Функции высших порядков

- Функция – полноценный объект и может создаваться и передаваться в другие функции точно таким образом, как и другие объекты!

Задачи

1. Реализовать функцию, которая инкрементирует каждый элемент списка

```
(1 2 3) => (2 3 4)
```

2. Реализовать функцию, которая возводит в квадрат каждый элемент списка

```
(1 2 3) => (1 4 9)
```

3. Реализовать функцию, которая вычисляет синус каждого элемента списка

```
(1 2 3) => (0.841 0.909 0.141)
```

4. Реализовать функцию, которая вычисляет расстояние от начала координат для каждого элемента списка, представленного парой координат

```
((3 4) (6 8)) => (5.0 10.0)
```

Функционалы

```
(defun foo(ls)
   (cond ((null ls) ls)
         (t (cons (1+ (car ls)) (foo (cdr ls)))))
   )
); (1 2 3) => (2 3 4)

(defun bar(ls)
   (cond ((null ls) ls)
         (t (cons (* (car ls) (car ls)) (bar (cdr ls)))))
   )
); (1 2 3) => (1 4 9)

(defun zet(ls)
   (cond ((null ls) ls)
         (t (cons (sin (car ls)) (zet (cdr ls)))))
   )
); (1 2 3) => (0.841 0.909 0.141)
```

Функционалы

```
(defun wvu(ls)
   (cond ((null ls) ls)
         (t (cons (sqrt (+ (* (caar ls) (caar ls))
                           (* (cadar ls) (cadar ls))))
                  (wvu (cdr ls)))))
   )
);((3 4) (6 8)) => (5.0 10.0)
```

Функциональная блокировка

- **`function`** `===` **`#'`** блокирует вычисления, образуя так называемое лексическое замыкание. Лексическое замыкание – это определение функции (лямбда-выражение) и контекст определения функции. Если блокируемая функция не содержит *свободных переменных*, то функциональная блокировка не отличается от обычной, выполняемой с помощью функции **`quote`**.

Лексическое замыкание

```
(defun x+1()(let ((x 1))
      #'(lambda()(setq x (+ x 1))))) => X+1

(setq x (x+1)) => ;связывание с x замыкания X+1
#<interpreted closure (LAMBDA NIL (SETQ X (+ X 1)))>

(setq y (x+1)) => ;связывание с y замыкания X+1
#<interpreted closure (LAMBDA NIL (SETQ X (+ X 1)))>

(funcall x) => 2 ;вызов функции

(funcall x) => 3 ;еще один вызов функции

(funcall y) => 2 ;а связи этой функции - другие!
```

Функционалы

- Функциональный объект – функция или лексическое замыкание.
- Функцию, которая в качестве аргумента получает функциональный объект, называют ***функционалом***.

Функционалы. `funcall`

```
funcall fn &rest args => result

(funcall #'+ 1 2 3 4) => 10
(funcall #'1+ 10) => 11
```

Функционалы. apply

```
apply fn &rest args+ => result

(apply #'+ 1 2 3) => ERROR
(apply #'+ 1 2 3 (4 5)) => ERROR
(apply #'+ 1 2 3 ()) => 6
(apply #'+ 1 2 3 '(4 5)) => 15
```

funcall vs apply

```
(funcall fn arg1 arg2 ...)
 ==   (apply fn arg1 arg2 ... nil)
 ==   (apply fn (list arg1 arg2 ...))
```

Функционалы

```
(defun my_mapcar(ls fn)
   (cond ((null ls) ls)
         (t (cons (funcall fn (car ls))
                  (my_mapcar (cdr ls) fn))))
   )
)

(my_mapcar '(1 2 3 4) #'1+)
(my_mapcar '(1 2 3 4) #'sin)
(my_mapcar '(1 2 3 4) #'(lambda(x)(* x x)))
(my_mapcar '((1 2) (3 4))
         #'(lambda(x)(sqrt (+ (* (car x) (car x))
                              (* (cadr x) (cadr
  x))))))
```

