Функциональное и логическое программирование

Лекция 2

1.3 Определение функций пользователем

1.3.1 Лямбда-функции

Основа определения и вычисления функций – лямбда-исчисление Черча (формализм описания функций).

Для описания функции используется *лямбда-выражение*:

(**LAMBDA** (x_1 x_2 … x_n) S_1 S_2 … S_k)

Формальные параметры

s-выражения, образуют тело функции

Список формальных параметров называется *лямбда-списком*.

Лямбда-выражение соответствует определению функции.

Пример 1: $xy + z$

```
( lambda ( x y z) (+ (* x y) z))
```

Лямбда-выражение нельзя вычислить, оно не имеет значения. Но можно организовать *лямбда-вызов* (соответствует вызову функции):

(лямбда-выражение a_1 a_2 … a_n),

где a_1, a_2, …, a_n – вычислимые s-выражения, задающие вычисления фактических параметров.

Пример 2: 3 · 4 + 5

((lambda (x y z) (+ (* x y) z)) 3 4 5)

2 этапа вычисления лямбда-вызова:

1. Вычисляются значения фактических параметров и соответствующие формальные параметры связываются с полученными значениями.
2. С учетом новых связей вычисляется тело функции, и последнее вычисленное значение возвращается в качестве значения лямбда-вызова.

После завершения лямбда-вызова фактические параметры получают те связи, которые были у них до вычисления лямбда-вызова, т.е. происходит передача параметров по значению.

Лямбда-вызовы можно ставить как на место тела функции, так и на место фактических параметров.

Лямбда-выражение является чисто абстрактным механизмом для определения и описания вычислений. Это безымянная функция, которая пропадает сразу после вычисления значения лямбда-вызова. Ее нельзя использовать еще раз, т.к. она не имеет имени.

1.3.3 Ключевые слова параметров в лямбда-списке

При определении функции можно в лямбда-списке использовать ключевые слова, с помощью которых можно по-разному трактовать аргументы функции при ее вызове. Ключевое слово начинается символом &, записывается перед параметрами, на которые действует, и его действие распространяется до следующего ключевого слова. Параметры, указанные до первого & обязательны при вызове.

Ключевое слово	Значение ключевого слова
&optional	необязательные параметры

Для необязательных параметров можно указать значение при его отсутствии (по умолчанию nil).

Пример:

```
(defun f(x &optional (y '(a b)) z)
  (list x y z)
)
```

Можно обращаться к функции с разным количеством параметров:

(f 1 2 3) → (1 2 3)

(f 1 2) → (1 2 NIL)

(f 1) → (1 (a b) NIL)

1.4 Предикаты

Если перед вычислением функции необходимо убедиться, что ее аргументы принадлежат области определения, или возникает задача подсчета элементов списка определенного типа, то используют специальные функции – предикаты.

Предикатом называется функция, которая используется для распознавания или идентификации и возвращает в качестве результата логическое значение – специальные символы t или nil.

Часто имена предикатов заканчиваются на **P** (от слова Predicate).

(**ATOM** s-выражение) – проверяет, является ли аргумент атомом.

(atom 'x) → t
(atom nil) → t
(atom '(1 2)) → nil

(**LISTP** s-выражение) – проверяет, является ли аргумент списком.

(listp nil) → t
(listp 'x) → nil

(**SYMBOLP** s-выражение) – проверяет, является ли аргумент символом.

(symbolp '*) → t

(**NUMBERP** s-выражение) – проверяет, является ли аргумент числом.

(**NULL** s-выражение) – проверяет, является ли аргумент пустым списком.

(null nil) → t

Предикаты для работы с числами:

Проверка на равенство:

(= n_1 … n_m),

где n_i – число или связанная с числом переменная.

Проверка на упорядоченность или попадание в диапазон:

(< n_1 … n_m),

где n_i – число или связанная с числом переменная.

