Содержание

Лекция 1. Основные понятия информатики и программирования

Данные — представление фактов, понятий, инструкций в форме, приемлемой для обмена, интерпретации или обработки человеком или с помощью автоматических средств.

Алгоритм — конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающий порядок действий исполнителя для решения некоторой задачи.

Свойства алгоритма:

1. **Дискретность** — наличие структуры, разбитие на отдельные команды, понятия, действия.
2. **Детерминированность** — для одного и того же набора данных всегда один и тот же результат.
3. **Понятность** — элементы алгоритма должны быть понятны исполнителю.
4. **Завершаемость** — алгоритм завершается за конечное число шагов.
5. **Массовость** — применимость алгоритма для некоторого класса похожих задач.
6. **Результативность** — алгоритм должен выдавать результат.

Компьютерная программа — алгоритм, записанный на некотором языке программирования.

Язык программирования — формальный язык, предназначенный для записи компьютерных программ.

Компьютер — универсальное программно-управляемое устройство для обработки информации (данных).

Парадигмы программирования — совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Основные парадигмы программирования делятся на три большие группы:

1. Императивное программирование.
2. Декларативное программирование.
3. Метапрограммирование.

Императивное программирование — способ записи программ, в котором указывается последовательность действий.

Основной признак императивной парадигмы (группы парадигм) — оператор деструктивного присваивания. Слово «деструктивное» означает, что присваивание может изменять значение, хранящееся в переменной — старое теряется безвозвратно, заменяясь новым значением.

Декларативное программирование — способ записи программ, в котором описываются взаимосвязь между данными; описывается цель, а не последовательность шагов для её достижения. Деструктивного присваивания в декларативной парадигме нет. Возможно лишь однократное присваивание значения при создании новой переменной.

Метапрограммирование — программа становится объектом управления со стороны программы — той же или другой.

В императивной группе выделяют три основные парадигмы.

1. **Структурное программирование** — каждый блок программы имеет ровно один вход и ровно один выход (кострукции вроде `goto`, `break`, `return` из середины функции, `continue` запрещены). В программе используются три основные управляющие констркции: следование (`{…}` в Си, `begin` в Scheme), ветвление (`if`/`else` в Си, `if` и `cond` в Scheme) и цикл (`while`, `for` в Си, `do` в Scheme).
2. **Процедурное программирование** — в рамках этого подхода программа рассматривается как набор подпрограмм, которые вызывают друг друга.
3. **Объектно-ориентированное программирование (ООП)** — программа написывается как набор взаимодействующих друг с другом объектов. Объект объединяет в себе даныые и поведение (код), объекты могут посылать друг другу сообщения.

Декларативная парадигма:

1. **Функциональное программирование** — алгоритм описывается как набор функций; порядок вычисления функций не существенен и на результат влиять не должен. В ленивых языках (например, Haskell) функции вызываются только когда нужен их результат.
2. **Логическое программирование** — алгоритм описывает взаимосвязь между понятиями; выполнение программы сводится к выполнению запросов. Представлено почти исключельно языком Prolog, сильно отчасти — SQL.

Пример. Требуется отсортировать последовательность чисел по возрастанию. Как это будет выглядеть в разных парадигмах.

1. Императивное программирование. Последовательность находится в массиве `numbers`. Для упорядочивания вызывается процедура `sort(array)`, меняющая содержимое своего аргумента:
   ```
   { Паскаль }
   sort(numbers);
   ```
   После вызова процедуры в массиве `numbers` будут находиться те же числа, что и ранее, но в порядке возрастания.
2. Функциональное программирование. Имеем список `numbers`, функция `sort` формирует новый список `sorted_numbers`, где будут располагаться те же числа, но по возрастанию:
   ```
   -- Хаскель
   sorted_numbers = sort numbers
   ```
   Содержимое списка `numbers` остаётся прежним.
3. Логическое программирование. Тут всё интересно. Определяем предикат
   ```
   % Пролог
   unsorted_sorted(Unsorted, Sorted) :- …
   ```
   Теперь рассмотрим обращения к предикату:
   ```
   ?- unsorted_sorted([8, 2, 5, 1], X).
     X = [1, 2, 5, 8];
   ```

```
false.
```
Получили сортированный список для несортированного
```
?- unsorted_sorted(X, [1, 2, 3]).
  X = [1, 2, 3];
  X = [1, 3, 2];
  X = [2, 1, 3];
  X = [2, 3, 1];
  X = [3, 1, 2];
  X = [3, 2, 1];
false.
```
Нашлись все перестановки сортированного списка.
```
?- unsorted_sorted(X, [1, 3, 2]).
false.
```
Для исходного списка не по возрастанию предикат не выполняется.

Парадигма метапрограммирования:

1. **Программы пишут программы:** макросы, генераторы кода, шаблонное метапрограммирование в C++.
2. **Рефлексия (интроспекция)** — программы взаимодействуют с вычислительной средой.

Подпрограмма — именованный блок кода; вызывающая программа приостанавливается, управление передаётся подпрограмме. При завершении работы подпрограммы вызывающая программа возобновляет свою работы; процедуры, функции, методы — разновидности подпрограмм.

Сопрограмма — в отличие от подпрограмм работает поочередно с вызывающей программой, при следующем вызове она возобновляет свою работу с точки остановки.

Лекция 2. Языки семейства LISP. Язык программирования Scheme

LISP (от **LIS**t **P**rocessing) — язык программирования, созданный Джоном МакКарти в 1950-1960-е годы. Породил целое семейство языков со сходным синтаксисом и идеологией: Common Lisp, Scheme, Closure и т.д.

Scheme — язык семейства LISP, созданный Гаем Стилом и Джеральдом Сассманом в 1970-е годы. Отличается простотой и минималистичным дизайном.

Диалект Scheme используется в книге Абельсона и Сассмана «Структура и интерпретация компьютерных программ» (известна под аббревиатурой SICP). Первые две главы этой книги содержат основы программирования на Scheme (но без макросов), ими можно пользоваться в качестве учебника.

В нашем курсе мы будем использовать диалект Scheme R5RS. Официальную спецификацию этого диалекта можно прочитать в PDF-ке <r5rs.pdf>.

Основные постулаты языков семейства Lisp:

1. Единство кода и данных.
2. Всё есть список.
3. Выражения являются спискам, операция указывается в первом элементы.
4. Все выражения вычисляют значения.

Грамматику списков можно следующим образом описать при помощи БНФ (формы Бэкуса-Наура):

```
<терм>   ::= <атом> | <список>
<список> ::= (<термы>)
<термы>  ::= <пусто> | <терм> <термы>
<атом>   ::= <переменная> | <число> | <символ> | <строка>
```

Иначе говоря:

- Терм — это либо атом, либо список.
- Список — последовательность термов (возможно пустая) в круглых скобках.
- Атом — имя переменной, число, символ или строка. Подробнее разновидности атомов мы изучим позже.

В выражениях языка Scheme после открывающей круглой скобки указывается операция. Операцией может быть либо вызов функции, либо так называемая **особая форма.** В случае вызова функции первым термом после скобок является имя функции или выражение, порождающее функцию. В случае особой формы после открывающей круглой скобки располагается **ключевое слово.**

(в процессе подготовки)

Переменные в Scheme определяются при помощи конструкции `define`. Её синтаксис:

```
(define ‹имя-переменной› ‹выражение›)
```

```
(define (area r)
(* r r 3.1415926))
```

Управляющие конструкции:

1. Объявление глобальных переменных

```
(define <var> <val>)
(define (f <args>) <expr>)
```

То же самое:

```
(define f
  (lambda (<args>)
    <expr>))
```

2. Ветвление

```
(if <условие>
    <если истина, по умолчанию #t>
    <если ложь, по умолчанию #f>)
```

```
(cond
  (<условие 1> <выражение>)
  (<условие 2> <выражение>)
  (else <выражение>)
```

```
(and/or <условие 1>
        <условие 2>
        ...
        <условие n>)

(and <усл>
     <выраж>);корректная запись
(if <усл>
    <выраж>
    #f);некорректная запись

(or <усл> <выраж>);корректно
(if <усл>
    #t
    <выраж);некорректно

(and (not <усл>) <выраж>);корректно
(if <усл>
    #f
    <выраж>);некорректно

(or (not <усл>) <выраж>);корректно
(if <усл>
    <выраж>
    #t);некорректно
```

Лекция 3. Функции высшего порядка

Значения языка Scheme:

- Числа: 1, 1.0, 6.022e23, 1/3...
- Строки: ”Scheme”
- Логический тип: #t, #f
- Литерный (character) тип: #\a #\newline ...
- Символьный (symbol) тип: ’x, ’sin...
- ...

- **Процедурный тип**

  (lambda (аргументы) выражение)

Конструкция lambda создаёт безымянную процедуру. Эту процедуру можно вызвать:

```
((lambda (x y) (+ x y)) 10 13)
;         ^--- формальные параметры
;фактические параметры --^
```

При вызове процедуры создаются новые переменные, соответствующие формальным параметрам и они связываются с фактическими параметрами.

```
(define f
  (lambda (x y) (+ x y)))
(f 10 13)
```

Синтаксический сахар:

```
(define f (lambda (парам) выраж))
```

эквивалентно

```
(define (f парам) выраж)
```

Передача процедуры как параметра:

```
(define (g f)
  (f 10 13))

(g (lambda (x y) (+ x y)))
(g +)
```

Возврат процедуры из процедуры

```
(define (select n)
  (if (> n 0)
      (lambda (x y) (+ x y))
      (lambda (x y) (- x y))))

((select +1) 10 13)
((select -1) 100 50)
```

Управляющие конструкции языка Scheme

Конструкции let, let* и letrec

```
(let ((var1 expr1)
      (var2 expr2)
      ...
      (varN exprN))
  выражение)
```

В теле let-выражения можно использовать переменные var1...varN. Выражения expr1...exprN могут вычисляться в произвольном порядке - порядок их вычисления не определён.

НО! Внутри expr1...exprN нельзя использовать var1...varN.

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


(let* ((var1 expr1)
       (var2 expr2)
       ...
       (varN exprN))
  body)
```

Переменную varK можно использовать не только в теле let*, но и в exprM, где M > K.

Выражения expr1...exprN вычисляются *последовательно.*

```
 (letrec ((var1 expr1)
          (var2 expr2)
          ...
          (varN exprN))
  body)
```

Внутри любого exprK можно использовать любую переменную из var1...varN.

Все эти конструкции являются синтаксическим сахаром. Для примера:

```
(let ((var1 expr1)
      (var2 expr2)
      ...
      (varN exprN))
  выражение)
```

эквивалентна

```
((lambda (var1 ... varN)
   выражение)
 expr1 ... exprN)
```

”let с рекурсией”

```
(let proc-name ((var1 expr1)
                (var2 expr2)
                ...
                (varN exprN))
  body)
```

Внутри тела let-выражения можно вызывать процедуру proc-name, передавая ей N параметров.

Это выражение эквивалентно

```
(letrec ((proc-name
          (lambda (var1 ... varN)
            body)))
  (proc-name expr1 ... exprN))
```

Рекурсия, итерация и хвостовая рекурсия

N! = 123*...*(N-1)*N

0! = 1 N! = N * (N-1)!

Рекурсия - делим задачу на меньшие подзадачи, подобные исходной.

Итерация - задача делится на некоторое количество одинаковых подзадач, одинаковых шагов, приближающих к цели.

Как итерацию выразить через рекурсию?

Итерация: пока цель не достигнута, повторять шаг вычисления.

Рекурсия:

- Цель достигнута?
  - Да - прекратить вычисления, вернуть результат.
  - Нет - выполнить один шаг вычисления, выполнить рекурсивный вызов.

Факториал в терминах итерации:

```
int fact(int N) {
  int res = 1;
  int i = 1;
  while (i <= N) {
    res = res * i;
    i = i + 1;
  }
  return res;
}
```

- Для цикла заводим вспомогательную процедуру.
- Переменные цикла становятся параметрами процедуры.
- Тело цикла превращается в рекурсивный вызов.
- Инициализация переменных цикла становится вызовом рекурсивной процедуры.

  (define (fact N) (define (loop i res) (if (<= i N) (loop (+ i 1) (* res i)) res)) (loop 1 1))

  (define (fact N) (let loop ((i 1) (res 1)) (if (<= i N) (loop (+ i 1) (* res i)) res)))

Хвостовая рекурсия

Хвостовой вызов - вызов, который является последним, результат этого вызова становится результатом работы функции.

