Деревья.

Лекция 13.

Специальности : 230105, 010501

n-арное дерево.

В Лиспе используется представление списка с разделением на голову и хвост. Бинарное дерево в указанной нотации представляется как список с двумя хвостами, соответствующими левому и правому поддереву :

(<инфо-часть узла> <левое поддерево> <правое поддерево>).

На основе указанного представления бинарных деревьев можно по индукции ввести аналогичное представление дерева произвольной степени ветвления :

(<инфо-часть узла> <поддерево 1> . . . <правое поддерево N>).

Таким образом, в общем случае дерево представляется как список с отсутствием ограничений на число хвостов :

```
(Информация узла1
 (Информация узла2
  (Информация узла5 nil)(Информация узла3 nil)
  . . . (Информация узлаM nil))
 (Информация узла4
  (Информация узла6 nil) . . . (Информация узлаR nil))
  . . .
 (Информация узлаK
  (Информация узла(K+1)) . . . (Информация узла(K+L)))
```

Информационное наполнение деревьев.

В задачах синтаксического и семантического анализа текстов мы рассматриваем деревья, имеющие содержательную лингвистическую интерпретацию – деревья зависимостей (в смысле грамматик деревьев А.В.Гладкого и И.А.Мельчука). В общем случае каждый узел дерева имеет достаточно сложную внутреннюю структуру информационного наполнения и не является атомом. Формализация обработки подобных деревьев основана именно на знании функциональной структуры информационного наполнения узлов : каждый элемент списочной структуры несет определенную смысловую нагрузку. Например, в случае поверхностных синтаксических структур информация каждого узла дерева описывается списком из трех элементов : первый элемент есть нормализованная формы слова, второй – метка синтаксического класса, третий – код набора грамматических характеристик. В свою очередь, нормализованную форму слова можно представить двухэлементным списком, первый элемент которого есть выделенная основа слова, второй – набор грамматических характеристик.

Основные действия над деревьями.

При построении дерева глубинной синтаксической на основе дерева синтаксического подчинения практический интерес представляют следующие основные операции над n-арным деревом :

- **Поиск элемента в дереве;**
- **Включение элемента в дерево – вырастание ветви;**
- **Стяжение ветви – склеивание узлов.**

Поиск элемента в дереве (muLISP).

```
; Поиск в дереве узла с заданными свойствами
(defun search_node (obj tree)
 ((null tree) nil)
 ((equal (caar tree) obj) T)
 (search_tree obj (cdr tree)))

; Поиск в лесу дерева, содержащего узел
; с заданными свойствами
(defun search_tree (obj forest)
 ((null forest) nil)
 ((search_node obj (car forest)) T)
 (search_tree obj (cdr forest)))
```

Примеры.

```
(search_node 30
'((20) ((15)((12) nil)((17) nil))
       ((30)((40) nil)((45) nil))
       ((70)((50) nil)((90) nil)))
)
```

дает T

```
(search_node 50
'((20) ((15)((12) )((17) ))
       ((30)((40) )((45) ))
       ((70)((50) )((90) )))
)
```

дает T

Поиск элемента в дереве (newLISP-tk).

```
; Поиск в дереве узла с заданными свойствами
(define (search_node obj tree)
 (cond
  ((null? tree) nil)
  ((= (first (first tree)) obj) true)
  (true (search_tree obj (rest tree)))))

; Поиск в лесу дерева, содержащего узел
; с заданными свойствами
(define (search_tree obj forest)
 (cond
  ((null? forest) nil)
  ((search_node obj (first forest)) true)
  (true (search_tree obj (rest forest)))))
```

Включение элемента в дерево (muLISP) : аргументы, результат и условие завершения рекурсии.

```
(defun split (parent_node new_node child_node tree flag)
; Аргументы : parent_node и child_node – узлы дерева
; tree, между которыми происходит вставка нового узла
; new_node. Для выбора одного из двух условий
; окончания рекурсии вводится флаг flag изначальной
; пустоты дерева.
; Окончание рекурсии при изначально непустом дереве
((and
    (equal flag T)
    (null tree)) nil)
; Случай изначально пустого дерева
((and
    (equal flag NIL)
    (null tree))
    (list new_node nil))
```