(< 1 x 2)

Аналогично определяются предикаты: > ; <= ; >=; /=

Предикат для сравнения s-выражений

(**EQUAL** s_1 s_2) - возвращает значение t, если совпадают внешние структуры s-выражений (аргументов функции).

eq, eql

Пример:

(equal '(1 2) '(1 2)) → t

1.5 Псевдофункция SETQ

Символы могут обозначать представлять другие объекты.

Связать символ с некоторым значением можно при помощи функции **SETQ**.

(**SETQ** p_1 s_1 … p_n s_n) – возвращает значение последнего аргумента (p_i-символ, s_i-s-выражение).

Это псевдофункция. Побочным эффектом ее работы является связывание символов-аргументов с нечетными номерами со значениями вычисленных s-выражений – четных аргументов. Все образовавшиеся связи действительны в течение всего сеанса работы с интерпретатором Лиспа.

Пример:

(setq a 1 b 2 c (+ 6 7)) $\rightarrow$ 13

Побочный эффект: a – 1, b – 2, c – 3

1.6 Разветвление вычислений

Существует специальная синтаксическая форма – предложение:

(**COND**

(P_1 V_1)

(P_2 V_2)

……

(P_n V_n)

),

где P_i – предикат, V_i – вычислимое выражение.

Вычисление значения **COND**:

Последовательно вычисляются предикаты P_1, P_2, … до тех пор, пока не встретится предикат, возвращающий значение отличное от nil. Пусть это будет предикат P_k. Вычисляется выражение V_k и полученное значение возвращается в качестве значения предложения **COND**. Если все предикаты предложения **COND** возвращают nil, то предложение **COND** возвращает nil.

Рекомендуется в качестве последнего предиката использовать специальный символ t, тогда соответствующее ему выражение будет вычисляться во всех случаях, когда ни одно другое условие не выполняется.

(**COND**

(P_1 V_1)
(P_2 V_2)
……
(t V_n)

)

Допустимо следующие использования:

1. (P_i). Если значение P_i отлично от nil, то **COND** возвращает это значение.
2. (P_i V_{i1}… V_{ik}). Если значение P_i отлично от nil, то **COND** последовательно вычисляет V_{i1}… V_{ik} и возвращает последнее вычисленное значение V_{ik}.

В предикатах можно использовать логические функции: AND, OR, NOT.
В случае истинности предикат **AND** возвращает значение своего последнего аргумента, а предикат **OR** - значение своего первого аргумента, отличного от nil.

1.7 Рекурсия

Функция называется *рекурсивной*, если в определяющем ее выражении содержится хотя бы одно обращение к ней самой (явное или через другие функции).

Работа рекурсивной функции

Когда выполнение функции доходит до рекурсивной ветви, функционирующий вычислительный процесс приостанавливается, а запускается с начала новый такой же процесс, но уже на новом уровне.

Прерванный процесс запоминается, он начнет исполняться лишь при окончании запущенного им нового процесса. В свою очередь, новый процесс так же может приостановиться и т.д. Таким образом, образуется стек прерванных процессов, из которых выполняется лишь последний запущенный процесс.

Функция будет выполнена, когда стек прерванных процессов опустеет.

Ошибки при написании рекурсивных функций:

- ошибочное условие, которое приводит к бесконечной рекурсии;
- неверный порядок условий;
- отсутствие проверки какого-нибудь случая.

Рекурсия хорошо подходит для работы со списками, так как списки могут содержать в качестве элементов подсписки, т.е. иметь рекурсивное строение. Для обработки рекурсивных структур естественно использовать рекурсивные функции.

В Лиспе рекурсия используется также для организации повторяющихся вычислений.

1.7.1 Трассировка функций

Включение трассировки:

(**TRACE** <имя функции>) – возвращает имя трассируемой функции или nil.

Если трассируется несколько функций, то их имена – аргументы **TRACE**.

Если была включена трассировка, то при обращении к функции будут отображаться имена вызываемых функций, их аргументов и возвращаемые значения после вычислений.

Цифрами обозначаются уровни рекурсивных вызовов.

После знака ==> указываются возвращаемые значения соответствующего рекурсивного вызова.