```
(define (f x y z)
  (if (a)
      (b x (c y))
      (d (if (e)
             (g)
             (h)))))
```

(Вызовы b и d - хвостовые)

В языке Scheme заложена оптимизация хвостового вызова, т.н. оптимизация хвостовой рекурсии. Фрейм стека (см. лекцию про продолжения) вызывающей процедуры замещается фреймом стека вызываемой процедуры.

Если хвостовой вызов является рекурсивным, фреймы стека не накапливаются.

Хвостовая рекурсия в языке Scheme эквивалента итерации по вычислительным затратам.

Рекурсивный факториал:

```
(define (fact N)
  (if (> N 0)
      (* (fact (- N 1)) N)
      1))
```

Итеративный факториал:

```
(define (fact N)
  (define (loop i res)
    (if (<= i N)
        (loop (+ i 1)
              (* res i))
        res))
  (loop 1 1))
```

Оптимизация хвостовой рекурсии изнутри:

```
int loop(int N, int i, int res) {
  if (i <= N) {
    loop(N, i + 1, res * i);
  } else {
    return res;
  }
}

int loop(int N, int i, int res) {
LOOP:
  if (i <= N) {
```

```
    res = res * i;
    i = i + 1;
    goto LOOP;
  } else {
    return res;
  }
}
```

Лекция 4. Списки

LISP — List processing, обработка списков. Список — основная структура данных языка Scheme.

(1 2 3 4) — список из четырёх чисел.

Создание списка

```
(list <элементы>)

(list 1 2 3 4)    →    (1 2 3 4)
(list)            →    ()
```

Операции над списками:

cons — конструирование

```
(cons <голова> <хвост>) → <список>
```

Создаёт новый список из некоторого значения («головы») и другого списка («хвоста»). Первым элементом нового списка будет «голова», последующими — хвост.

```
(cons 1 (list 2 3 4))   →   (1 2 3 4)
```

car, cdr, null?

```
(car <список>) → <голова списка>
(cdr <список>) → <хвост списка>
(null? <список>) → <bool>
```

Это селекторы, запрашивают голову и хвост списка, список должен быть непустым.

```
(car (list 1 2 3 4)) → 1
(cdr (list 1 2 3 4)) → (2 3 4)

(null? (list 1 2 3 4)) → #f
(null? (list)) → #t
```

Пустой список, запись списка через цитирование

```
'() — пустой список
```

Запись списка при помощи цитирования:

```
'(1 (2 3 4) 5 6)
```

Вложенные списки:

```
(list (list 1 2 3) (list 4 5 6)) → ((1 2 3) (4 5 6))
'((1 2 3) (4 5 6))               → ((1 2 3) (4 5 6))
```

Список можно связать с переменной:

```
(define L '(1 2 3))

(define x 1)

(define (f y)
  (list x y))

(define (f y)
  (cons x (cons y '())))

'(x y)
```

Встроенная функция `length`

```
(length '(1 1 2 1)) → 4
```

Встроенная функция `append` — конкатенация списков:

```
(append '(1 2 3) '(4 5 6)) → (1 2 3 4 5 6)

(append '(1 2) '(3 4) '(5 6)) → (1 2 3 4 5 6)
```

Списков не существует — cons-ячейки или пары

Объект, который строится функцией `cons` — т.н. cons-ячейка или пара. Аргументами функции `cons` могут быть любые объекты.

Правильный список — это или пустой список, или cons-пара, вторым элементом которой является правильный список.

```
(cons 1 2)            → (1 . 2)   ;; неправильный список
(cons 1 (cons 2 '())) → (1 2)     ;; правильный список


(cons 1 (cons 2 (cons 3 4))) → (1 2 3 . 4)  ;; тоже неправильный список
```

Пример. Как могла бы быть определена встроенная функция `length`:

```
(define (length xs)
  (if (null? xs)
      0
      (+ 1 (length (cdr xs)))))

(define (length xs)
  (define (loop len xs)
```

```
    (if (null? xs)
        len
        (loop (+ len 1) (cdr xs))))
  (loop 0 xs))
```

Функция `pair?` возвращает истину, если аргумент — cons-ячейка.

Встроенная функция `map`

Используется для того, чтобы единообразно преобразовать все элементы списка, принимает процедуру и исходный список, строит новый список той же длины, что и исходный, каждый элемент этого списка является результатом применения процедуры к элементу исходного списка.

```
(define (square x) (* x x))
(map square '(1 2 3 4 5))  →  '(1 4 9 16 25)
```

Расширенный вариант использования `map`:

```
(map
  (lambda (x y) (+ (* 2 x) (* 3 y)))
  '(1 2 1 2)
  '(2 3 2 3 2)
) →
  '(8 13 8 13)
```

Функцию `map` («простой вариант») можно описать как:

```
(define (map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs)) (map f (cdr xs)))))
```

Процедуры с переменным числом параметров

Безымянная процедура, принимающая произвольное число аргументов:

```
(lambda xs …)
```

`xs` — список аргументов

```
((lambda xs xs) 1 2 3 4) → (1 2 3 4)
```

Безымянная процедура, принимающая n+ аргументов:

```
(lambda (a b c . xs) …)
```

```
((lambda (a b c . xs)
   (list (+ a b c) xs))
 1 2 3 4 5) →
   (6 (4 5))
```

Именованная процедура:

```
(define (f <фиксированные параметры> . <список параметров>)
  …)

(define (f a b c . xs)
  (list (+ a b c) xs))

(f 1 2 3 4 5) → (6 (4 5))

(define (f . xs)
  (list xs xs))

(f 1 2 3) → ((1 2 3) (1 2 3))
```

Функцию `list` можно описать так:

```
(define (list . xs) xs)
```

Пример:

```
((lambda x x)) → ()
```

Вычислительная сложность

Вычислительная сложность — асимптотическая оценка времени работы программы. Асимптотическая, значит, нас интересует не конкретное время, а поведение.

`T(<данные>)` — функция, возвращающая точное значение времени работы программы на конкретных входных данных.

Асимптотическая оценка `O(f(<данные>))` показывает, что функция `T(•)` при росте входных данных ведёт себя как функция `f(•)` с точностью до некоторого постоянного сомножителя.

Т.е. существует такое `k`, что

```
T(data) <= k×f(data)
```

при росте аргумента `data`.

Оценку вычислительной сложности для некоторого алгоритма и некоторого абстрактного вычислителя обычно оценивают в числе элементарных команд этого абстрактного вычислителя.

Для Scheme элементарными операциями считаются вызов функции, `cons`, `car`, `cdr`, получение значения переменной, создание процедуры (`lambda`), объявление глобальной переменной (`define`), присваивание переменной (`set!`), арифметические действия с фиксированным числом операндов (не свёртка!), call/cc (создание и переход на продолжение), `delay`, `force`, `null?` (и другие встроенные предикаты), `if`, `cond`.

Встроенные функции могут иметь разную сложность!

Например,

- `(map f xs)` — `O(len(xs)×T(f))`, где `T(f)` — среднее время работы `(f x)`.
- `(length xs)` — `O(len(xs))`.
- `(append xs ys)` — `O(len(xs))`.

.

```
(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

Замыкания, области видимости и захват переменных

```
(define (f x)
  (lambda (y) (+ x y))

(define f1 (f 1))
(define f7 (f 7))

(f1 10) → 11
(f7 10) → 17
```

Проверка на равенство

- `eqv?` — атомы сравнивает по значению, сложные типы данных (списки, векторы, lambda) — по ссылке.
- `eq?` — может и атомы сравнивать по ссылке.
- `equal?` — сравнивает аргументы по значению.
- `=` — равенство чисел. Может сравнивать числа разных типов.
- Функции сравнения отдельных типов вроде `string=?`...

Функция `equal?` медленная, т.к. сравнивает аргументы по значению, в частности, для списков сравнивает их содержимое. Но она наиболее предсказуемая.

Функции `eq?` и `eqv?` работают быстро, но могут давать неожиданные результаты.

```
(define x …)
(define y x)

(eq? x y)     → #t
(eqv? x y)    → #t
(equal? x y)  → #t
```

Лекция 5. Императивное программирование на языке Scheme

Сведения ко второй части домашнего задания

Тип данных `vector`

Списки — основные структуры данных в языках семейства Lisp. В Scheme они не примитивный тип, а надстройка над cons-ячейками.

Списки по своей сути однонаправленны — можем их читать слева-направо при помощи `car` и `cdr` и наращивать справа-налево при помощи `cons`.

Недостаток списков — это производительность при доступе по номеру.

Есть встроенная функция `(list-ref xs n)`, возвращающая n-й элемент:

```
(define xs '(a b c d))
(list-ref xs 2)                    →  c
```

(элементы нумеруются с нуля)

Но сложность той же `list-ref` — `O(n)`, где `n` — номер элемента.

Для преодоления этого недостатка в Scheme есть встроенный тип данных `vector`, допускающий произвольный доступ к элементам для чтения и записи за константное время.

Нужно помнить, что `vector` — ссылочный тип, в том смысле, что если мы в две переменные положим один и тот же вектор, то изменения вектора через одну переменную будут видны через другую.

```
(define v #(1 2 3 4))
(define w v)

(vector-set! v 2 77)
w                                  → #(1 2 77 4)
```

Создаётся вектор при помощи литерала `#(…)`, при этом его содержимое неявно цитируется, также как и при `'(…)`.

```
(define a 100)
(define v #(1 2 3 a 4 5 6))
a                                  → #(1 2 3 a 4 5 6)
```

Т.е. `a` внутри вектора будет не переменной, а процитированным символом.

Функция `make-vector` создаёт новый вектор:

```
(make-vector size)
(make-vector size init)
```

где `size` — размер вектора, а `init` начальное значение элементов. Т.к. вектора используются чаще для расчётов, инициализация по умолчанию — `0`.

```
(make-vector 10)                    → #(0 0 0 0 0 0 0 0 0 0)
(make-vector 10 'a)                 → #(a a a a a a a a a a)
```

Предикат типа — `vector?`.

```
(vector? #(1 2 3))                  → #t
(vector? '(1 2 3))                  → #f
```

Обращения к элементам вектора:

```
(vector-ref v n)                    → n-й элемент вектора (начиная с 0)
(vector-set! v n x)                 ; присваивает n-му элементу
                                    ; новое значение x

(define v (make-vector 5))
v                                   → #(0 0 0 0 0)
(vector-ref v 2)                    → 0
(vector-set! v 2 100)
v                                   → #(0 0 100 0 0)
(vector-ref v 2)                    → 100
```

Что будет?

```
(define m (make-vector 4 (make-vector 4)))
```

На первый взгляд, мы создаём квадратную матрицу. На самом деле, мы создаём два вектора, все элементы одного вектора содержат `0`, все элементы второго — ссылку на первый.

```
m              → #(#(0 0 0 0) #(0 0 0 0) #(0 0 0 0) #(0 0 0 0))
(vector-set! (vector-ref m 0) 0 1)
m              → #(#(1 0 0 0) #(1 0 0 0) #(1 0 0 0) #(1 0 0 0))
```

Вектор можно преобразовать в список и наоборот

```
(vector->list #(a b c))             → (a b c)
(list->vector '(a b c))             → #(a b c)
```

Строки

Тип данных `string` хранит в себе последовательность литер (characters). Литерал для строки — текст, записанный внутри двойных кавычек: `"Hello!"`.