Случай расщепления имеющейся ветви.

```
((and (not (null child_node))
      (equal (car tree) parent_node)
      (equal (caadr tree) child_node))
  (cons
      (car tree)
      (cons
         (list new_node
              (split parent_node new_node child_node
                     (cadr tree) flag))
         (split_forest parent_node new_node child_node
                       (cddr tree) flag)
      )
  )
 )
```

Случай вырастания новой ветви.

```
((and (null child_node)
      (equal (car tree) parent_node))
 (cons
    (car tree)
    (cons (list new_node nil)
       (cons
          (split parent_node new_node
                 child_node (cadr tree) flag)
          (split_forest parent_node new_node
                        child_node (cddr tree) flag)
       )
    )
 )
)
```

Случай отсутствия узлов с заданными свойствами на текущем ярусе дерева.

```
; Случай отсутствия на рассматриваемом уровне узлов
; с заданными свойствами
 (cons
     (car tree)
     (split_forest parent_node new_node
                   child_node (cdr tree) flag)
 )
; Закрытие описания функции
)
; Обработка леса-списка дочерних узлов
(defun split_forest (parent_node new_node
                     child_node forest flag)
 ((null forest) nil)
 (cons
     (split parent_node new_node
            child_node (car forest) flag)
     (split_forest parent_node new_node
                   child_node (cdr forest) flag)
 )
)
```

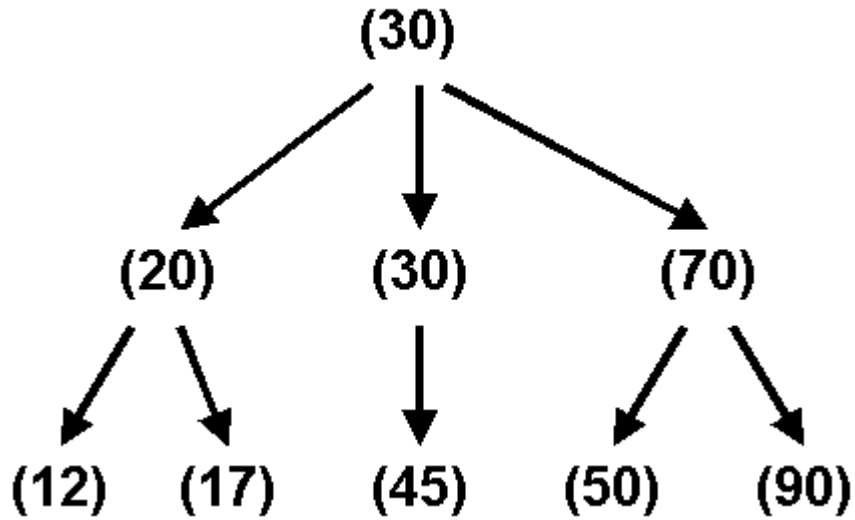
Пример : расщепление имеющейся ветви.

```
(split '(30) '(40) '(20)
       '((30)((20)((12) )((17) ))
             ((30)((45) ))
             ((70)((50) )((90) )))
       T)
```

