вызовы.**
- **Специальные формы для управления вычислением и контекстом.**
- **Макровызовы.**

Лямбда-выражение без фактических параметров не является формой !

Управляющие структуры.

Управляющие структуры (предложения) имеют вид вызовов функций. Записываются в виде скобочных выражений, первый элемент которых соответствует имени управляющей структуры, остальные – аргументам. Результатом вычисления, как и у функции, является значение. Однако предложения не являются вызовами функций и разные предложения используют аргументы по-разному.

Классификация управляющих структур – по виду использования (пример для muLISP) :

Работа с контекстом :
- **QUOTE или блокировка вычисления;**
- **Лямбда-вызовы;**
- **Предложение LET и LET*.**

Последовательное исполнение :
- **Предложения PROG1, PROG2 и PROGN.**

Разветвление вычислений :
- **Условные предложения COND, IF, WHEN, UNLESS;**
- **Выбирающее предложение CASE.**

Итерации :
- **Циклические предложения DO, DO*, LOOP, DOTIMES, DOUNTIL.**

Передача управления :
- **Предложения PROG, GO и RETURN.**

Динамическое управление вычислением :
- **THROW и CATCH, а также BLOCK.**

Последовательные вычисления.

Предложения PROG1, PROG2, PROGN в Коммон Лиспе и muLISP'е позволяют работать с несколькими вычисляемыми формами :

(PROG1 <форма1> <форма2> … <формаN>)

(PROG2 <форма1> <форма2> … <формаN>)

(PROGN <форма1> <форма2> … <формаN>)

Данные предложения имеют переменное число аргументов, которое они последовательно вычисляют и возвращают в качестве значения значение первого (PROG1), второго (PROG2) или последнего (PROGN) аргумента.

Пример :

(progn (setq x 2)(setq y (* 3 x))) возвращает в качестве значения 6, но значение 2 остается связанным с символом x.

В Microsoft muLISP предложение PROG2 описано в библиотеке COMMON.LSP.

Аналогом PROGN в newLISP-tk является предложение BEGIN.

Использование последовательных вычислений при описании фреймовой структуры в muLISP'e.

```
(defun make_slot (frm)
 (progn
    (princ "Input the name of slot : ")
    (setq slot (read))
    (princ "Input the value of slot : ")
    (setq value_of_slot (read))
    (put frm slot value_of_slot)))
(defun make_children (father n_chld)
 ((= n_chld 0) nil)
 (progn
    (make_frame father)
    (make_children father (- n_chld 1))))


(defun build_frm_struct ()
 (progn
    (princ "Input the number of frames : ")
    (setq n_frm (read))
    (loop
       ((= n_frm 0) nil)
       (make_frame nil)
       (setq n_frm (- n_frm 1)))))
```

```
(defun make_frame (father)
 (progn
    (princ "Father frame : ")
    (print father)
    (princ "Input the name of frame : ")
    (setq frm (read))
    (put frm 'frm_name frm)
    (put frm 'father father)
    (princ "Input the number of slots : ")
    (setq n_slots (read))
    (loop
       ((= n_slots 0) nil)
       (make_slot frm)
       (setq n_slots (- n_slots 1)))
    (princ "Input the number of children frames : ")
    (setq n_children (read))
    (make_children frm n_children)))
```

Реализация этой же структуры в newLISP-tk.

```
(define (make_slot frm)
  (begin
   (print "\n" "Input the name of slot : ")
   (setq slot (read-line))
   (print "\n" "Input the value of slot : ")
   (setq value_of_slot (read-line))
   (set frm (cons (list slot value_of_slot) frm))))

(define (make_children father n_chld)
 (cond
   ((= n_chld 0) true)
   ((> n_chld 0)
        (begin
          (make_frame father)
          (make_children father (- n_chld 1))))))
```

```
(define (make_frame father)
 (begin
     (print "\n" "Father frame : ")
     (print father)
     (print "\n" "Input the name of frame : ")
     (setq frm (sym (read-line)))
     (set frm (list (list 'frm_name frm)
                    (list 'father father)))
     (print "\n" "Input the number of slots : ")
     (setq n_slots (int (read-line)))
     (while (> n_slots 0)
        (make_slot frm)
        (setq n_slots (- n_slots 1)))
     (print "\n" "Input the number of children frames : ")
     (setq n_children (int (read-line)))
     (make_children frm n_children)))
```

```
(define build_frm_struct
 (begin
     (print "\n" "Input the number of frames : ")
     (setq n_frm (int (read-line)))
     (while (> n_frm 0)
        (make_frame 'empty)
        (setq n_frm (- n_frm 1)))))
```

Условные предложения в muLISP'e.