Внутри строк допустимы стандартные escape-последовательности языка Си:

```
"one line\ntwo lines"               ; строка со знаком перевода строки
"I say: \"Hello!\""                 ; заэкранированная кавычка
```

Операции над строками:

```
(make-string 10 #\a)               → "aaaaaaaaaa"
(string-ref "abcdef" 3)            → #\d               ; счёт тоже с 0
(string->list "abcdef")            → (#\a #\b #\c #\d #\e #\f)
(list->string '(#\H #\e #\l #\l #\o)) → "Hello"
(string? "hello")                  → #t
(string? 'hello)                   → #f
(string-append "штука" "турка")    → "штукатурка"
```

Литеры задаются так:

```
#\x            ; буква «икс»
#\7            ; цифра «семь»
#\(            ; литера «круглая скобка»
#\space        ; пробел
#\newline      ; \n в Си
#\return       ; \r в Си
#\tab          ; \t в Си
#\             ; хотели пробел, но получили ошибку синтаксиса
```

В ДЗ потребуется функция `(whitespace? char)`, возвращающая истину, если литера — пробельная (пробел, табуляция, новая строка, возврат каретки).

```
(whitespace? #\space)              → #t
(whitespace? #\z)                  → #f
(whitespace? (string-ref "a b" 1)) → #t
```

Выбор подстрок `(substring …)` изучить самостоятельно.

Императивное программирование на языке Scheme

До этого мы рассматривали декларативное программирование, в котором у нас не было:

- присваиваний,
- циклов,
- процедур с побочными эффектами,
- недетерминированных процедур — процедур, результат которых определяется не только значениями аргументов.

В Scheme есть средства не только декларативного программирования, но и императивного. Т.е. можно и присваивать переменным новые значения, и пользоваться процедурами, которые вызываются не только ради возвращаемого значения, но и дополнительных действий (побочного эффекта).

В Scheme не определён порядок вычисления аргументов в вызове процедуры. Но в императивном программировании порядок вычисления (а вернее, выполнения) операций существенен. Поэтому в первую очередь нам нужно средство упорядочивания выполнения операций.

1. `begin`

Если мы имеем вызов вида

```
(f (g …))
```

то в Scheme гарантируется, что сначала вычислится `(g …)`, а потом `(f …)`. (В Haskell не гарантируется.)

Но если мы имеем вызов вида

```
(f (g …) (h …))
```

то, что выполнится раньше — `g` или `h` — зависит от реализации. Разные реализации Scheme могут вычислять аргументы справа налево или слева направо.

Но если нужно вывести на печать несколько значений, то порядок вызова будет существенен: функции должны вызваться в правильном порядке. Можно извратиться, например, конструкцией `let*`:

```
(let* ((x (display "Hello, "))
       (y (display "World!")))
  #f)
```

Но это избыточно, т.к. в Scheme уже есть особая форма `(begin …)`, гарантирующая порядок вычисления:

```
(begin
  (display "Hello, ")
  (display "World!"))
```

(На самом деле `begin` может быть библиотечным макросом, который неявно трансформируется в тот же `let*`).

`begin` выполняет действия в том порядке, в котором они записаны.

Результатом `begin`'а является результат последнего действия.

```
(begin (* 7 3) (+ 6 4))            → 10
```

Результат умножения будет отброшен, умножение тут вообще бессмысленно.

Неявный `begin` Некоторые конструкции Scheme позволяют записывать несколько действий подряд, например `lambda`, `define`, определяющий процедуру, `cond`, `let`, `let*`, `letrec`.

```
; синтаксический сахар
(lambda (x y)
  (display x)
  (display y))

(define (f x y)
  (display x)
  (display y)
```

```
; эквивалентен
(lambda (x y)
   (begin
      (display x)
      (display y)))

(define (f x y)
   (begin
      (display x)
```

```
  (+ x y))                                  (display y)
                                            (+ x y)))

(let ((x 100)                          (let ((x 100)
      (y 200))                               (y 200))
  (display x)                            (begin
  (display y)                              (display x)
  (* x y))                                 (display y)
                                           (* x y)))

(cond ((> x y)  (display x) (- x y))
      …)

(cond ((> x y)  (begin (display x) (- x y)))
     …)
```

2. Присваивания, `set!`

Синтаксис:

```
(set! ‹имя переменной› ‹выражение›)
```

Переменной может быть как имя, объявленное при помощи `define`, так и параметр процедуры или имя, определённое `let`, `let*`, `letrec`.

Например

```
(define counter 0)
counter                              → 0
(set! counter 100)
counter                              → 100
(set! counter 0)

(define (next)
  (set! counter (+ counter 1))
  counter)

(next)                               → 1
(next)                               → 2
(next)                               → 3
counter                              → 3
(set! counter 7)
(next)                               → 8
```

Статические переменные в Scheme В языке Си есть понятие **статическая переменная** — глобальная переменная, видимость которой ограничена одной функцией. Объявляется она с использованием ключевого слова `static`:

```
void f() {
  int x = 0;
```

```
  static int y = 0;

  x = x + 1;
  y = y + 1;

  printf("x = %d\n", x);
  printf("y = %d\n", y);
}

int main(int argc, char **argv) {
  f();
  f();
  f();

  return 0;
}
```

Напечатается:

```
x = 1
y = 1
x = 1
y = 2
x = 1
y = 3
```

Значение статической переменной сохраняется между вызовами функции (сравните выше поведение `x` и `y`).

В языке Scheme статических переменных нет, но есть идиома (приём программирования), позволяющая их имитировать: т.е. создавать переменные, видимые только внутри функции, но при этом сохраняющие значение между вызовами.

Вспомним, что конструкция

```
(define (f x y)
  ‹тело процедуры›)
```

есть синтаксический сахар для

```
(define f
  (lambda (x y)
    ‹тело процедуры›))
```

Что будет, если мы эту лямбду обернём в let-конструкцию?

```
(define f
  (let (‹объявления каких-то переменных›)
    (lambda (x y)
      ‹тело процедуры›)))
```

Let-конструкция свяжет с переменными значения и вернёт лямбду как свой результат. Переменная `f` будет связана с лямбдой. Что же будет с переменными?

Эти переменные будут видимы внутри лямбды, не будут видимы вне конструкции let, их значения будут сохраняться между вызовами.

Эти переменные будут вести себя как статические переменные в Си.

Перепишем пример с `(next)`, чтобы переменная `counter` была статической.

```
(define next
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))

(next)                              → 1
(next)                              → 2
(next)                              → 3
```

3. Цикл do

Цикл `do` используется редко, выглядит он вот так:

```
(do ((‹перем› ‹нач› ‹модиф [необ.]›)       ; почти как в let
     ...
     (‹перем› ‹нач› ‹модиф [необ.]›))
   (‹усл. выраж.› ‹возврат [необ.]›)
   ‹выраж›
   ...
   ‹выраж›)
```

Пример:

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
   ((= i 5) vec)
 (vector-set! vec i i))             → #(0 1 2 3 4)
```

Две переменные цикла: `vec` и `i`. `vec` присваивается новый вектор, `i` — `0`, `vec` не меняется (модификация переменной отсутствует), `i` увеличиваетя на `1`, условие выхода — `(= i 5)`, возвращаемое значение — `vec`. В теле цикла в `i`-ю позицию вектора присваивается число `i`.

4. Изменяемые структуры данных

В Scheme некоторые значения в памяти можно менять. Прежде всего это вектор — его элементам можно присваивать новые значения при помощи `vector-set!`. Но можно менять и cons-ячейки.

Есть такие функции:

```
(set-car! ‹cons-ячейка› ‹значение›)
(set-cdr! ‹cons-ячейка› ‹значение›)
```

Например, можно создать кольцевой список:

```
(define loop-xs '(a b c))
(set-cdr (cdr loop-xs) loop-xs)
```

Получится кольцевой список вида `(a b a b a b …)`.

```
(list-ref loop-xs 0)              → a
(list-ref loop-xs 37)             → b
(length loop-xs)                  → зависло
```

Вообще, рекомендуется работать со списками как с неизменяемыми данными. Если содержимое списков менять на месте при помощи `set-car!` или `set-cdr!`, то можно сильно запутать программу, поскольку разные списки могут разделять общий хвост.

```
(define xs '(a b c d))
(define ys (append '(1 2 3) xs))
(define zs (cons 'x xs))
(define us (append xs xs))

(set-car! xs 'hello)

ys                                → (1 2 3 hello b c d)
zs                                → (x hello b c d)
us                                → (a b c d hello b c d)
```

Наиболее ожидаемым было изменение `zs`. Наиболее неожиданным — `us`.

```
(define (my-append xs ys)
  (if (null xs)
      ys
      (cons (car xs) (my-append (cdr xs) ys))))
```

Лекция 6a. Понятие свёртки

Свёртка — объединение нескольких значений одной операцией. Примеры: вычислить сумму нескольких чисел, произведение нескольких чисел и т.д.

`a • b • c • … • k`

Здесь знаком • обозначена некоторая двуместная операция.

Свёртка может быть *правой* и *левой*.

Правая свёртка:

`a • (b • (c • (… • k)…))`

Левая свёртка:

```
((…(a • b) • c) … • k)
```

Иногда для свёртки может быть определён некоторый нейтральный элемент z:

```
((a • b) • c) = (((z • a) • b) • c)
(a • (b • c)) = (a • (b • (c • z)))
```

В Scheme некоторые операции обладают свойством свёртки, такие функции могут принимать произвольное количество параметров.

1. Сложение: `(+ 1 2 3 4)` → `10`
2. Умножение: `(* 1 2 3 4)` → `24`
3. Вычитание: `(- 10 5 3)` → `2`
4. Деление: `(/ 120 6 5)` → `4`
5. Функции `min` и `max`: `(min 3 8 2 5)` → `2`, `(max 3 8 2 5)` → `8`.
6. Конкатенация списков: `(append '(a b) '(c d e) '(f g))` → `(a b c d e f g)`.
7. Конкатенация строк: `(string-append "ab" "cde" "fg")` → `"abcdefg"`.

Особые формы `(and …)` и `(or …)` тоже обладают обладают свойством свёртки, не смотря на то, что это не функции.

Для свёрток определена такая аксиома, что если `op` — свёрточная операция, то

```
(op x) ≡ x
```

кроме `-` и `/` — `(- x)` меняет знак числа, `(/ x)` — вычисляет обратное значение.

Для ряда свёрточных операций существует нейтральный элемент. Он возвращается при вызове операции без параметров:

```
(+)            → 0
(*)            → 1
(append)       → '()
(string-append) → ""
(and)          → #t
(or)           → #f
```

Для вызова свёрточных операций часто используется функция `apply`:

```
(define xs '(1 2 3 4))

(apply * xs)                    → 24
(apply + xs)                    → 10
```

Помимо свёрточных операций (+, * и т.д.) в Scheme произвольное число параметров принимают операции арифметических отношений:

```
(= 1 1 1)                       → #t
(= 1 2 1)                       → #f
(> 7 5 2)                       → #t
```

