# Работа с графами. Представление знаний.

# Пример: поиск пути в лабиринте

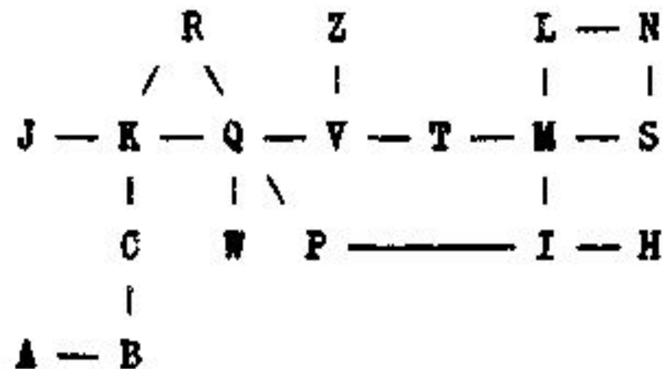
**Задача: поиск пути в лабиринте. Представим лабиринт в виде отдельных комнат, соединенных проходами. Комнаты обозначим идентификаторами.**

Составим список, в котором после каждого имени комнаты укажем список из имен комнат, с которыми данная комната непосредственно соединена проходами. Полученный список присвоим переменной LABYRINTH. Так будет задан лабиринт в программе.

**Задача состоит в том, чтобы найти все пути без циклов (где нет комнат, проходимых более одного раза) из одной заданной комнаты в другую. Путь будем задавать списком из пройденных комнат.**

# Лабиринт

**Пусть имеется следующий лабиринт:**




**Он будет представлен в программе так:**

(SETQ LABYRINTH
'(A (B) B (C A) C (B K) H (I) I (P M H) J (K) K (J C Q R) L (M N) M (T I L S) N (L S) P (Q I) Q (V P R K W) R (Q K) S ( M N) T (M V) V (Q T Z) W (Q) Z (V)))

# Лабиринт. Программа.

Функция **WAY** выполняет поиск путей из A в B1:

```
(DEFUN WAY (A B1) (SETQ B B1) (PRLISTS  (PATH A ())))
```

Функция **PATH** выдает список, содержащий все пути из комнаты **A** в комнату **B** (**B** — глобальная переменная для **PATH**).

Второй аргумент задает пройденный путь. В начальный момент — это NIL.

---

Функция **PRLISTS** служит для выдачи на экран найденных ею путей.

```
(DEFUN PRLISTS (L)
    (COND  ((NULL L) 0)
           (T (PRINT (CAR L)) (+ 1 (PRLISTS (CDR L)))) ))
```

Значением **PRLISTS** является число найденных путей.

# Лабиринт. Функции PATH, NEXT

```
(DEFUN   PATH (TR  P)
    (COND  ((EQ TR  B)  (LIST (REVERSE (CONS  B  P))))
         ((MEMBER TR  P) ())
         (T  (NEXT  (CADR  (MEMBER  TR  LABYRINTH))
                                    (CONS  TR  P)))  ))
```

1 проверка: совпадает ли имя текущей комнаты **TR** с именем конечной комнаты **B**. Если да, то выдается список, содержащий найденный путь.

2 проверка: не была ли ранее пройдена комната **TR**. Если да, то значением **P** будет **NIL**, что означает: путь не найден.

Иначе: продвижение в соседние комнаты.

Исходя из текущего пути **P**, формирует список путей, получаемых при переходе в соседние комнаты:

```
(DEFUN   NEXT (N  P)
    (COND  ((NULL   N)  NIL)
        (T  (APPEND  (PATH  (CAR  N)  P) (NEXT  (CDR N)  P)))  ))
```

# Результаты работы программы

```
CL-USER 5 : 3 > (way `a `r)

(A B C K Q R)
(A B C K R)
2

CL-USER 6 : 3 > (way `j `i)

(J K Q V T M I)
(J K Q P I)
(J K R Q V T M I)
(J K R Q P I)
4
```

# Остовное дерево

Пусть **G=(N, A)** – неориентированный связный граф.