Ранее нами были рассмотрены условные предложения IF и COND. Условные предложения WHEN и UNLESS являются частными случаями IF :

WHEN : если условие соблюдается, то выполняются формы.

UNLESS : если условие не соблюдается, то выполняются формы.

(WHEN <условие> <форма1> <форма2> <форма3> …)

(UNLESS <условие> <форма1> <форма2> <форма3> …)

```
; Пример использования
; условного предложения when.
; Вывод элементов списка на экран
(defun print_list_fun (lst number)
 (when (not (null lst))
      (princ "element number ")
      (princ number)
      (princ " is : ")
      (print (car lst))
      (print_list_fun (cdr lst)(+ number 1))
      ))
(defun print_list (lst)
 (print_list_fun lst 1))
```

```
; Пример использования
; условного предложения unless.
; Вывод элементов списка на экран
(defun print_list_fun (lst number)
 (unless (null lst)
      (princ "element number ")
      (princ number)
      (princ " is : ")
      (print (car lst))
      (print_list_fun (cdr lst)(+ number 1))
      ))
(defun print_list (lst)
 (print_list_fun lst 1))
```

Условные предложения while и unless в newLISP-tk.

```
; Пример использования
; условного предложения while.
; Вывод элементов списка на экран

(define (print_list lst)
 (setq number 1)
 (setq lst_temp lst)
 (while (not (null? lst_temp))
   (print "element number " number " is : "
          (first lst_temp) "\n")
   (setq number (+ number 1))
   (setq lst_temp (rest lst_temp)))
 (setq number nil)
 (setq lst_temp nil))
```

```
; Пример использования
; условного предложения unless.
; Вывод элементов списка на экран

(define (print_list_fun lst number)
 (unless (null? lst)
   (silent
     (print "element number " number
            " is : " (first lst) "\n")
     (print_list_fun (rest lst) (+ number 1)))
  nil))

(define (print_list lst)
 (print_list_fun lst 1))
```

В newLISP-tk предложение *when* отсутствует. Предложение *while* в newLISP-tk сходно с предложением *when* в muLISP'е по проверке условия, но является циклическим предложением : вычисления в теле while будут продолжаться до тех пор, пока условие истинно.

Предложение выбора.

Помимо “классических” условных предложений, в Лиспе используют выбирающее предложение CASE (аналог оператора выбора в Паскале) :

(CASE <ключ>

(<список-ключей1> <форма11> <форма12> …)

(<список-ключей2> <форма21> <форма22> …) …).

Сначала в форме CASE вычисляется значение ключевой формы <ключ>. Затем его сравнивают с элементами списков ключей <список-ключей i>. Когда в списке найдено значение ключевой формы, начинают вычисляться соответствующие формы <формаi1> <формаi2> … , значение последней из которых и возвращается в качестве значения всего предложения CASE.

В Microsoft muLISP предложение CASE отсутствует, но предложения WHEN и UNLESS описаны в библиотеке COMMON.LSP

Динамическое прекращение вычислений.

Динамическое прекращение вычислений из некоторого вычислительного контекста с выдачей результата в более раннее состояние без осуществления нормальной последовательности возвратов из всех вложенных вызовов производится в случаях :

- **Нахождения нужного решения при бесполезности поиска альтернатив;**
- **Обнаружения вложенной программой ошибки, по которой можно сделать вывод о бесполезности, либо вреде дальнейших вычислений;**
- **Завершения процесса “порождения и проверки” в задачах искусственного интеллекта, в частности, моделирования игр.**

Динамическое прекращение вычислений в Лиспе есть в некотором роде аналог использованию отсечений в логическом программировании.

Формы передачи управления.

В Коммон Лиспе, newLISP-tk и Microsoft muLISP’е динамическое прерывание вычислений можно запрограммировать с помощью форм CATCH и THROW. Подготовка к прерыванию осуществляется специальной формой CATCH. Её синтаксис в muLISP’е и newLISP-tk :

(CATCH <метка> <форма1> <форма2> …)

При вычислении формы сначала вычисляется <метка>, а затем формы <форма1>, <форма2> … слева направо. Значением всей формы будет последнее значение при условии, что во время вычисления непосредственно этих форм (или форм, вызванных из них) не встретится предложение THROW :

(THROW <метка> <значение>).

Если аргумент <метка> вызова THROW представляет собой тот же лисповский объект, что и метка в форме CATCH, то управление передается обратно в форму CATCH и его значением станет значение второго аргумента формы THROW.