Функционалы. mapcar

```
mapcar fn &rest args
```

```
(mapcar #'list '(1 2 3) '(4 5 6)) =>
((1 4) (2 5) (3 6))
```

Функционалы. maplist

```
maplist fn &rest args
```

```
(maplist #'list '(1 2 3) '(4 5 6)) =>
(((1 2 3) (4 5 6)) ((2 3) (5 6)) ((3) (6)))
```

Функционалы . **mapcan и mapcon**

```
(mapcan #'list '(1 2 3) '(4 5 6)) =>
(1 4 2 5 3 6)

(mapcon #'list '(1 2 3) '(4 5 6)) =>
((1 2 3) (4 5 6) (2 3) (5 6) (3) (6))
```

Функционалы. **map**

map result-type fn &rest args

```
(map 'list #'+ #(1 2 3 4)) =>
(1 2 3 4)
(map 'list #'+ #(1 2 3) #(3 3 3)) =>
(4 5 6)
```

Другие функционалы.

remove-if

```
(remove-if #'numberp '(1 a 3 (4 5))) =>
```

```
(remove-if #'listp '(1 a 3 (4 5))) =>
```

```
(remove-if #'(lambda(x)(<= 3 x 8)) '(1
  10 3 7 8 9 5)) =>
```

Другие функционалы.
remove-if

```
(remove-if #'numberp '(1 a 3 (4 5))) =>
(A (4 5))

(remove-if #'listp '(1 a 3 (4 5))) =>
(1 A 3)

(remove-if #'(lambda(x)(<= 3 x 8)) '(1
  10 3 7 8 9 5)) =>
(1 10 9)
```

Другие функционалы.

sort

```
(sort '(1 2 3 4 3 2 1) #'>) =>

(sort '(1 2 3 4 3 2 1) #'<) =>

(sort '((1 1) (2 0) (3 2)) #'< :key
  #'cadr) =>
```

Другие функционалы.

sort

```
(sort '(1 2 3 4 3 2 1) #'>) =>
(4 3 3 2 2 1 1)

(sort '(1 2 3 4 3 2 1) #'<) =>
(1 1 2 2 3 3 4)

((sort '((1 1) (2 0) (3 2)) #'< :key
  #'cadr) => ((2 0) (1 1) (3 2))
```

Другие функционалы.

sort

```
(sort '((1 2) (3 4) (3 2))
      #'(lambda(x y)
          (< (+ (* (car x) (car x))
                (* (cadr x)(cadr x))
             )
             (+ (* (car y) (car y))
                (* (cadr y)(cadr y))
             )

          )
        )
) => ((1 2) (3 2) (3 4))
```

Другие функционалы.
set-difference

```
(set-difference '(1 2 3) '(2 3 4)) =>

(set-difference '((1 2) (3 4)) '((2 3) (3 4))
  :key #'cadr) =>

(set-difference '((1 2) (1 7)) '((0 3) (2 6))
    :test #'(lambda(x y)
           (= (apply #'+ x) (apply #'+ y))))
=>
```

Другие функционалы.
set-difference

```
(set-difference '(1 2 3) '(2 3 4)) => (1)
;L x L → L
(set-difference '((1 2) (3 4)) '((2 3) (3 4))
  :key #'cadr) => ((1 2))
;L x L x F → L
(set-difference '((1 2) (1 7)) '((0 3) (2 6))
    :test #'(lambda(x y)
            (= (apply #'+ x) (apply #'+ y))))
=> NIL
```

Ещё функционалы?

```
(defun twice(fn)
  #'(lambda(x)(funcall fn (funcall fn x)))
)
; F → F

(defun four-times(fn)
  (twice (twice fn))
)
; F → F
```

Ещё функционалы…

```
(funcall
    (twice #'(lambda(x)(+ x 1))) 0
)

(funcall
    (four-times #'(lambda(x)(+ x 1))) 0
)

(funcall
    (four-times #'(lambda(x)(* x x))) 2
)
```

Ещё функционалы?