```
(> 7 5 5)                          → #f
(>= 7 5 5 2)                       → #t
...
```

Операции арифметических отношений не являются свёрточными, т.к. запись вида `(a < b) < c` бессмысленна, но здесь упоминаются для полноты картины. Их тоже можно вызывать при помощи `apply`.

Назначение функции `apply`:

1. Вызов свёрточных операций.
2. Вызов функции с неизвестным числом аргументов.

При помощи `apply` невозможно вызвать `and` и `or`, поскольку они не функции (попытка вызова будет ошибкой синтаксиса).

Лекция 6б. Типы данных и типизация

Типы данных

Тип данных — множество значений, множество операций над ними и способ хранения в памяти компьютера (машинное представление).

Абстрактный тип данных — множество значений и множество операций над ними, т.е. способ хранения не задан.

Первая классификация типов данных:

1. Простые — неделимые порции данных: число, символ, литера.
2. Составные — содержащие значения других типов: cons-ячейка, список, вектор, строка.

Вторая классификация типов данных:

1. Встроенные типы данных — уже заранее есть в языке.
2. Пользовательские — их определяет пользователь.

В ряде языков программирования (например, в Си) есть встроенные в язык средства для определения пользовательских типов данных. Например, в Си встроены различные числовые типы. Пользователь на их основе может создавать массивы, массивы массивов, структуры, объединения и т.д.

В языке Scheme нет языковых средств для определения новых типов данных. Вместо этого пользователь придумывает способ представления некоторого значения при помощи встроенных типов данных и описывает операции над ним в виде набора процедур (иногда, макросов).

Т.е. проектируем представление типа данных и набор операций. При этом не рекомендуется работать с типом данных в обход предоставленных операций.

Если мы задокументируем только набор операций, но не опишем представление, то мы создали *абстрактный тип данных.*

Для типов данных языка Scheme обычно определены четыре вида операций:

- **конструктор** — процедура, имя которой имеет вид `make-‹имя-типа›`, например, `make-vector`, `make-set` (см. дз), конструктор предназначен для создания новых значений данного типа,
- **предикат типа** — процедура, возвращающая `#t`, если её аргумент является значением данного типа, имеет имя `‹имя-типа›?`: `vector?`, `set?` (см. дз), `multi-vector?` (см. дз),
- **модификаторы** — операции, меняющие на месте содержимое объекта, их имя имеет вид `‹тип›-‹операция›!`, например, `vector-set!`, `multivector-set!` (см. дз),
- **прочие операции** имеют имя вида `‹тип›-‹операция›`, `vector-ref`, `set-union`, `string-append` и т.д.

Пользовательские типы данных часто представляют как списки, первым элементом которых является символ с именем типа, а остальные — хранимые значения.

Пример. Тип данных — круг.

```
(define (make-circle x y r)
  (list 'cicrle x y r))

(define (circle? c)
  (and (list? c) (equal? (car c) 'circle)))

(define (circle-center c)
  (list (cadr c) (caddr c)))       ; (cadr xs) = (car (cdr xs))

(define (circle-radius c)
  (cadddr c))

(define (circle-set-center! c p)
  (let ((x (car p))
        (y (cadr p)))
    (set-car! (cdr c) x)
    (set-car! (cddr c) y)
    c))

(define (circle-set-radius! c r)
  (set-car! (cdddr c) r)
  c)
```

Типизация и системы типов

Система типов — совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным

конструкциям, составляющим программу — переменные, выражения, функции и модули.

Определение по Пирсу, **система типов** — разрешимый синтаксический метод доказательства отсутствия определённых поведений программы путём классификации конструкции в соответствии с видами вычисляемых значений.

Классификации систем типов:

1. Наличие системы типов: есть/нет.
2. Типизация статическая/динамическая.
3. Типизация явная/неявная.
4. Типизация сильная/слабая.

Наличие системы типов:

- Нет: язык ассемблера, язык FORTH, язык B (Би) — предшественник Си.
- Есть: все остальные языки.

Статическая и динамическая типизация:

- **Статическая типизация** — у каждой именованной сущности (переменной, функции…) есть свой фиксированный тип, он не меняется в процессе выполнения программы. Примеры: Си, C++, Java, Haskell, Rust, Go.
- **Динамическая** — тип переменной/функции известен только во время выполнения программы. Примеры: Scheme, JavaScript, Python.

Явная и неявная типизация:

- **Явная** — тип данных для сущностей явно записывается в программе. Например, `int x` в языке Си. Языки с явной типизацией: Си, C++, Java и т.д.
- **Неявная** — тип данных можно не указывать. Неявная типизация характерна прежде всего для динамически типизированных языков. В статически типизированных языках используется совместно с выводом типов. Вывод типов переменных присутствует в следующих языках: C++ (ключевое слово `auto`), Go (когда тип переменной не указан), Rust, Haskell и т.д.

```
/* Язык Си, тип указывается явно */
int x = 100;

/* тип выводится компилятором */
auto x = 100;      /* int */
auto y = "abc";    /* const char * */

var x int = 100
var y = 200        /* тип выведет компилятор */
```

Типизация сильная/слабая:

- **Сильная** — неявные преобразования типов запрещены. Например, нельзя сложить строку и число. Языки с сильной типизацией: Scheme, Python, Haskell.
- **Слабая типизация** — неявные преобразования допустимы. Например, в JavaScript при сложении строки с числом число преобразуется в строку. Если в JavaScript в переменной лежит строка с последовательностью цифр, то, при умножении её на число, она неявно преобразуется в число: `'1000' * 5 → 5000`. Примеры языков: JavaScript, Си, Perl, PHP.

Встроенные типы данных языка Scheme

Литерный тип (character)

Слово «символ» в русском языке, применительно к типам в ЯП, двузначно: это и печатные знаки (из которых состоят строки), и некоторые имена (например, часть компилятора — таблица символов (symbol table) хранит в себе свойства именованных сущностей — переменных, функций, типов и т.д.).

По-английски первое называется «character», второе — «symbol». Чтобы нам не путаться, слово «character» в курсе «Основы информатики» мы будем называть *литерой.*

Литерный тип хранит в себе печатные знаки, т.е. знаки, которые вводятся с клавиатуры и выводятся на экран. Из литер состоят строки.

Сообщение о языке R5RS не описывает множество символов, реализация Racket допускает использование всех знаков Юникода (в том числе, кириллицы).

Предикат типа — `(char? ch)`.

Литералы, т.е. то, как записываются символы в программе. Они бывают двух видов: когда символ можно представить печатным знаком, и когда нельзя. В первом случае они записываются как `#\x`, где `x` — сам знак. Например: `#\a` — строчная латинская `a`. `#\!` — восклицательный знак.

Во втором случае они записываются как `#\‹слово›`, например:

- `#\tab` — знак табуляции
- `#\space` — пробел
- `#\newline` — перевод строки (в Си — `\n`)
- `#\return` — возврат каретки (в Си — `\r`)

Замечу, что `#\n` — строчная латинская буква `n`.

Преобразование между символом и его числовым кодом:

```
(char->integer char)              → code
(integer->char code)              → char

(char->integer #\@)               → 48
```

```
(integer->char 48)                  → #\@
```

Сравнение символов (по числовым кодам):

```
(char<? ch1 ch2)
(char>? ch1 ch2)
(char<=? ch1 ch2)
(char>=? ch1 ch2)
(char=? ch1 ch2)
```

Сравнение без учёта регистра:

```
(char-ci<? ch1 ch2)
(char-ci>? ch1 ch2)
...
```

Преобразование регистра (??):

```
(char-upcase #\a)                   → #\A
(char-downcase #\Q)                 → #\q
(char-upcase #\1)                   → #\1
(char-downcase #\!)                 → #\!
```

Предикаты видов литер:

```
(char-whitespace? ch)    ; пробельный символ: пробел, табуляция,
                         ; перевод строки и т.д.
(char-numeric? ch)       ; цифра
(char-alphabetic? ch)    ; буква
(char-upper-case? ch)    ; большая буква
(char-lower-case? ch)    ; строчная буква
```

Строковой тип (string)

Строка — последовательность литер. Строка может быть пустой.

Литерал — текст, записанный в двойных кавычках. Внутри строки допустимы стандартные escape-последовательности языка Си. Примеры:

```
"Hello!"
"First line\nSecond line"
"Он крикнул: \"Превед!\""
```

Создание строк:

```
(make-string count char)
(make-string count)
```

Если символ `char` не указан, то создаётся строка, состоящая `count` символов с кодом `0` (т.е. `(integer->char 0)`).

```
(string-ref str k)                  → k-й символ строки
(string-set! str k char)            ; присваивает k-му символу
```

Нумерация ведётся с нуля.

Функция `string-set!` будет работать со строками, созданными при помощи `make-string`, но при этом может не работать со строками, созданными при помощи литералов.

```
(define s1 (make-string 3 #\a))
s1                                  → "aaa"
(define s2 "aaa")
s2                                  → "aaa"
(string-set! s1 1 #\b)
s1                                  → "aba"
(string-set! s2)                    → ОШИБКА
```

Строку можно создать из отдельных литер:

```
(string #\H #\e #\l #\l #\o)        → "Hello"
```

Можно преобразовать строку в список литер и наоборот:

```
(string->list "Hello")              → (#\H #\e #\l #\l #\o)
(list->string '(#\H #\e #\l #\l #\o))   → "Hello"
```

Сравнение строк (в лексикографическом порядке), с учётом регистра и без него (с суффиксом `-ci`):

```
(string=? str1 str2)
(string<? str1 str2)
...
(string-ci=? str1 str2)
(string-ci<? str1 str2)
```

(Для всех пяти знаков =, <, >, <=, >=.)

```
(string<? "Hello" "Hi")             → #t     ; #\e < #\i
(string<? "Hello" "Hell")           → #f     ; вторая строка короче
```

Длина строки:

```
(string-length "abcdef")            → 6
```

Выбор подстроки:

```
(substring "abcdef" 2 5)            → "cde"
(substring "abcdef" 2 3)            → "c"
```

Числовые типы

Башня числовых типов (каждый верхний предикат включает в себя все нижние):

```
(number? x)                         ; это число
(complex? x)                        ; комплексное число
(real? x)                           ; вещественное число
(rational? x)                       ; дробное число
(integer? x)                        ; целое число
```

Литералы типов:

```
3/4                                 ; дробное число
+3.14  6.022e23   1.38e-23          ; вещественные числа
3+5i   -10-7.5i   2/3+3/4i          ; комплексные числа
```

Целые числа в Scheme имеют неограниченную точность, т.е. число цифр в них ограничено только памятью компьютера. Внутренне, скорее всего, небольшие числа представлены как длинные машинные числа (т.е. `long` в языке Си), большие числа — как массивы цифр в некоторой системе счисления (например, как `unsinged int[]` в системе по основанию 2^{32}).

Предикат `eq?` может различать два больших равных по значению целых числа, если они в памяти представлены двумя разными массивами.

Вещественные числа имеют ограниченную точность (мантисса имеет конечное число значимых цифр). Скорее всего, они будут представлены как `double`.

Язык Scheme — один из редких языков программирования, где поддерживаются рациональные числа на уровне языка:

```
(/ 1 3)                             → 1/3
(/ 10 3)                            → 3 1/3
```

В Scheme числа делятся на точные (exact) и неточные (inexact). Синтаксически неточные записываются с использованием знаков `.` (точка) и `e` (показатель степени).

Арифметические операции с точными числами дают точный ответ. Если хотя бы один из операндов неточный — результат будет неточный (приближённый).

Точные числа — целые числа, рациональные числа и комплексные числа, обе компоненты которых тоже точные (т.е. целые или рациональные).

Неточные числа — вещественные числа или комплексные с вещественными компонентами.

Предикаты:

```
(exact? num)
(inexcat? num)
```

Есть операции преобразования:

```
(exact->inexact num)              → ближайшее вещественное число
(inexact->exact num)              → ближайшее дробное число
```

Примеры:

```
(define pi (* 4 (atan 1)))
pi                                 → 3.141592653589793
(exact? pi)                        → #f
(inexact? pi)                      → #t
(inexact->exact pi)                → 3 39854788871587/281474976710656
```

```
(define googol 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000)
googol                              → 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
(exact? googol)                     → #t
(exact->inexact googol)             → 1e+100

(exact? 1+2/3i)                     → #t
(exact? 1.0+2i)                     → #f
```

Функции преобразования типов