*Остовным деревом* **S** графа **G** называется неориентированное дерево вида

$$\mathbf{S=(N, T), T \in A}$$

# Алгоритм построения остовного дерева

*A* – множество ребер графа G: ((a b)(b c)(c d)…)
*T* – искомое остовное дерево
*V* – множество узлов графа вида: ((a) (b) (c) …)
Алгоритм:

1) Формируется множество *V*;
2) Последовательно выбираются ребра *(v, w)* из *A*. Проверяется, принадлежат ли узлы *v* и *w* разным подмножествам множества *V*. Если да, то эти два подмножества объединяются, объединение добавляется к множеству *V*, а исходные два подмножества удаляются из *V*. Ребро *(v, w)* добавляется к множеству *T*.

# Формирование множества V

```
(defun list_V (gr v)
   (mapcar `list (list2_V (lin_sp gr) v)))
; преобразование множества ребер в линейный список
(defun lin_sp (gr)
   (cond ((null gr) nil)
             (t (append (car gr) (lin_sp (cdr gr)) )) ))
; формирование списка множества вершин вида (a b c …)
(defun list2_V (z v)
    (cond ((null z) v)
              ((member (car z) v) (list2_V (cdr z) v))
              (t  (list2_V (cdr z) (cons (car z) v) )) ))
```

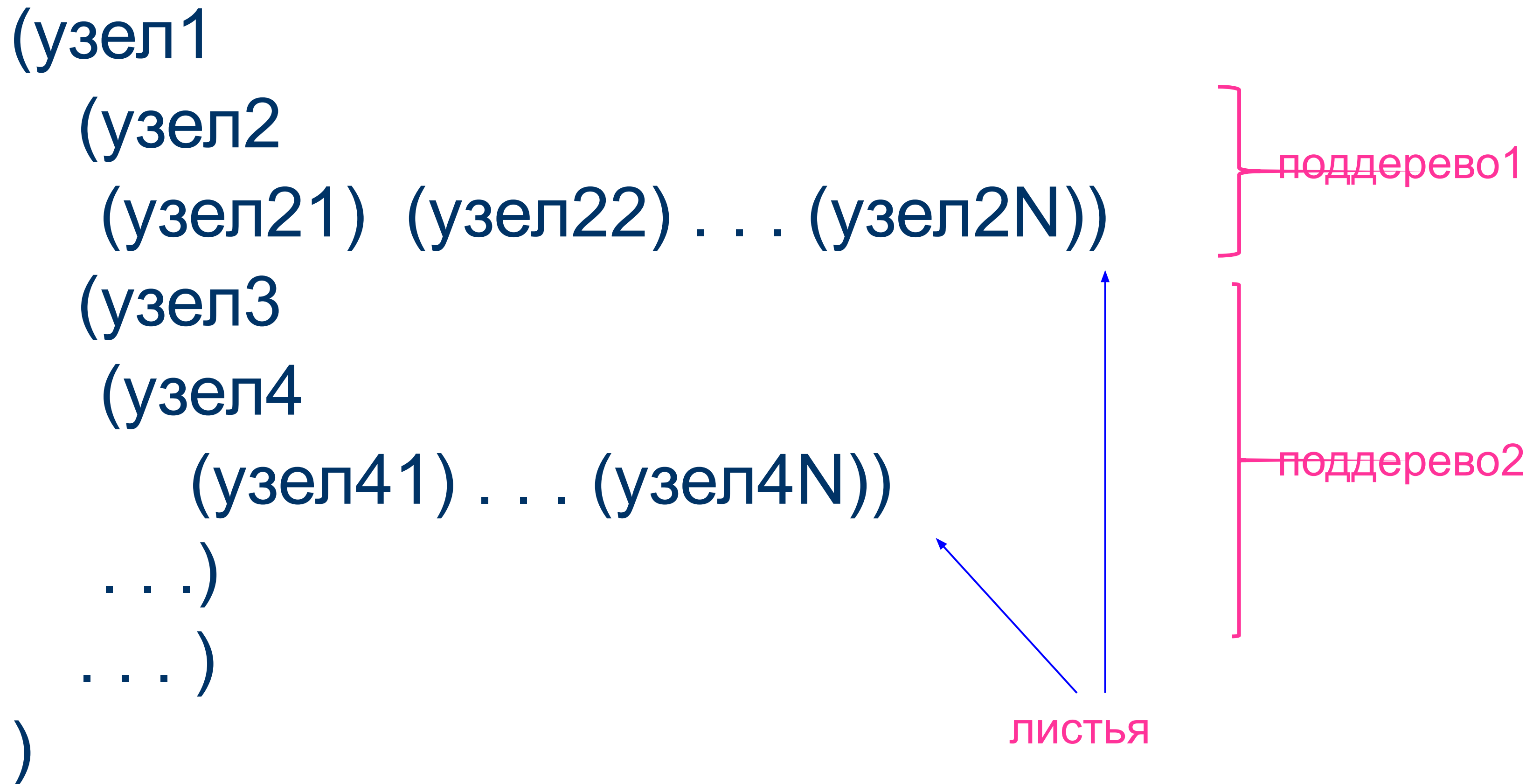
# Формирование остовного дерева

```
(defun ost2 (graf)
    (ost_gr  graf  ()  (list_V  graf  ()) ))

(defun ost_gr (old  new  v)
   (( lambda (x y)
      (cond ((null old) new)
               ((dif x y v) (ost_gr (cdr old)(cons (car old) new)
                                  (inst_del  x  y  v ()) ))
               (t  (ost_gr (cdr old)  new  v)) ))
   (caar old) (cadar old) ) )
```