Задача: построить функционал, строящий композицию функций!

F1 F2 F3…FN -> F1(F2(F3…(FN…)

Композиция списка функций

```
(
defun compose(funs) (
 cond ((null funs)
            #'(lambda(x)))
       ((null (cdr funs))
            #'(lambda(x)(funcall (car funs) x)))
       (t   #'(lambda(x)(funcall (car funs)
                 (funcall (compose (cdr funs)) x)))
       )
   )
)
```

Другая композиция функций

```
(
defun composer(&rest funs) (
 cond ((null funs)
         #'(lambda(x)))
       ((null (cdr funs))
         #'(lambda(x)(funcall (car funs) x)))
       (t #'(lambda(x)(funcall (car funs)
            (funcall (compose (cdr funs)) x))))
   )
)
```

Композиция

```
(funcall (compose '(sin cos tan)) 1) =>
0.013387802193205699
```

```
(funcall (composer #'sin #'asin)) 1) =>
 1.0
```

```
(funcall (composer #'sin #'asin (cos acos))) 1) =>
```
ERROR

```
(funcall (apply #'composer #'sin #'asin '(cos acos))) 1) =>
1.0
```

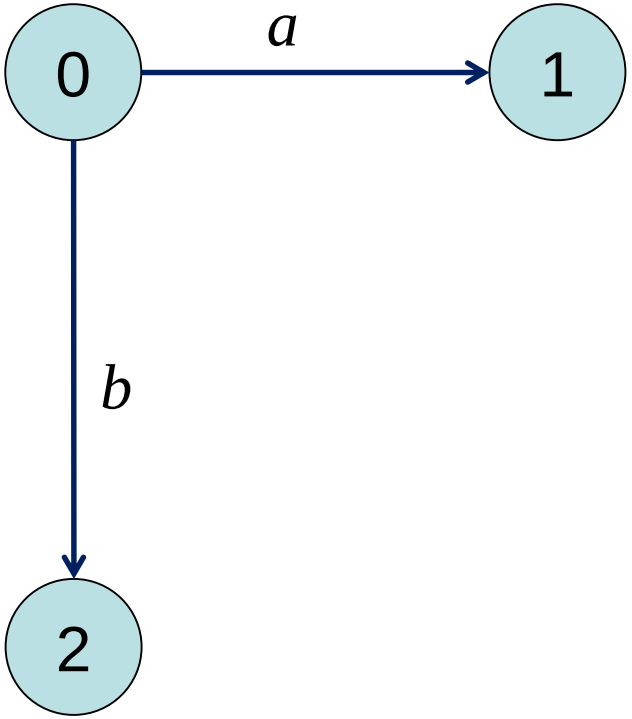
Грамматика

- *S* ::= *BA* | *AB* | *BAB*
- *A* ::= *a* | *aA*
- *B* ::= *b* | *bB*

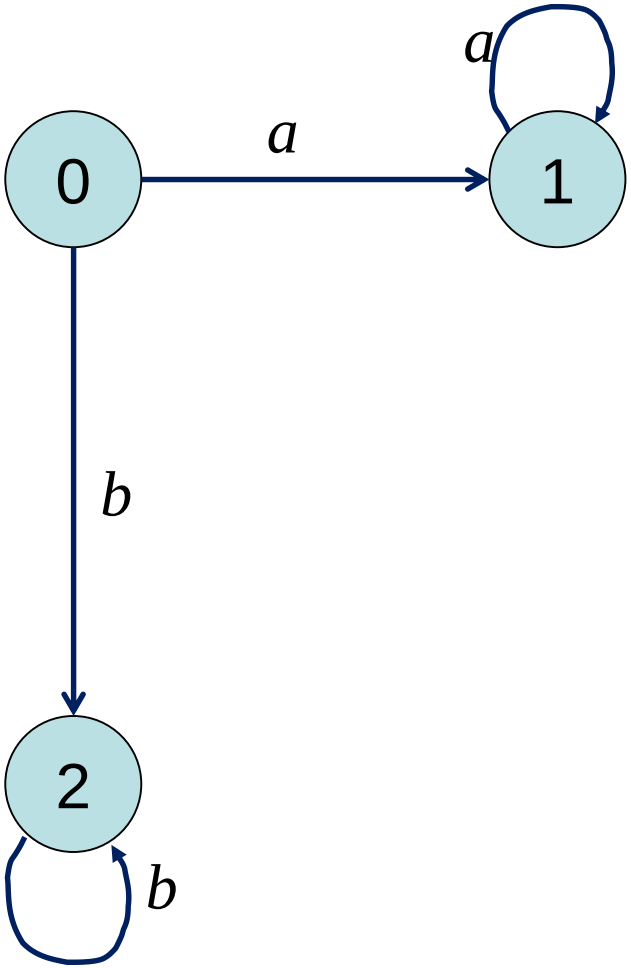
- *ba*, *ab*, *bab*, *bbba*, *bbaabb*, *baaabbb*, …