```
(string->number "10.3e7")           → 103000000.0
(number->string 100500)             → "100500"
(string->list "abc")                → (#\a #\b #\c)
(list->vector '(1 2 3))             → #(1 2 3)
(string->number "qwerty")           → #f
```

Символьные вычисления и макросы

Символьные вычисления — это преобразования символьных данных: алгебраические выкладки, трансляция языков программирования и т.д.

ЛИСП был одним из первых языков, ориентированных на символьные вычисления.

Средства языка Scheme для символьных вычислений

Как мы помним, один из постулатов языков семейства ЛИСП — единство кода и данных. Т.е. основным типом данных в ЛИСП является список, и сама программа представлена в виде вложенных списков.

Это не случайно. Программы на ЛИСПе могут преобразовывать программы на ЛИСПЕ как данные, т.е. ЛИСП естественным образом реализует парадигму метавычислений.

Символьный тип данных — это «зацитированное», «замороженное» имя:

```
'hello
```

Имеет известный нам предикат типа и функции преобразования:

```
(symbol? 'hello)                    → #t
(symbol? "hello")                   → #f
(symbol? #\a)                       → #f
(symbol->string 'hello)             → "hello"
(string->symbol "hello")            → hello
(string->symbol
  (string-append
    (symbol->string 'hell)
    (symbol->string 'o))))          → hello
```

Нельзя сказать, что у символов есть свои литералы. Но символы создаются операцией цитирования:

```
'hello                              → hello
(quote hello)                       → hello
'(hello world)                      → (hello world)
(quote (hello world))               → (hello world)
```

Операция цитирования — это особая форма `(qoute …)`, которая принимает терм и «цитирует» его — все идентификаторы в нём становятся символами, остальные атомарные значения остаются как есть, выражения превращаются в списки.

У операции цитирования `(qoute X)` имеется синтаксический сахар `'X`, что демонстрирует пример выше. Ещё пример:

```
(car ''x)                           → ‹что тут будет?›
```

Раскроем сахар:

```
(car (quote (quote x)))             → quote
(equal? (car (qoute (quote x)))
        'qoute)                     → #t
(equal? (car (quote (quote x)))
        (string->symbol "quote"))   → #t
```

Это не ошибка — запись ключевого слова в цитате. Т.е. `'if` — корректная запись. Получится символ `if`.

Литерал вектора подразумевает неявное цитирование:

```
(vector-ref #(a b c) 1)             → b
```

Получим символ `b`.

```
#(10 (+ 5 5) 100)                   → #(10 (+ 5 5) 100)
```

В первом элементе зацитирован список `(+ 5 5)`.

Квазицитирование позволяет внутри цитаты вычислять какие-то значения.

Обозначается она обратной кавычкой (на клавиатуре на букве ё). Для «разцитирования» внутри квазицитаты перед термом записывается знак «запятая».

```
`(10 20 ,(+ 15 15) 40)              → (10 20 30 40)
        ^^^^^^^^^^ — эта часть будет вычислена
```

В квазицитату можно вставлять списки двумя способами. Просто запятая перед переменной вставит список как значение. Знак `,@` (запятая-собачка) вклеит список:

```
(define xs '(a b c d))

`(1 2 ,xs 3 4)                      → (1 2 (a b c d) 3 4)
`(1 2 ,@xs 3 4)                     → (1 2 a b c d 3 4)
```

Т.е. если внутри списка имеем `,@`, то списки будут сконкатенированы.

Конкатенацию двух списков можно написать так:

```
(define (my-append xs ys)
  `(,@xs ,@ys))
```

Квазицитирование тоже является синтаксическим сахаром для особых форм:

```
`expr          ≡ (quasiquote expr)
,expr          ≡ (unquote expr)
,@expr         ≡ (unqoute-splicing expr)
```

Во многих современных языках программирования присутствует похожий на квазицитирование механизм: интерполяция переменных внутри строковых констант. Например, в Python

```
x = 100
y = f"Square of {x} is {x*x}"
y                                    → "Square of 100 is 10000"
```

При помощи цитирования мы можем делать из программы данные.

```
;; Это выражение:
(lambda (x) (* x x))

;; А это цитата, список из трёх элементов, причём два последних
;; тоже списки
'(lambda (x) (* x x))
```

Но возможен ли обратный механизм? Можно ли выражение, записанное в цитаты, выполнить как выражение?

Можно. Для этого используется встроенная функция `eval`. Её синтаксис:

```
(eval ‹цитата› ‹окружение›)        → ‹результат вычисления
                                      ‹цитаты› в ‹окружении››
```

Здесь ‹цитата› — некоторое зацитированное выражение, ‹окружение› — множество значений переменных, которые могут использоваться в ‹цитате›.

Окружение вручную создать нельзя, его можно запросить другими встроенными функциями:

```
(scheme-report-environment 5)     ; встроенные функции и макросы среды
                                  ; R5RS
(null-environment 5)              ; только встроенные макросы R5RS
(interaction-environment)         ; текущие глобальные переменные среды
```

Последнее наиболее употребительное.

Заметим, что `eval` может менять среду, в частности добавлять новые переменные:

```
(eval (list '+ 5 7)
      (interaction-environment))   → 12

(define x 100)
(define y 500)
(eval (list '* 'x 'y)
      (interaction-environment))   → 5000000

(define ie (interaction-environment))

z                                  → ОШИБКА «Нет переменной z»

(eval '(define z 100500) ie)
z                                  → 100500
```

`interaction-environment` хранит только глобальные переменные. Локальные в нём не видны:

```
(define a 100)

(let ((a 200))
  (eval '(* a a)
        (interaction-environment))) → 10000
```

Функции `member` и `assoc`, ассоциативные списки

Функция `member` ищет элемент в списке. Если найден — возвращает хвост списка, начиная с этого элемента. Если нет — `#f`.

```
(member 'b '(a b c d))             → (b c d)
(member 'c '(a b c d))             → (c d)
(member 'z '(a b c d))             → #f
```

Ассоциативный список — это способ реализации ассоциативного массива (т.е. структуры данных, отображающей ключи на значения) при помощи списка, это список пар (cons-ячеек), где в `car` находится ключ, а в `cdr` — связанное значение. Частный случай — список списков, где `car`'ы — ключи, а хвосты — значения. Чаще всего это частный случай и встречается, т.к. правильные списки просто удобнее.

Пример:

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

отображает имена на некоторые числа.

Функция `assoc` принимает ключ и ассоциативный список и возвращает первый элемент с заданным ключом:

```
(assoc 'b '((a 1) (b 2) (c 3)))    → (b 2)
(assoc 'x '((a 1) (b 2) (c 3)))    → #f
```

Синтаксис `cond` со стрелкой используется с `assoc`:

```
(cond ((assoc key table) -> (lambda (val) (cadr val))
      (else 'not-found))
```

У `member` и `assoc` есть «функции-сёстры», которые отличаются предикатом сравнения:

Поиск	Ассоц. список	Предикат
`member`	`assoc`	`equal?`
`memv`	`assv`	`eqv?`
`memq`	`assq`	`eq?`

Макросы

Лирическое отступление. На взгляд лектора, можно выделить три уровня познания языков программирования:

1. Ученик почти наугад подбирает последовательность инструкций, дающую вроде как верный результат. Ну или не почти наугад.
2. Программист выражает свои мысли на языке программирования.
3. Программист выбирает или даже сам создаёт язык программирования, наиболее подходящий для выражения решения задачи.

Язык предметной области (domain specific language, DSL) — это некоторый ограниченный по средствам язык, предназначенный для решения конкретной задачи. Это язык, на котором решение данной задачи лучше всего выражается. Язык предметной области определяется или как интерпретатор (внешний DSL), либо как библиотека для некоторого имеющегося языка (внутренний DSL).

В Scheme для определения DSL'ей часто используются макросы.

Макрос — это инструмент переписывания кода. Т.е. способ создать новую языковую конструкцию на основании имеющихся.

Процесс выполнения выражений на Scheme. Пусть у нас есть выражение вида

`(‹имя› ‹термы…›)`

1. Если `‹имя›` — ключевое слово языка (`if`, `define`, `quote`, `lambda`, и т.д.), то выражение интерпретируется как особая форма.
2. Если `‹имя›` — имя макроса, то данное выражение перезаписывается согласно определению макроса.
3. Если `‹имя›` — имя переменной, то в переменной должна быть процедура, эта процедура вызывается.

Т.е. можно считать, что вычисление выражения состоит из двух этапов:

1. Раскрытие макросов.

2. Собственно вычисления (выполнения особых форм, вызовы процедур).

Синтаксис определения макроса:

```
(define-syntax ‹имя›
  (syntax-rules (‹ключевые слова›)
    (‹образец› ‹шаблон›)
    (‹образец› ‹шаблон›)
    (‹pattern› ‹template›)))
```

‹образец› (‹pattern›) — вид, который должно иметь обращение к макросу. ‹шаблон› (‹template›) — то, на что макрос заменяется.

Образцы проверяются сверху вниз и выбирается тот, который первым подходит.

Что значит: *образец подходит к обращению к макросу (применению макроса)?*

В правилах макроса могут быть переменные (правильнее сказать, **метапеременные**), которым соответствуют фрагменты кода на Scheme. Если в ‹образце› мы можем вместо вхождений переменных подставить фрагменты кода на Scheme таким образом, что получим запись применения макроса, то считаем, что применение макроса с образцом сопоставилось успешно, и правило применяется.

Рассмотрим примеры некоторых макросов. Макрос, имитирующий встроенный макрос `begin`:

```
(define-syntax my-begin
  (syntax-rules ()
    ((my-begin one-action) one-action)
    ((my-begin action . other-actions)
     (let ((x action))
       (my-begin . other-actions)))))
```

Как это определение читается?

Если обращение к макросу имеет вид (`my-begin ‹одно какое-то действие›`), то это дейтвие результатом раскрытия макроса (первое правило).

```
(my-begin (display 'hello))        ≡ (display 'hello)
```

В этом примере вместо метапеременной `my-begin` может быть подставлено имя макроса `my-begin`, вместо метапеременной `one-action` может быть подставлено подвыражение `(display 'hello)`, поэтому первое правило применимо. Первое правило выполнится, макрос заменится на шаблон правила, состоящий из одной переменной `one-action`, вместо неё будет подставлено `(display 'hello)`.

Второе правило применимо, когда не применимо первое правило и обращение к макросу может быть сопоставлено с образцом

`(my-begin action . other-actions)`. Образец состоит из трёх переменных, последняя в позиции после точки, т.е. она будет сопоставлена с концом списка. Образец говорит о том, что обращение к макросу должно быть списком из минимум двух элементов: первый будет отображён на переменную `my-begin`, второй — на `action`, все остальные — на `other-actions`.

Поскольку случай ровно двух элементов в списке перехватывается предшествующим правилом, в `other-actions` у нас всегда будет непустой список.

По смыслу это означает, что второе правило описывает конструкцию `my-begin`, в которой не менее двух выражений: имя макроса `my-begin` сопоставится с первой переменной, первое выражение сопоставится с `action`, последующие как список — с `other-actions`.

Рассмотрим правую часть (шаблон, template) второго правила:

```
((my-begin action . other-actions)
 (let ((x action))
   (my-begin . other-actions)))
```

По шаблону будет построено let-выражение с одной переменной, с переменной свяжется значение выражения, попавшего в `action`, внутри let'а будет рекурсивно применён макрос `my-begin` со всеми остальными выражениями. Чтобы из списка `other-actions` получить список, где первым элементом будет `begin`, а хвостом — `other-actions`, мы строим cons-пару при помощи точечной нотации (в макросах точечная нотация используется вместо `cons`).

Если аргументом макроса будет несколько каких-то выражений (т.е. список выражений), то строится let-выражение, результат первого действия связывается с переменной и тем самым вычисляется до всех остальных. Остальные (их может быть несколько) заворачиваются в `my-begin`, который обеспечит их последовательное выполнение (т.е. макрос вызывается рекурсивно).

Рассмотрим последовательные раскрытия макроса `my-begin` на примере:

```
(my-begin                              (let ((x (display 'hello)))
  (display 'hello)                        (my-begin
  (display 'my)                             (display 'my)
  (display 'world))                         (display 'world)))
```

В первом применении макроса первое правило не применимо, т.к. невозможно отобразить список из 4 элементов на список из двух элементов `(my-begin one-action)`. Второе правило применимо: можно отобразить список из четырёх элементов на `(my-begin action . other-actions)`, получим следующие подстановки для переменных:

- `my-begin` ← `my-begin`,
- `(display 'hello)` ← `action`,

- `((display 'my) (display 'world))` ← `other-actions`.

Подстановка их в правую часть

```
(let ((x action))
  (my-begin . other-actions))
```

даст

```
(let ((x (display 'hello)))
   (my-begin
     (display 'my)
     (display 'world)))
```

Полное раскрытие приведёт к выражению:

```
(let ((x (display 'hello)))          (let ((x (display 'hello)))
   (let ((x1 (display 'my)))           (let ((x1 (display 'my)))
      (my-begin (display 'world))))      (display 'world)))
```

Подробнее о следующих шагах раскрытия.

Второй шаг раскрытия тоже задействует второе правило (т.к. список из трёх элементов, переменные будут следующие:

- `my-begin` ← `my-begin`,
- `(display 'my)` ← `action`,
- `((display 'world))` ← `other-actions`.

Их подстановка даст

```
(let ((x1 (display 'my)))
   (my-begin (display 'world))))
```

Третье раскрытие будет по первому правилу, т.к. список из двух элементов можно отобразить на список двух переменных (`my-begin one-action`):

- `my-begin` ← `my-begin`,
- `(display 'world)` ← `one-action`.

Подстановка в правую часть

```
    ((my-begin one-action) one-action)
```

даст одно `one-action`, т.е. `(display 'world)`.

Макросы в Scheme гигиенические, т.е. о конфликте имён при их раскрытии беспокоиться не нужно. В примере выше для различных раскрытий макроса сгенерированы разные имена `let`'ов: `x` и `x1`.

Ключевые слова в макросе не являются метапеременными и трактуются буквально.

Пример. Определим макрос `my-cond`, частично имитирующий встроенный макрос `cond`:

```
(define-syntax my-cond
  (syntax-rules (else)
    ;; последняя ветка else
    ((my-cond (else . actions)) (begin . actions))

    ;; последняя ветка не else
    ((my-cond (condition . actions))
     (if condition
         (begin . actions)
         #f))  ;; когда нам нечего вернуть, возвращаем #f

    ;; не последняя ветка
    ((my-cond (condition . actions) . branches)
     (if condition
         (begin . actions)
         (my-cond . branches)))))
```

Исходный код

```
(my-cond ((> x 0) (display 'pos) (newline))
         ((< x 0) (display 'neg) (newline)))
```

Здесь сработает третье правило:

- `condition` → `(> x 0)`
- `actions` → `((display 'pos) (newline))`
- `branches` → `(((< x 0) (display 'neg) (newline)))`

Код перепишется в

```
(if (> x 0)
    (begin (display 'pos) (newline))
    (my-cond ((< x 0) (display 'neg) (newline))))
```

На рекурсивном обращении к макросу сработает вторая ветка, в результате макрос раскроется в

```
(if (> x 0)
    (begin (display 'pos) (newline))
    (if (< x 0)
        (begin (display 'neg) (newline))
        #f))
```

Пример. Цикл со счётчиком.

```
(define-syntax for
  (syntax-rules (:= to downto do)
    ((for var := start to end do . actions)
     (let ((limit end))
       (let loop ((var start))
         (and (<= var limit)
              (begin
                (begin . actions)
                (loop (+ var 1)))))))
    ((for var := start downto end do . actions)
```

```
    (let ((limit end))
      (let loop ((var start))
        (and (>= var limit)
             (begin
               (begin . actions)
               (loop (- var 1)))))))))

(for x := 1 to 10 do
  (display x)
  (newline))
```

Макросы в Scheme **гигиенические.** Это означает, что для каждого раскрытия макроса имена переменных в `let`, `letrec`, `let*`, параметрах `lambda` и `define` генерируются новые. А значит, конфликт имён исключён.

В образцах макросов можно использовать вместо имени переменной знак _, означающий безымянную переменную. Он используется, когда конкретное значение не нужно (игнорируется). Чаще всего он используется для имени самого макроса:

```
(define-syntax my-begin
  (syntax-rules ()
    ((_ one-action) one-action)
    ((_ action . other-actions)
     (let ((x action))
       (my-begin . other-actions)))))
```

В макросах можно использовать т.н. «эллипсис», т.е. `...`. Эллипсис в макросах оставляется вам на самостоятельное изучение.

Разработка через тестирование

Разработка через тестирование — способ разработки программы, предполагающий написание **модульных тестов** (unit tests) до написания кода, который они проверяют.

Модульный тест — автоматизированный тест, проверяющий корректность работы небольшого фрагмента программы (процедуры, функции, класса и т.д.). Модульный тест обязательно должен быть самопроверяющимся, т.е. без контроля пользователя запускает тестируемую часть программы и проверяет, что результат соответствует ожидаемому.

Цикл разработки через тестирование:

1. Пишем тест для нереализованной функциональности. Этот тест при запуске *проходить не должен.*
2. Пишем функциональность, *но ровно на столько,* чтобы новый тест проходил. При этом все остальные тесты тоже должны проходить (не сломаться).
3. **Рефакторинг** — это эквивалентное преобразование программы, направленное на улучшение её внутренней структуры

(повышение ясности программы, её расширяемости, эффективности). *В процессе рефакторинга ни один из модульных тестов сломаться не должен.*

Продолжительность одного цикла — около минуты.

Ещё к лабораторной работе

`(load ‹имя файла›)`

Эта процедура читает и выполняет указанный файл. Её можно считать примерным аналогом `#include` в языке Си.

```
(write expr)
(display expr)
(newline)
```

Процедура `write` печатает машиночитаемом формате, т.е., например, строки выводит в кавычках и с escape-последовательностями. Процедура `display` — в человекочитаемом, т.е. символы строк выводит буквально. `newline` печатает перевод на новую строчку.

`(display "1234")` и `(display 1234)` выведут идентичный текст.

Лекция 9a. Ввод-вывод в языке Scheme

Мы будем рассматривать сегодня ввод-вывод в языке Scheme R5RS. Язык Scheme R5RS — не промышленный, а академический, поэтому средства ввода-вывода в нём довольно ограничены.

Для сравнения, Common Lisp — промышленный язык, в нём средства ввода-вывода более обширны.

Для абстракции ввода-вывода в Scheme R5RS используется понятие порта. Есть порт ввода и порт вывода по умолчанию, можно создавать новые порты, связанные с файлами.

Порты ввода-вывода, открытие и закрытие

Порт ввода по умолчанию связан с клавиатурой (`stdin`, в терминах языка Си), порт вывода — с экраном (`stdout`, в терминах языка Си). Эти порты по умолчанию можно переназначать.

Предикат типа «порт»:

```
(port? x)                          → #t или #f
```

Создание порта:

```
(open-input-file "имя файла")      → port
(open-output-file "имя файла")     → port
```

Предусловие для `open-output-port`: файл существовать не должен (иначе ошибка).

После использования порты, связанные с файлами, нужно закрывать:

```
(close-input-port port)
(close-output-port port)
```

Порты по умолчанию:

```
(current-input-port)                → port
(current-output-port)               → port
```

Временное перенаправление портов:

```
(with-output-to-file "имя файла" proc)       ≡ (proc)
(with-input-from-file "имя файла" proc)      ≡ (proc)
```

Здесь `proc` — процедура без параметров, во время выполнения этой процедуры порты вывода и ввода, соответственно, по умолчанию будут перенаправлены.

Возвращаемое значение у этих функций то же, что у вызванной процедуры.

```
(with-output-to-file "D:/test.txt"
  (lambda ()
    (display 'hello)))
```

В файл `D:/test.txt` будет записана строчка `hello`.

Открытие портов с автоматическим закрытием:

```
(call-with-input-file "имя файла" proc)      ≡ (proc port)
(call-with-output-file "имя файла" proc)     ≡ (proc port)
```

Здесь функция `proc` будет принимать порт в качестве параметра:

```
(call-with-output-file "D:/test2.txt"
  (lambda (port)
    (display 'hello port)))
```

Порт, переданный в процедуру, будет закрыт при завершении вызова самой процедуры.

Чтение и запись символов

Для чтения и записи используются функции `read-char`, `peek-char`, `write-char`:

```
(read-char)                         → char | eof-object
(read-char port)                    → char | eof-object

(peek-char)                         → char | eof-object
(peek-char port)                    → char | eof-object

(write-char char)
(write-char char port)
```

Если порт не указан, то используется порт по умолчанию. Функции `read-char` и `peek-char` возвращают либо литеру, либо признак конца файла (end-of-file). Проверка прочитанного на конец файла делается предикатом:

```
(eof-object? obj)                    → #t | #f
```

Функция `read-char` читает литеру и забирает его из источника, функция `peek-char` — читает литеру и оставляет её в источнике. Т.е. вызов

```
(let* ((x (read-char))
       (y (read-char))
       (z (read-char)))
   (list x y z))
```

вернёт три последовательных символа из порта. Вызов

```
(let* ((x (peek-char))
       (y (peek-char))
       (z (peek-char)))
   (list x y z))
```

построит список из трёх одинаковых литер, причём литера останется во входом порту — её можно будет получить следующим вызовом `read-char` или `peek-char`.

Ввод-вывод выражений

Можно читать и записывать целые s-выражения, синтаксический анализ будет выполнен библиотекой, а мы получим готовое выражение.

```
(write expr)
(write expr port)
```

Функция `write` выписывает в порт выражение в машиночитаемом виде. Т.е. выписанное выражение однозначно понятно. Для чтения машиной записанного выражения используется функция

```
(read)                               → expr | eof-object
(read port)                          → expr | eof-object
```

То, что мы записали при помощи `write`, мы можем потом прочитать при помощи `read`. Однако, не все данные поддаются чтению обратно.

Снова прочитать мы можем только данные следующих типов:

- `#t`, `#f`,
- числа: `100500`, `2/3`, `3.1415926`, `7+2i`,
- строки: `"Hello!"`,
- литеры: `#\H`, `#\!`, `#\newline`,
- символы: `'hello`,
- списки и вектора всего вышеперечисленного.

Снова прочитать мы можем только объекты, для которых существуют литералы. Собственно, функция `write` и выписывает данные в виде литералов, а функция `read` их разбирает.

Снова прочитать мы не можем объекты, существующие при выполнении конкретного процесса: процедуры (`lambda`), порты, продолжения (continuations). Для двух последних литералов не существует. Синтаксис `(lambda …)` можно считать литералом для процедуры, но процедура — вещь существенно динамическая, т.к. захватывает переменные из своего окружения (см. идиому статических переменных).

Прочий вывод

Здесь рассмотрим вывод человекочитаемых данных, он представлен двумя функциями:

```
(display)
(display port)
(newline)
(newline port)
```

`display` выводит данные в человекочитаемом виде. Т.е. строки выводятся не как свои литералы, а с буквальной интерпретацией символов в них (перевод строки приведёт к печати перевода строки в файле, а не выдаче литер `\` и `n`). Вывод следующих трёх вызовов будет идентичен:

```
(display #\x)
(display "x")
(display 'x)
```

По выводу будет непонятно, что же было реальным аргументом. Но, когда пользуются функцией `display`, это и не нужно.

Если для отладки нужно выводить какое-то выражение, то его лучше выводить при помощи `write`, т.к. по выводу будет однозначно понятно содержимое. В частности, `write` следует использовать в макросе `trace-ex` и каркасе модульных тестов в ЛР3 для вывода выражений.

REPL (read-evaluate-print loop)

REPL (read-evaluate-print loop) — режим интерактивной работы с интерпретируемыми языками программирования. Пользователь вводит конструкцию языка (выражение, оператор), она тут же интерпретируется и результат выводится на экран. После чего пользователь может снова что-то ввести.

Впервые REPL появился для языка LISP, сейчас он поддерживается многими интерпретаторами языков программирования. Например, среда IDLE в Python, консоль JavaScript, доступная в любом браузере (часто вызывается по F12).

REPL можно реализовать в Scheme самостоятельно:

```
(define (print expr)
  (write expr)
  (newline))

(define (REPL)
  (let* ((e (read))                                    ; read
         (r (eval e (interaction-environment)))        ; eval
         (_ (print r)))                                ; print
    (REPL)))                                           ; loop
```

Добавим поддержку конца файла:

```
(define (REPL)
  (let* ((e (read)))                                   ; read
    (if (not (eof-object? e))
        (let* ((r (eval e (interaction-environment)))  ; eval
               (_ (print r)))                          ; print
          (REPL)))))                                   ; loop
```

Встроенная функция `load` позволяет прочитать и проинтерпретировать содержимое файла:

```
(load "trace.scm")
(load "unit-tests.scm")
```

Аналог функции `load` можно написать самостоятельно:

```
(define (my-load filename)
  (with-input-from-file filename REPL))
```

Примечание. Чтобы среда DrRacket не печатала `#<void>` для конструкций без значения в нашем импровизированном REPL'е, функцию `print` можем уточнить:

```
(define the-void (if #f #f))

(define (print expr)
  (if (not (equal? expr the-void))
      (begin
        (write expr)
        (newline))))
```

Лекция 9б. Мемоизация и нестрогие вычисления

Мемоизация — оптимизация, позволяющая избегать повторного вычисления функции, вызванной с теми же аргументами, как и в один из прошлых вызовов.

В общем случае нужно мемоизировать чистые функции: детерминированные и без побочных эффектов. Обоснование:

- Нет смысла мемоизировать функцию `(read port)` — она на каждом вызове должна выдавать очередное значение из файла.
- Нет смысла мемоизировать функцию `(display message)`, т.к. она должна выполнять побочный эффект.

Пример мемоизации «нечистой» функции. Функция `(get-messages lang-id)` загружает из файла ресурсов сообщения программы на некотором языке. В процессе работы программы пользователь может залезть в настройки и поменять выбранный язык. Если функцию вызывать при каждой потребности вывести на экран сообщение, то она будет многократно читать один и тот же файл. Если загружать все ресурсы для всех языков в начале работы программы, то будут загружены лишние ресурсы. Приемлемый вариант — мемоизировать вызовы `get-messages`.

Приём мемоизации в Scheme. Пусть нам нужно мемоизировать функцию вида

```
(define (func x y z)
  (+ x y z))
```

Данная функция должна при вызове с ранее известными аргументами «вспоминать» свой результат, не вычисляя его заново. Но если аргументы новые, она должна результат вычислить и запомнить его.

Для этой цели нам нужно некоторое хранилище, где мы будем сопоставлять аргументы с вычисленными результатами. Для этой цели проще всего использовать ассоциативный список. Делать хранилище открытым тоже не стоит — снаружи должна быть видна только функция `func` (хороший стиль — минимизировать область видимости переменных). Соответственно, будем использовать приём статических переменных в языке Scheme:

```
(define func-memo
  (let ((known-results '()))
    (lambda (x y z)
       ...)))
```

Если значения аргументов есть в хранилище, то нужно вернуть известный результат. Если нет — вычислить и положить в список `known-results`:

```
(define func-memo
  (let ((known-results '()))
    (lambda (x y z)
      (let* ((args (list x y z))
             (res (assoc args known-results)))
        (if res
            (cadr res)
            (let (res (+ x y z))
              (set! known-results (cons (list args res) known-
results))
```

```
            res))))))
```

Так мы запоминаем все предыдущие значения. Иногда функция часто вызывается с теми же значениями, которые были в прошлый раз. Тогда имеет смысл запоминать только последнее значение:

```
(define func-memo-last
  (let ((last-arg #f)
        (known-result #f))
    (lambda (x y z)
      (let ((arg (list x y z)))
        (if (equal? arg last-arg)
          known-result
          (let ((res (+ x y z)))
            (set! last-arg arg)
            (set! known-result res)
            res))))))
```

Строгие вычисления — аргументы функции полностью вычисляются до того, как эта функция вызывается. Вызовы процедур в Scheme всегда строгие. Строгую стратегию вычислений часто называют call-by-value.

В случае **нестрогих вычислений** значения выражений могут вычисляться по необходимости, их вызов может быть отложен.

Примеры нестрогих вычислений:

- `(if cond then else)` — вычисляется либо `then`, либо `else`.
- `(and …)`, `(or …)`.
- В языке Си логические операции `&&`, `||` тоже не строгие.

В теории рассматривают две разновидности нестрогих вычислений:

- call-by-name, вызов по имени — нормальная редукция в лябмда-исчислении,
- call-by-need, вызов по необходимости — ленивые вычисления.

В некотором смысле разновидность call-by-name — макроподстановка:

```
(define-syntax double
  (syntax-rules ()
    ((double x) (+ x x)))

(define-syntax ++
  (syntax-rules ()
    ((++ var) (begin (set! var (+ var 1))
                     var))))

(define x 10)

(double (++ x))  ;; выведет 23
x                ;; выведет 12
```

Вызов по имени в Алголе-60

```
function sum(i, start, end, val): real;
    integer i, start, end;
    real val;
    value start, end;
begin
  real res := 0;
  for i := start to end do
    res := res + val;

  comment почему нельзя res := (end - start + 1) * val?;

  sum := res
end;

function square(x): real;
     integer x;
begin
  square := x * x;
end;

real temperature[1 : 100];

integer k;

print(sum(k, 1, 10, square(k)));
print(sum(k, 1, 100, temperature[k]) / 100);
```

Пример стратегии `call-by-need` — ленивый язык программирования Хаскель

```
test xs = head (map (\x -> x*x) xs)
```

Будет вычисляться квадрат только самого первого элемента списка.

Примитивы Scheme для обеспечения ленивых вычислений

Это макрос `(delay expr)` и функция `(force promise)`. Макрос `delay` принимает выражение и формирует обещание (promise) вычислить это выражение, когда потребуется. `force` вычисляет этот promise, результат мемоизируется.

В первом приближении:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (lambda () expr))))
```

```
(define (force promise)
  (promise))
```

С мемоизацией:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (list #f (lambda () expr)))))

(define (force promise)
  (if (car promise)
      (caar promise)
      (begin
        (set-car! (list ((cadr promise))))
        (caar promise))))
```

Лекция 10. Стек вызовов в Scheme. Продолжения

В сегодняшней лекции мы рассмотрим, как интерпретатор Scheme выполняет вызовы функций и какие средства контроля за этим Scheme даёт программисту.

Стек вызовов. Как осуществляются вызовы функций на Scheme

Выполнение программы на языке Scheme состоит из двух этапов: раскрытие всех макроподстановок и синтаксического сахара до базовых примитивов языка и выполнение программы, записанной в терминах этих базовых примитивов. Сегодня мы будем рассматривать второй этап вычислений, поэтому большинство примеров кода будет написано в терминах базовых примитивов.

В число базовых примитивов, помимо `define`, `if`, `lambda`, `quote`, `set!`, мы добавим также `begin` для удобства изложения (хотя он может быть выражен через `let`, а тот, в свою очередь, через `lambda`).

Процесс вычисления выражений мы будем изображать путём редукции: выбираем очередное подвыражение и заменяем его на результат.

Пример. Рассмотрим редукцию выражения

```
((lambda (foo bar)
   (begin
     (set! bar (+ foo bar))
     (* foo bar 2)))
 3 4)
```

Шаги редукции:

```
Выражение                                                    Среда

((lambda (foo bar) (begin (set! bar (+ foo bar)) (* foo bar 2))) 3 4)
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
; вызывается lambda, появляется среда с локальными переменными

(begin (set! bar (+ foo bar)) (* foo bar 2))                 foo → 3, bar → 4
                  ^^^
; подстановка значения вместо переменной foo

(begin (set! bar (+ 3 bar)) (* foo bar 2))                   foo → 3, bar → 4
                    ^^^
; подстановка значения вместо переменной bar

(begin (set! bar (+ 3 4)) (* foo bar 2))                     foo → 3, bar → 4
                   ^
; вызов встроенной функции +

(begin (set! bar 7) (* foo bar 2))                           foo → 3, bar → 4
       ^^^^
; выполнение оператора присваивания

(begin #<void> (* foo bar 2))                                foo → 3, bar → 7
 ^^^^^
; редукция begin

(* foo bar 2)                                                foo → 3, bar → 7
   ^^^
; подстановка значения вместо переменной foo

(* 3 bar 2)                                                  foo → 3, bar → 7
     ^^^
; подстановка значения вместо переменной bar

(* 3 7 2)                                                    foo → 3, bar → 7
 ^
; вызов встроенной функции *