Выключение трассировки:

(UNTRACE)

Если отключается трассировка некоторых функций, то их имена - аргументы **UNTRACE**.

1.7.2 Простая рекурсия

Рекурсия называется *простой*, если вызов функции встречается в некоторой ветви лишь один раз. В процедурном программировании простой рекурсии соответствует обыкновенный цикл.

Виды простой рекурсии:

- рекурсия по значению (рекурсивный вызов определяет результат функции);
- рекурсия по аргументу (результат функции – значение другой функции, аргументом которой является рекурсивный вызов исходной функции).

При написании рекурсивных функций старайтесь условия останова рекурсии ставить в начало, делайте проверку всех возможных случаев. Попытайтесь проговорить алгоритм словами.

Пример 1: Определим функцию **FACT**, вычисляющую факториал.

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

```
(defun fact(n)
 (cond
   ((= n 0) 1)
   (t (* n (fact (- n 1))))
 ))
```

Это рекурсия по аргументу.

Пример 2: Определим функцию **COPY**, копирующую список на верхнем уровне (без учета вложенностей).

(a b c d)

Умеем копировать (b c d) => (cons (car l) (copy (cdr l)))

```
(defun copy(L)
 (cond
  ((null L) L)
  (t (cons (car L)(copy (cdr L))))
))
```

Это рекурсия по аргументу.

Пример 3: Определим функцию **MEMBER_S**, проверяющую принадлежность s-выражения списку на верхнем уровне. В случае, если s-выражение принадлежит списку, функция возвращает часть списка, начинающуюся с первого вхождения s-выражения в список.
В Лиспе имеется аналогичная встроенная функция **MEMBER** (но она использует в своем теле функцию **EQ**, поэтому не работает для вложенных списков).

```
(defun member_s(s L)
 (cond
  ((null L) L)
  ((equal s (car L))L)
  (t(member_s s (cdr L)))
))
```

Это рекурсия по значению.

Пример 4: Определим функцию **REMOVE_S**, удаляющую все вхождения заданного s-выражения в список на верхнем уровне. В Лиспе имеется аналогичная встроенная функция **REMOVE**, но она не работает для вложенных списков.

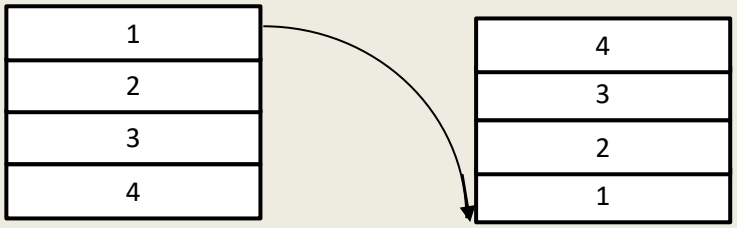
```
(defun remove_s(s L)
 (cond
  ((null L)L)
  ((equal s (car L))(remove_s s (cdr L)))
  (t (cons (car L)(remove_s s (cdr L))))
))
```

Для разных условий предложения COND имеем рекурсию по значению и по аргументу.

1.7.3 Использование накапливающих параметров

При работе со списками их просматривают слева направо. Но иногда более естественен просмотр справа налево. Например, обращение списка было бы легче осуществить, если бы была возможность просмотра в обратном направлении. Для сохранения промежуточных результатов используют вспомогательные параметры.

Пример 1: Определим **REVERSE1**, обращающую список на верхнем уровне, с дополнительным параметром для накапливания результата обращения списка.




```
(defun reverse_1(L &optional L1)
 (cond
   ((null L) L1)
   (t (reverse_1 (cdr l) (cons (car L) L1)))
))
```

Это рекурсия по значению.

Пример 2: Определим функцию **POS**, определяющую позицию первого вхождения s-выражения в список (на верхнем уровне).

```
(defun pos(s L &optional (n 1))
 (cond
   ((null L) L)
   ((equal s (car L)) n)
   (t (pos s (cdr L) (+ n 1)))
))
```

Это рекурсия по значению.