# Вспомогательные функции

```
; проверка, принадлежат ли вершины x, y разным подмножествам
(defun dif (x y v)
   (cond ((null v) t)
            ((and (member x (car v)) (member y (car v))) nil)
            (t (dif x y (cdr v))) ))

; объединение подмножеств, исключение старых из V
(defun inst_del (x  y  v  pr)
  (cond ((null v) (list pr))
           ((member x (car v)) (inst_del  x  y (cdr v) (append (car v) pr)))
           ((member y (car v)) (inst_del  x  y (cdr v) (append (car v) pr)))
           (t   (cons (car v) (inst_del x y (cdr v) pr)) )  ))
```

# Представление деревьев



# Основные действия над деревьями

- Поиск элемента в дереве
- Включение элемента в дерево
  - Расщепление имеющейся ветви
  - Вырастание новой ветви
- Стяжение ветви – склеивание узлов

# Поиск элемента в дереве вида (1 (2 (3 (4))(5)) (6 (7)) (8 (9)(10)) )

```
; поиск узла в дереве
(defun search_1 (x  tree)
    (cond ((null tree) nil)
              ((equal (car  tree) x) t)
              (t  (search_2  x  (cdr  tree)))))
; поиск поддерева, содержащего заданный узел
(defun search_2 (x  tr)
    (cond ((null tr) nil)
              ((search_1  x  (car tr)) t)
              (t  (search_2  x  (cdr tr))) ))
```

# Включение элемента в дерево: аргументы, условие пустого дерева

Необходимо вставить узел new между узлами prnt и chld

```
(defun split (prnt new chld tr)
    (cond
; окончание поиска – не нашли куда вставить узел
     ((null  tr) nil)
```

# Включение элемента в дерево: расщепление, вырастание имеющейся ветви

```
; расщепление имеющейся ветви
((and (not(null chld))(equal(car tr) prnt) (equal
                    (caadr tr) chld))
      (cons (car tr)
          (cons (list new (cadr tr)) (cddr tr))))
; вырастание новой ветви
 ((and (null chld)(equal(car tr)prnt))
      (cons (car tr)
          (cons (list new)  (cdr tr))))
```

# Включение элемента в дерево (продолжение)

```
; на данном уровне нет искомых узлов
      (t
        (cons (car tr)
              (split_tr  prnt  new  chld  (cdr tr)  ))) ))
```

---

```
; обработка списка дочерних узлов
(defun split_tr (prnt  new  chld  tr )
   (cond ((null tr) nil)
             (t (cons (split  prnt  new chld (car tr) )
                          (split_tr  prnt  new  chld (cdr tr) ) ))))
```

# Список свойств

С символом можно связывать не только значение, но и информацию, называемую списком свойств (property list).