- *aa*, *bb*, *aaa*, *bbb*, *abbaabb*, *baaabbba*, …

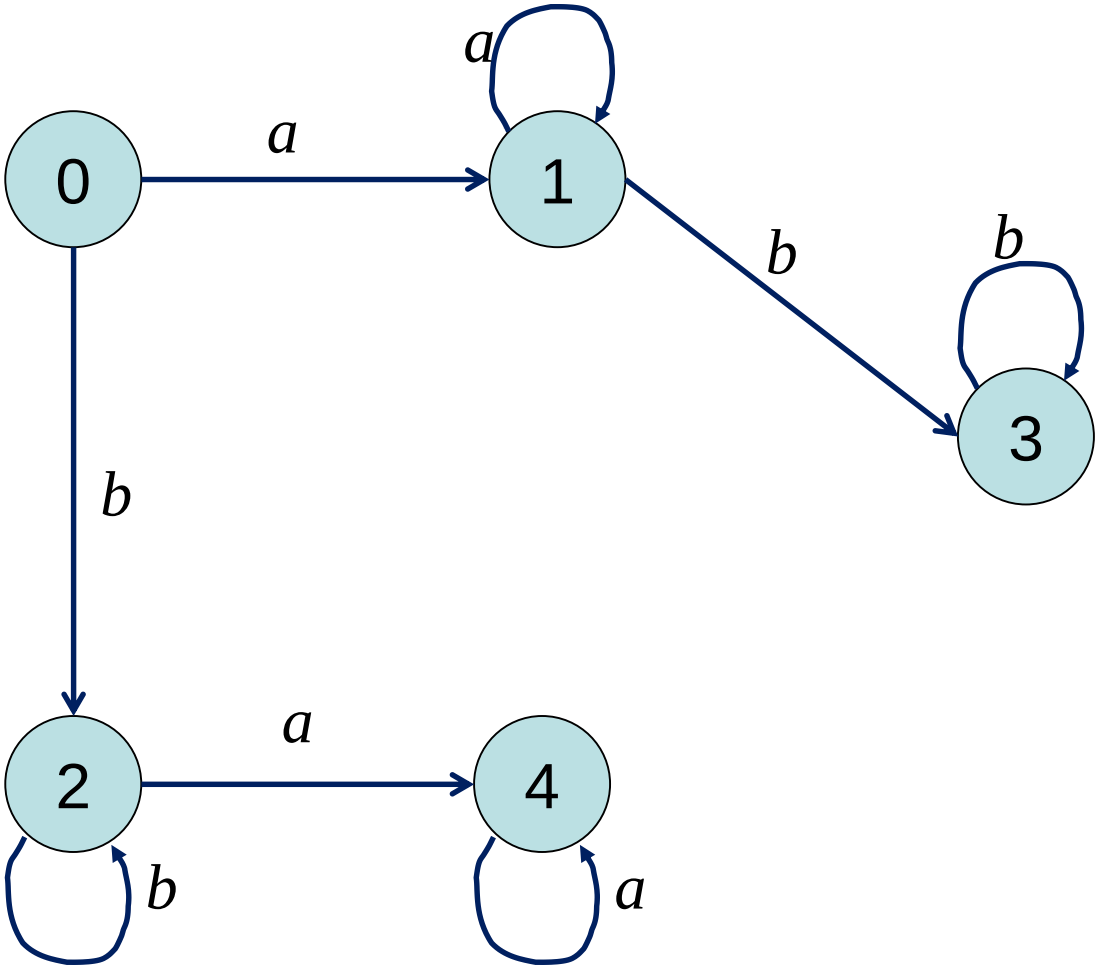
Автоматная функция



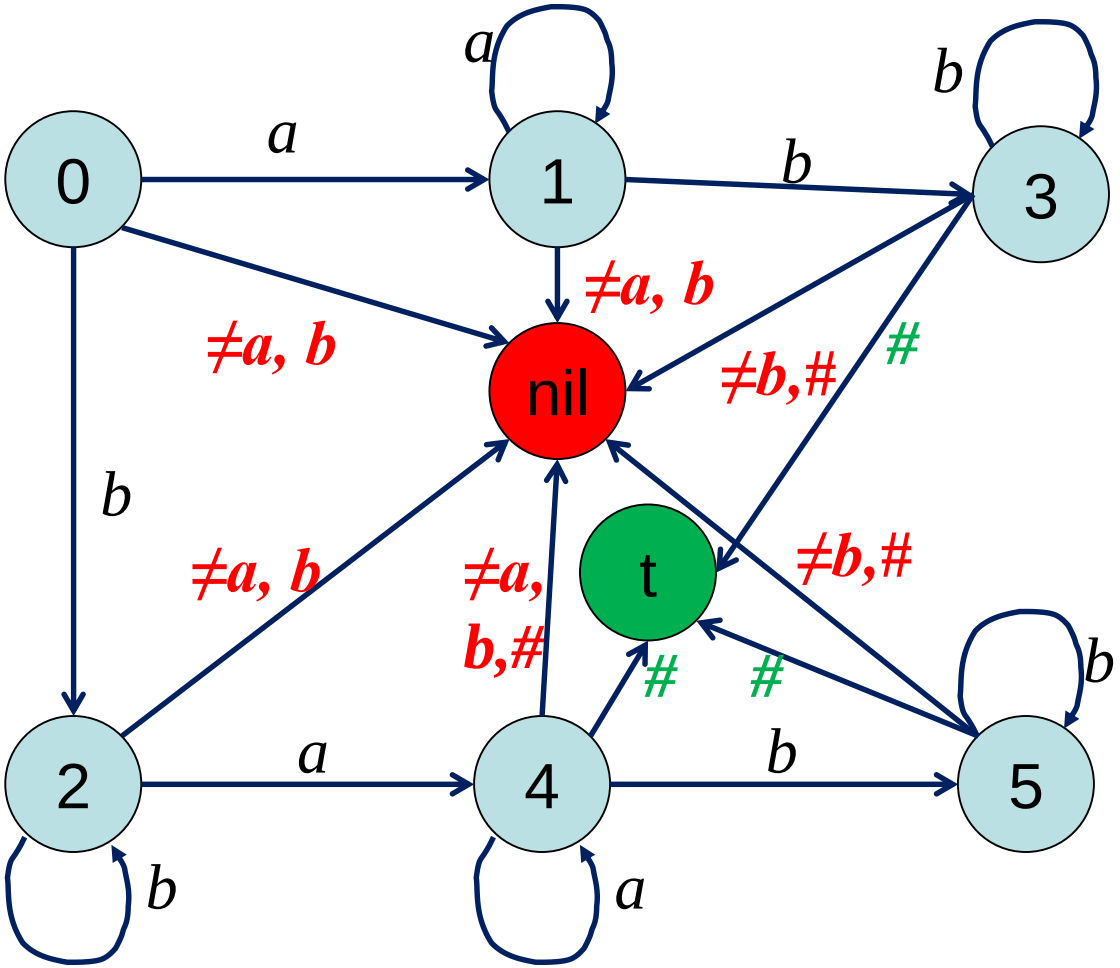
Автоматная функция



Автоматная функция



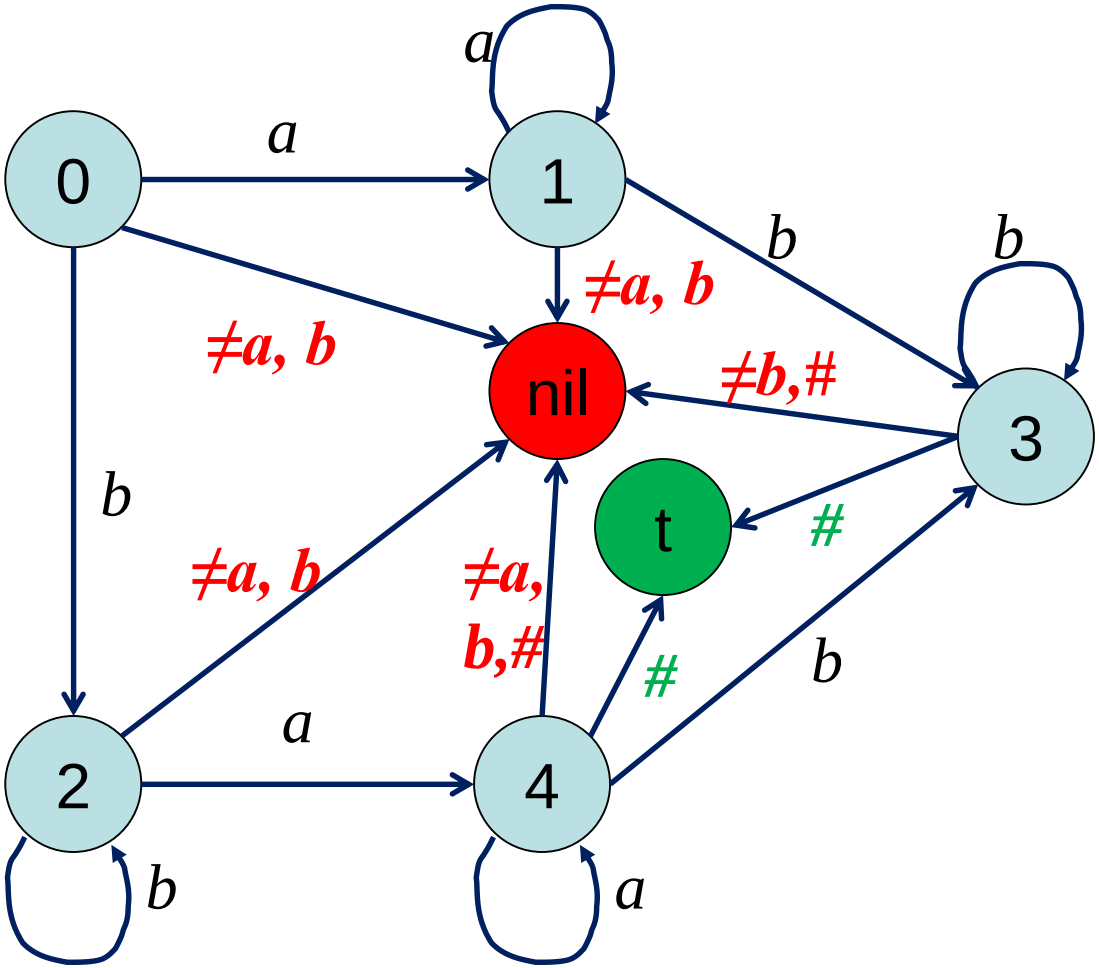
Автоматная функция



Автоматная функция

```
(defun auto(x &optional (state 0))
  (cond
    ((= state 0)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 1))
        ((eq (car x) 'b) (auto (cdr x) 2))
        (t nil)
      )
    )
    ((= state 1)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 1))
        ((eq (car x) 'b) (auto (cdr x) 3))
        (t nil)
      )
    )
    ((= state 2)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 4))
        ((eq (car x) 'b) (auto (cdr x) 2))
        (t nil)
      )
    )
    ((= state 3)
      (cond
        ((eq (car x) 'b) (auto (cdr x) 3))