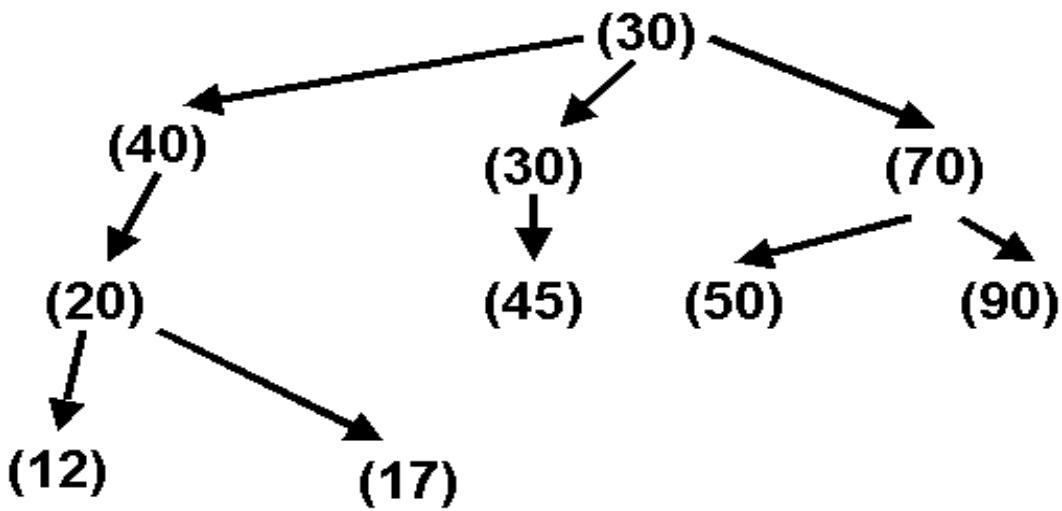
Результат :

```
((30) ((40) ((20) ((12) )((17) )
            ((30) ((45) ))
            ((70) ((50) )((90) ))
```

Исходное дерево :



Дерево-результат :



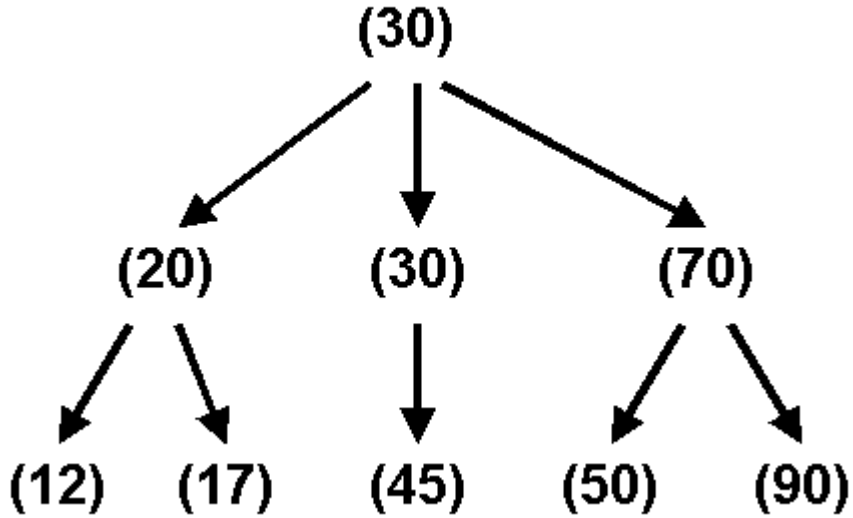
Пример : вырастание новой ветви.

```
(split '(30) '(40) nil
       '((30)((20)((12) nil)((17) nil))
             ((30)((45) nil))
             ((70)((50) nil)((90) nil)))
       T)
```

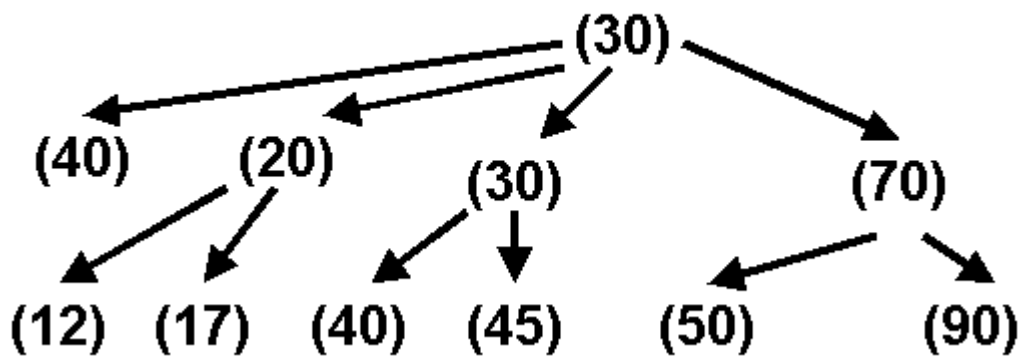
Результат :

```
       ((30) ((40) NIL)
             ((20) ((12) NIL) ((17) NIL))
             ((30) ((40) NIL) ((45) NIL))
             ((70) ((50) NIL) ((90) NIL))
       )
```