Пример (muLISP) : выделение класса объекта.

```
; Фрагмент транслитератора
(defun is_a_sym (obj)
 ((and (symbolp obj)
       (null (is_a_sign obj)))
       (throw return_result 'SYMBOLIC))
 nil)
(defun is_a_num (obj)
 ((numberp obj)(throw return_result 'NUMERIC))
 nil)
(defun is_a_sign (obj)
 ((or (equal obj '+)
      (equal obj '-)
      (equal obj '*)
      (equal obj '/)
      (equal obj '=)
      (equal obj '>)
      (equal obj '<))(throw return_result 'SPECIAL_SYMBOL))
 nil)
(defun atom_kind (obj)
 (catch 'return_result
        (is_a_sym obj)(is_a_num obj)(is_a_sign obj)))
```

Формы передачи управления в newLISP-tk.

В newLISP-tk метки не используются, форма *catch* получает одну вычисляемую форму в качестве аргумента, а полученное значение в качестве побочного эффекта связывается с символом – вторым (необязательным) аргументом.

Пример (предыдущий слайд) – распознавание класса объекта.

```
; Фрагмент транслитератора
(define (is_a_sym obj)
 (cond
  ((and (symbol? obj)(not (is_a_sign obj)))(throw 'SYMBOLIC))
  (true nil)
 ))

(define (is_a_num obj)
 (cond
  ((number? obj)(throw 'NUMERIC))
  (true nil)
 ))

(define (is_a_sign obj)
 (cond
   ((or (= obj '+)(= obj '-)(= obj '*)(= obj '/)(= obj '=)(= obj '>)(= obj '<))
     (throw 'SPECIAL_SYMBOL))
   (true nil)
 ))

(define (atom_kind obj)
 (catch
  (or (is_a_sym obj)(is_a_num obj)(is_a_sign obj))
  'return_result))
```

PROG-механизм.

В старых Лисп-системах существует PROG-механизм. Он позволяет :

•Реализовывать последовательное вычисление форм;

•Организовывать циклы с помощью команд перехода;

•Использовать локальные переменные формы.

Структура предложения PROG :

(PROG m1 m2 … mN)

<оператор 1> <оператор 2>

…

<оператор M>)

Переменные mi есть локальные статические переменные формы, которые можно использовать для хранения промежуточных результатов (как это делается при программировании на операторных языках). Если некоторая форма <оператор i> является символом или целым числом, то это метка перехода, на которую можно передать управление оператором GO : (GO <метка>). GO не вычисляет своего аргумента. Кроме того, в PROG-механизм входит оператор окончания вычисления и возврата значения : (RETURN <результат>).

Пример (muLISP) : определение функции возведения в степень.

```
; Вычисление степени
; с использованием предложения PROG
(defun expt3 (x n)
 (prog (result)
       (setq result x)
  loop1
       (if (equal n 1)
           (return result))
       (setq result (* result x))
       (setq n (- n 1))
       (go loop1)))
```

newLISP-tk : предложения do-until и do-while как альтернатива традиционному prog-механизму.

В newLISP-tk предложения наподобие PROG, GO и RETURN отсутствуют. Функция возведения в степень с использованием операторного стиля на newLISP-tk может быть описана на основе циклических форм *do-until* и *do-while* :

```
; Вычисление степени в newLISP-tk
; с использованием предложения do-until

(define (expt3_do_until x n)
  (setq result x)
  (cond
    ((= n 0) 1)
    ((= n 1) result)
    (true (do-until (= n 1)
            (setq result (* result x))
            (setq n (- n 1))
          ) result )))
```

```
; Вычисление степени в newLISP-tk
; с использованием предложения do-while

(define (expt3_do_while x n)
  (setq result x)
  (cond
    ((= n 0) 1)
    ((= n 1) result)
    (true (do-while (> n 1)
            (setq result (* result x))
            (setq n (- n 1))
          ) result )))
```

Выводы.

1. **При эквивалентности вычислительных возможностей свойства итеративных (в нашем случае – основанных на использовании механизма передачи управления) и рекурсивных программ может существенно отличаться.**
2. **Итеративные программы, будучи более длинными и трудными в осуществлении, позволяют получать результат быстрее и проще в силу двух причин :**
   - **Ориентации в общем случае вычислительных машин на последовательные вычисления;**
   - **Отсутствие у ряда трансляторов возможности преобразования рекурсивного определения в итеративное.**
3. **Рекурсивное программирование в общем случае более короткое и содержательное. В наибольшей степени рекурсивная организация обработки полезна для тех данных, которые рекурсивны по своей природе.**