Например, рассмотрим информацию о **Лене**:

| Свойство | Значение |
|---|---|
| Возраст | 28 |
| Профессия | Юрист |
| Зарплата | 90 |
| Дети | Ира, Юра, Петя |

Список свойств в этом случае выглядит:

**(возраст 28 профессия юрист зарплата 90 дети (Ира Юра Петя))**

# Присвоение свойства

Задать новое свойство:

**( setf ( get <символ> <свойство>) <значение>)**

( setf ( get ‘Лена ‘дети) \`(Ира Юра Петя)) ==> (Ира Юра Петя)

( setf ( get ‘Лена ‘зарплата) 90) ==> 90

( setf ( get ‘Лена ‘профессия) \`юрист) ==> юрист

( setf ( get ‘Лена ‘возраст) 28) ==> 28

У каждого свойства только одно значение.

При внесении свойства, оно помещается в начало списка свойств.

Замена значения свойства – повторным присвоением.

# Чтение свойства

Узнать свойство атома можно используя функцию:

**(GET <символ> <свойство>)**

возвращает значение свойства

( get ‘Лена ‘возраст) => 28

( get ‘Лена ‘дети) => (Ира Юра Петя)

( get ‘Лена ‘хобби) => nil

# Удаление и просмотр информации свойств

**Удаление свойства**

Удаление свойства и его значения производится функцией

**(remprop <символ> <свойство>)**

(remprop ‘Лена ‘возраст)   =>  T

**Информация о списке свойств**

Функция **(SYMBOL-PLIST <символ>)** даст информацию о списке свойств

(SYMBOL-PLIST ‘Лена)   =>  (возраст 28 профессия юрист зарплата 90 дети (Ира Юра Петя))

# Разрушающие функции и список свойств

```
CL-USER 30 : 2 > X
(A 0 C)

CL-USER 34 : 2 > ( setf ( get ‘Лена ‘дети) x)
(A 0 C)

CL-USER 35 : 2 > (symbol-plist `Лена)
(дети (A 0 C))

CL-USER 36 : 2 > (rplaca x 2)
(2 0 C)

CL-USER 37 : 2 > (symbol-plist `Лена)
(дети (2 0 C))
```

# Пример. Дифференцирование выражений

Напишем программу дифференцирования алгебраических выражений. Для наглядности ограничимся алгебраическими выражениями в следующей форме:

**(+ x y) (* x y)**

Сложение и умножение можно свободно комбинировать.
Например,
**(*(+ a ( * a b)) c)**

# Пример программы

l – арифметическое выражение

x – имя переменной, по которой берется производная

```
(defun ddif (l x)
   (cond ((atom l)(cond ((eq l x) 1)
                                        (t 0)))
            ((eq (car l) `+)(list `+ (ddif (cadr l) x)
                                              (ddif (caddr l) x)))
            ((eq (car l) `*)(list `+
                                    (list `* (ddif (cadr l) x) (caddr l))
                                    (list `* (ddif (caddr l) x) (cadr l))))
            (t l)))
```

# Работа программы

```
> (setq x 3)
3

> (ddif `(+ x (* 3 x)) `x)          ; (x+3x)
(+ 1 (+ (* 0 X) (* 1 3)))           ; (1+0*x+1*3)

> (eval (ddif `(+ x (* 3 x)) `x))
4

> (ddif `(+ x (+ 5 (* 2 (* x x)))) `x)                        ;(x+5+2*x^2)
(+ 1 (+ 0 (+ (* 0 (* X X)) (* (+ (* 1 X) (* 1 X)) 2))))   ;(1+0+0*x^2+(x+x)*2)

> (eval (ddif `(+ x (+ 5 (* 2 (* x x)))) `x))
13
```

# Модульный подход

Приведенная программа неудобна, так как ее трудно расширять, приходится все группировать в один cond.

Она не является модульной.