Исходное дерево :



Дерево-результат :



Включение элемента в дерево (newLISP-tk) : случай расщепления имеющейся ветви.

```
(define (split parent_node new_node child_node tree flag)
 (cond

; Окончание рекурсии при изначально непустом дереве
  ((and
     (= flag true)
     (null? tree)) '())

; Случай изначально пустого дерева
  ((and (= flag nil)(null? tree))(list new_node '()))

; Случай расщепления имеющейся ветви
  ((and (not (null? child_node))
     (= (first tree) parent_node)
     (= (first (first (rest tree))) child_node))
   (cons
     (first tree)
     (cons
       (list new_node
          (split parent_node new_node child_node (first (rest tree)) flag))
       (split_forest parent_node new_node child_node
                     (rest (rest tree)) flag)
     )
   )
  )
```

Вырастание новой ветви.

```
; Случай вырастания новой ветви

  ((and (null? child_node)
        (= (first tree) parent_node))
   (cons
      (first tree)
      (cons (list new_node '())
         (cons
            (split parent_node new_node child_node (first (rest tree)) flag)
            (split_forest parent_node new_node child_node
                          (rest (rest tree)) flag)
         ))))

; Случай отсутствия на рассматриваемом уровне
; узлов с заданными свойствами

  (true (cons
         (first tree)
         (split_forest parent_node new_node child_node (rest tree) flag)
        )
  )
; Закрытие условного предложения
 )
; Закрытие описания функции
)
```

Обработка леса-списка дочерних узлов.

```
; Обработка леса-списка дочерних узлов

(define (split_forest parent_node new_node child_node forest flag)
 (cond
  ((null? forest) '())
  (true
   (cons
    (split parent_node new_node child_node (first forest) flag)
    (split_forest parent_node new_node child_node (rest forest) flag)
   )
  )
 )
)
```

Стяжение ветви (muLISP).

```
; Функция стяжения ветви
(defun clue (parent_node child_node tree)
; Аргументы : parent_node и child_node – склеиваемые
; узлы дерева tree
; Окончание рекурсии
  ((null tree) nil)
; Случай изначально пустого дерева
; Склеиваемые узлы найдены
  ((and (equal (car tree) parent_node)
        (equal (caadr tree) child_node))
   (cons (car tree)
         (clue_in_forest parent_node child_node
                         (append (cdadr tree)(cddr tree)))
   )
  )
```

Стяжение ветви - случай отсутствия узлов с заданными свойствами на текущем ярусе дерева.

```
; Случай отсутствия на текущем ярусе
; узлов с заданными свойствами
(cons
     (car tree)
     (clue_in_forest parent_node child_node (cdr tree))
 )
)
; Обработка леса-списка дочерних узлов
(defun clue_in_forest (parent_node child_node forest)
 ((null forest) nil)
 (cons
     (clue parent_node child_node (car forest))
     (clue_in_forest parent_node child_node (cdr forest))
 )
)
```