        ((eq (car x) nil) t)
        (t nil)
      )
    )
    ((= state 4)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 4))
        ((eq (car x) 'b) (auto (cdr x) 5))
        ((eq (car x) nil) t)
        (t nil)
      )
    )
    ((= state 5)
      (cond
        ((eq (car x) 'b) (auto (cdr x) 5))
        ((eq (car x) nil) t)
        (t nil)
      )
    )
    (t nil)
)
```

Автоматная функция



Автоматная функция

```
(defun auto(x &optional (state 0))
  (cond
    ((= state 0)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 1))
        ((eq (car x) 'b) (auto (cdr x) 2))
        (t nil)
      )
    )
    ((= state 1)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 1))
        ((eq (car x) 'b) (auto (cdr x) 3))
        (t nil)
      )
    )
    ((= state 2)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 4))
        ((eq (car x) 'b) (auto (cdr x) 2))
        (t nil)
      )
    )
    ((= state 3)
      (cond
        ((eq (car x) 'b) (auto (cdr x) 3))
        ((eq (car x) nil) t)
        (t nil)
      )
    )
    ((= state 4)
      (cond
        ((eq (car x) 'a) (auto (cdr x) 4))
        ((eq (car x) 'b) (auto (cdr x) 3))
        ((eq (car x) nil) t)
        (t nil)
      )
    )
    (t nil)
  )
```

Функционалы в грамматическом разборе

```
(defun is-a(x) (if (eq (car x) 'a) (null (cdr x))))
(defun is-b(x) (if (eq (car x) 'b) (null (cdr x))))
;L → Bool

(defun is-alt(p q); |
  #'(lambda (x) (or (funcall p x) (funcall q x)))
)
;F x F → F
```

Функционалы в грамматическом разборе

```
(defun is-chain(p q)
  #'(lambda (x)
     (cond
       ((null x) (and (funcall p nil) (funcall q nil)))
       ((and (funcall p nil) (funcall q x)) t)
       (t (funcall (is-chain
                    #'(lambda(y)
                        (funcall p (cons (car x) y)))q
                   )
                   (cdr x)
                  )
              )
         )
     )
)
; F x F → F
```

- *XY*

Функционалы в грамматическом разборе

```
(defun is-a-gr(x)
  (funcall (is-alt #'is-a
                   (is-chain #'is-a #'is-a-gr)) x)
)
;L → Bool
```

- *A* ::= *a* | *aA*

```
(defun is-b-gr(x)
  (funcall (is-alt #'is-b
                   (is-chain #'is-b #'is-b-gr)) x)
)
;L → Bool
```

B ::= *b* | *bB*

Функционалы в грамматическом разборе

```
(defun is-syllable(x)
  (funcall
     (is-alt
        (is-chain #'is-b-gr #'is-a-gr)
        (is-alt
            (is-chain #'is-a-gr #'is-b-gr)
            (is-chain #'is-b-gr
                      (is-chain #'is-a-gr #'is-b-gr)
            )
         )
      )
     x
  )
)
```

- *S* ::= *BA* | *AB* | *BAB*