Мы получим более удобное решение, если для каждого действия определим свою дифференцирующую функцию и через свойство diff свяжем эту функцию с символом, обозначающим действие.

# Программа с использованием модулей

Упростим запись самой дифференцирующей функции:

```
(defun dif1 (l x)
  (cond ((atom l) (cond ((eq l x) 1)
                          (t 0)))
        (t  (funcall  (get (car l) `diff) (cdr l) x))))
```

Функции дифференцирования становятся значениями свойства 'diff:

```
(setf  (get  '+  'diff)  'dif+)
(setf  (get  '*  'diff)   'dif*)
```

# Функции дифференцирования

Сами функции:

```
(defun dif*  (l x)
       (list  '+ (list  '*  (dif1 (car l) x) (cadr l))
                  (list  '*  (dif1 (cadr l) x) (car l))))
(defun dif+  (l x)
       (list  '+  (dif1 (car l) x) (dif1 (cadr l) x)))
```

Благодаря модульности можно дополнить для минуса:

```
 (setf  (get  '-  'diff)  'dif-)
 (defun dif- (l x)
       (list  '-  (dif1 (car l) x) (dif1 cadr l) x)))
```

# Ассоциативные списки

**Ассоциативный список** или просто а-список (a-list) есть основанная на списках и точечных парах структура данных, описывающая связи наборов данных obji и ключевых полей keyi, для работы с которой существуют готовые функции.

Ассоциативный список можно рассматривать как отображение множества ключей на множество соответствующих им объектов.

Структура ассоциативного списка :

((key1.obj1) (key2.obj2) … (keyN.objN))

# Создание ассоциативного списка

Функция **PAIRLIS**

формирует а-список из списка ключей keys и списка соответствующих им объектов objects.

Формат вызова :

**(pairlis keys objects a_list)**

Третий аргумент функции a_list есть формируемый а-список, в начало которого добавляются новые пары “ключ-объект”. При вызове в качестве значения a_list либо задается nil, либо предполагается, что a_list был сформирован ранее.

Пример:

(pairlis `(a b c) `(1 2 3) ())

=> ((c . 3) (b . 2) (a . 1))

# Поиск элементов в ассоциативном списке

**Функция ASSOC**

Формат вызова :

(assoc key a_list)

key - ключ

a_list – имя ассоциативного списка

В качестве значения assoc возвращает пару “ключ-объект”.

Пример:

```
(setq X (pairlis `(a b c) `(1 2 3) ()))
=>   ((c . 3) (b . 2) (a . 1))
(assoc `b X)
=>   (b . 2)
```

# Пример (совместное использование списка свойств и ассоциативного списка )

```
(setf (get `lena `salary) 90)
(setf (get `lena `age) 28)
(setf (get `lena `profes) `юрист)
(setf (get `lena `children) `(ira jura petya))

(symbol-plist `lena) ==>
    (CHILDREN (IRA JURA PETYA) PROFES
    юрист AGE 28 SALARY 90)
```

# Продолжение примера

(remprop `lena `salary)

(symbol-plist `lena) ==>

(CHILDREN (IRA JURA PETYA) PROFES юрист AGE 28)

Зарплата по штатному расписанию:

(setq штаты (pairlis `(бухгалтер юрист менеджер)
`(70 90 80) ()))  ==>

((менеджер . 80) (юрист . 90) (бухгалтер . 70))

# Продолжение примера

Какая у Лены зарплата?

```
(cdr (assoc (get `lena `profes) штаты))
   ==>  90
```

Если изменить зарплату в штатном расписании, то выражение не изменится!

# Изменение ассоциативного списка

x – имя ассоциативного списка
k – ключ
n – новое значение ключа

```
(defun new_assoc (x k n)
   (cond ((null x) nil)
            ((eq (caar x) k) (cons (cons k n) (cdr x)))
            (t (cons (car x) (new_assoc (cdr x) k n)))))
```

```
(setq штаты (new_assoc штаты `юрист 110)) ==>
    ((менеджер . 80) (юрист . 110) (бухгалтер . 70))
(cdr (assoc (get `lena `profes) штаты)) ==> 110
```