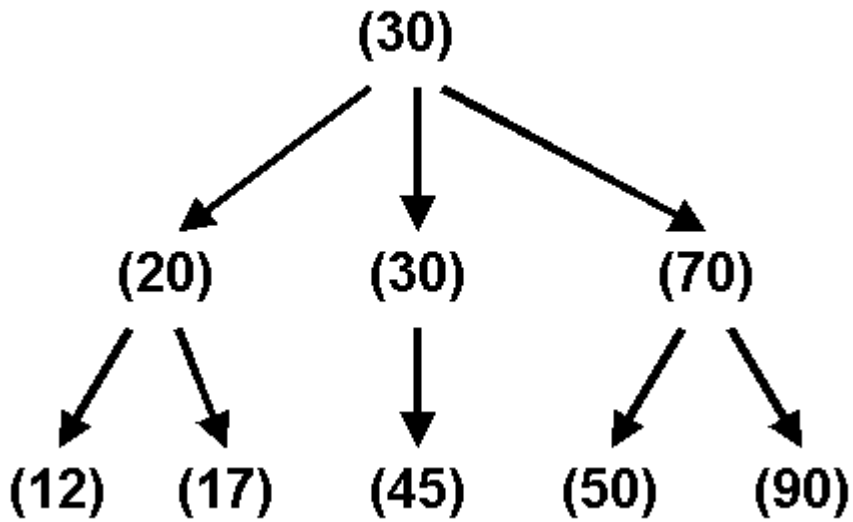
Пример : стяжение ветви.

```
(clue '(30) '(20)
      '((30)((20)((12) nil)((17) nil))
            ((30)((45) nil))
            ((70)((50) nil)((90) nil)))
      )
```

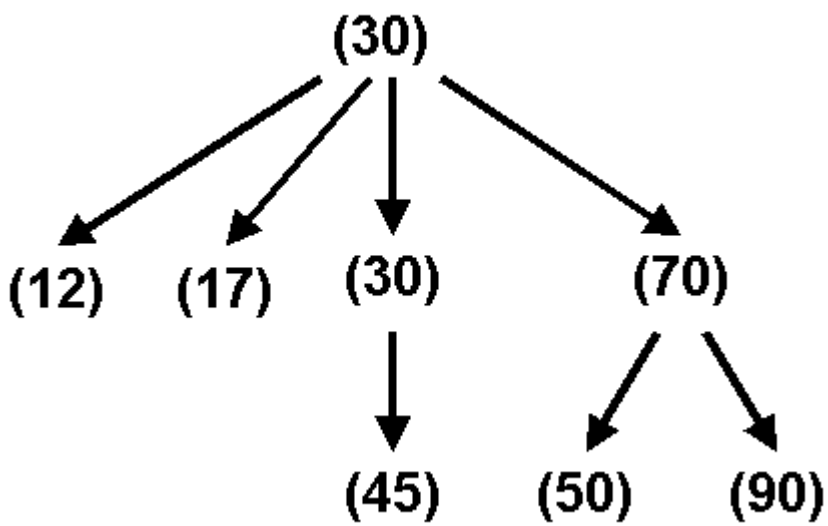
Результат :

```
      ((30)((12) NIL)
            ((17) NIL)
            ((30)((45) NIL))
            ((70)((50) NIL)((90) NIL)))
```

Исходное дерево :



Дерево-результат :



Стяжение ветви (newLISP-tk).

```
(define (clue parent_node child_node tree)
 (cond
   ((null? tree) '())
   ((and (= (first tree) parent_node)(= (first (first (rest tree))) child_node))
    (cons
       (first tree)
          (clue_in_forest parent_node child_node
                    (append (rest (first (rest tree)))(rest (rest tree)))
          )))

; Случай отсутствия на рассматриваемом уровне узлов с заданными свойствами
   (true (cons
            (first tree)
            (clue_in_forest parent_node child_node (rest tree))
          ))))

; Обработка леса-списка дочерних узлов
(define (clue_in_forest parent_node child_node forest)
 (cond
   ((null? forest) '())
   (true (cons
            (clue parent_node child_node (first forest))
            (clue_in_forest parent_node child_node (rest forest))
          ))))
```