42
```

При выполнении вызовов функций мы будем предполагать, что аргументы вычисляются слева направо. Согласно R5RS, порядок вычислений не определён, но среда DrRacket вычисляет их слева направо. Также, вслед за DrRacket, будем считать, что операция `set!` и функции типа `display` возвращают значение типа `#<void>` (согласно R5RS значение также не определено).

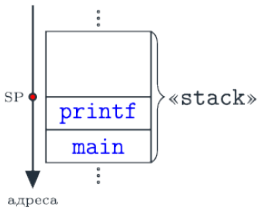
В последующих выкладках некоторые очевидные шаги редукции (вроде подстановки значений вместо переменных) мы будем опускать.

В примере на редукцию выше, мы не касались вопроса о механизме вызовов функций. Сейчас рассмотрим его подробнее.

По курсу «Алгоритмы и структуры данных» мы знакомы с понятием «стек вызовов» и с тем, как в языке Си осуществляются вызовы функций:

Фреймы функций в автоматической памяти

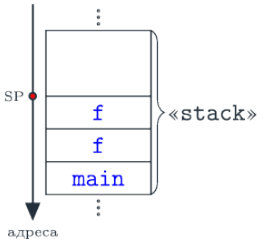
Фрейм функции – это участок автоматической памяти, в котором хранятся локальные переменные (включая формальные параметры) и адрес, на который должно быть передано управление после завершения функции (*адрес возврата*).



При запуске программы в старших адресах области «stack» создаётся фрейм функции `main` и управление передаётся по её адресу.
Если, например, из функции `main` вызывается функция `printf`, то перед фреймом `main` размещается фрейм `printf`. При этом в качестве адреса возврата в этот фрейм помещается адрес инструкции функции `main`, которая следует за инструкцией вызова функции `printf`.
При завершении функции `printf` управление передаётся по записанному в её фрейме адресу возврата, а сам фрейм уничтожается.

Поддержка рекурсии

Хранение локальных переменных функции во фрейме, который создаётся в момент вызова функции, обеспечивает возможность *рекурсии*, т.е. вызова функцией самой себя.



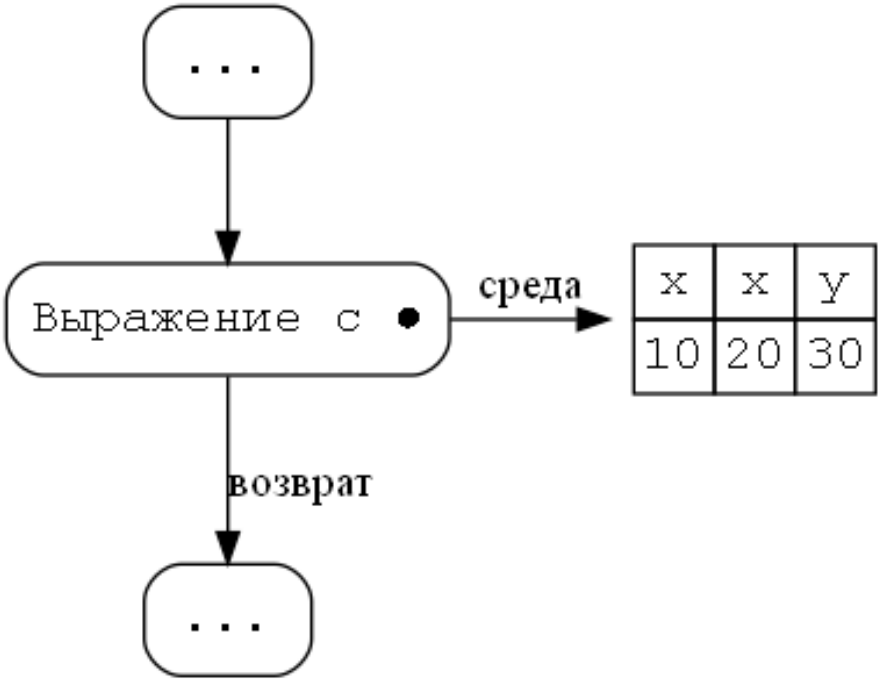
Например, пусть из функции `main` вызывается функция `f`, а из неё ещё раз вызывается функция `f`. В результате в области «stack» появляются два фрейма функции `f`, каждый со своим набором локальных переменных. Естественно, локальные переменные (в которые входят и формальные параметры) в разных фреймах функции `f` никак не пересекаются и могут иметь разные значения.
Бывают более сложные виды рекурсии, например, когда функция `f` вызывает `g`, а та, в свою очередь, вызывает `f`.
Отметим, что использование рекурсии может привести к исчерпанию свободного места в области «stack».

В языке Scheme дело обстоит аналогичным образом, с той лишь разницей, что и фреймы стека, и данные распределяются в динамической памяти. И если ссылка на фрейм где-то сохранена, то объект фрейма останется «жить» даже после возврата функции.

Фреймы стека языка Си содержат адрес возврата и локальные переменные. Параметры функций являются разновидностью локальных переменных.

Как мы помним, конструкции `let`, `let*` и `letrec`, а также `define` внутри `begin`, вводящие локальные переменные, являются синтаксическим сахаром, реализованным поверх `lambda`. Поэтому для языка Scheme локальные переменные во фреймах стека — это *только* параметры функций.

Фрейм стека будем изображать следующим образом:



Фрейм стека содержит две ссылки. Одна из называется «среда» и ссылается на значения локальных переменных — аргументов функции. Вторая — «возврат» ссылается на предыдщий фрейм стека. Если фрейм стека не верхний, то выражение будет содержать символ ●, означающий точку, куда будет возвращено выполнение вызова другой функции.

Рассмотрим пример — вычисление числа Фибоначчи по номеру. Пусть нам дана функция

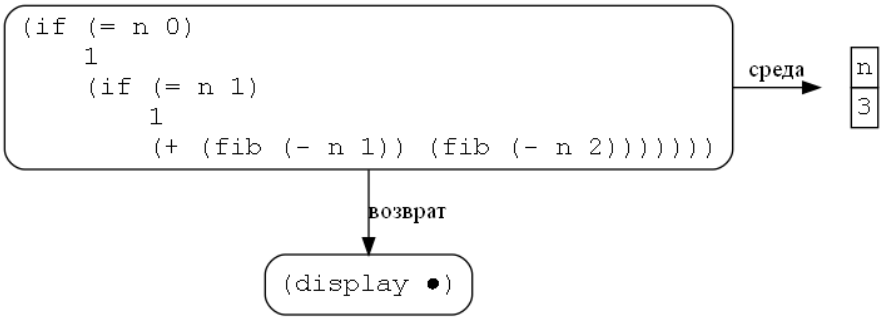
```
(define fib
  (lambda (n)
```

```
(if (= n 0)
    1
    (if (= n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2)))))))
```

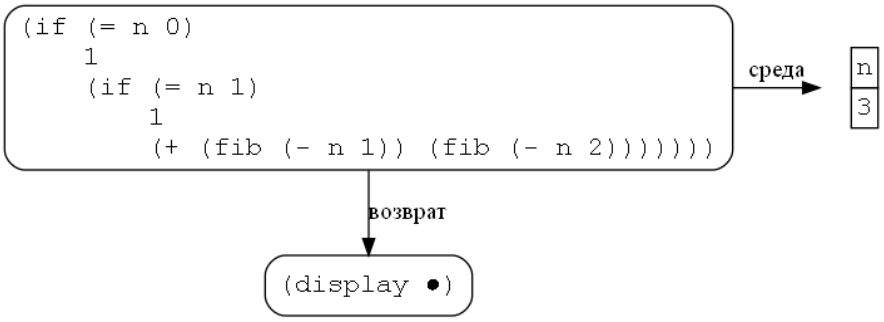
Рассмотрим процесс вычисления выражения

`(display (fib 3))`

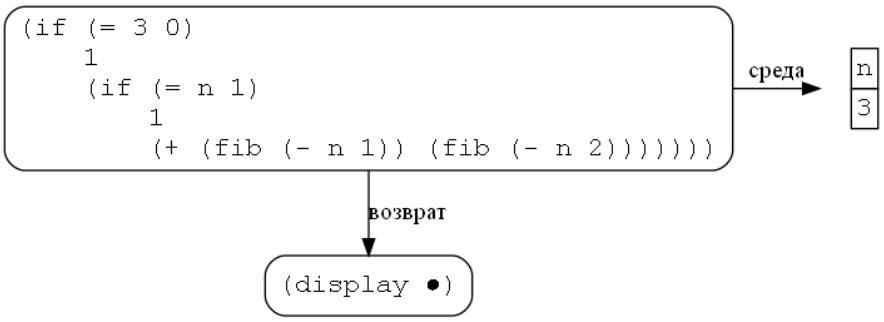
Начальное состояние:



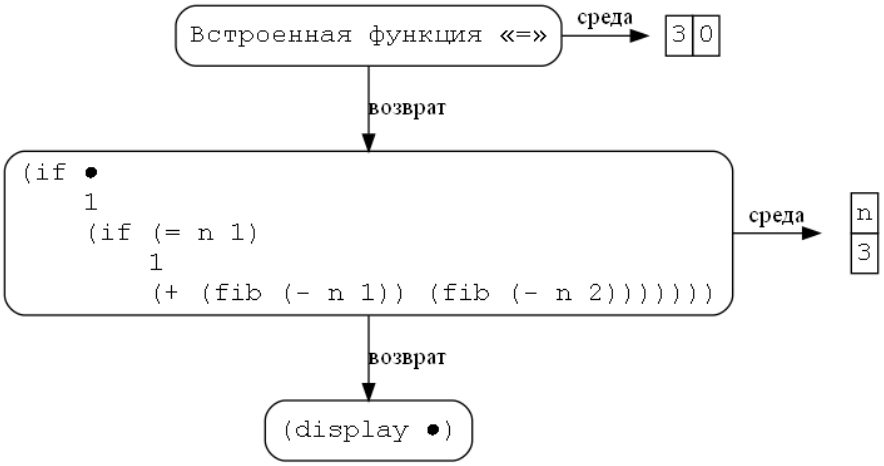
Создаётся фрейм стека для `(fib 3)`:



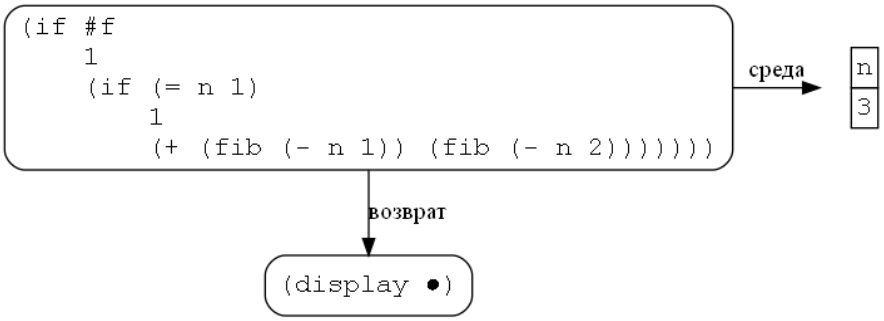
Подстановка значения вместо переменной в `if`:



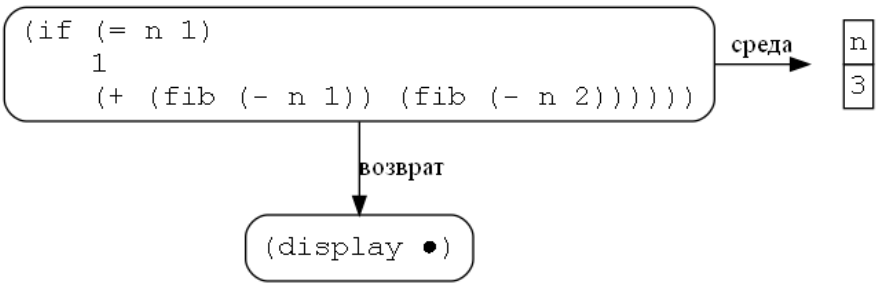
Аргументы для встроенной функции = вычислены, вызов встроенной функции. Имена переменных в среде не указаны, т.к. функция встроенная и мы их не знаем.



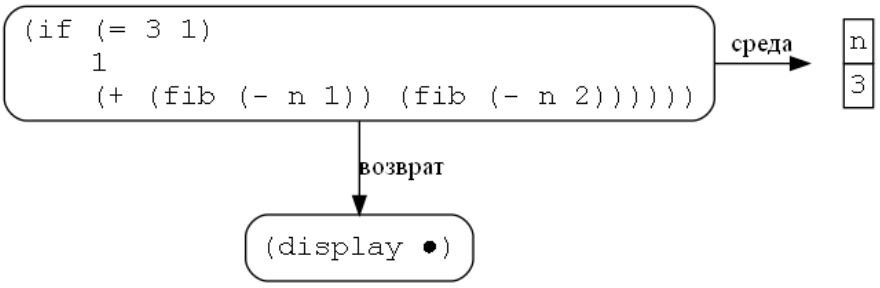
Функция = вернула ложь:



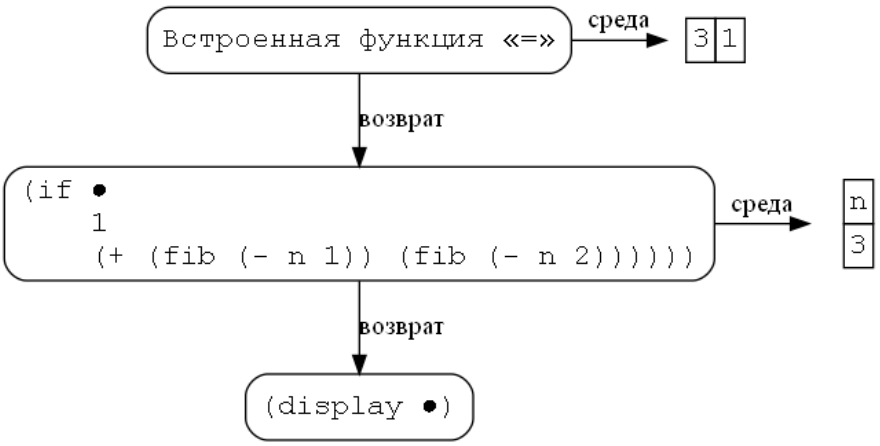
Редукция `if`:



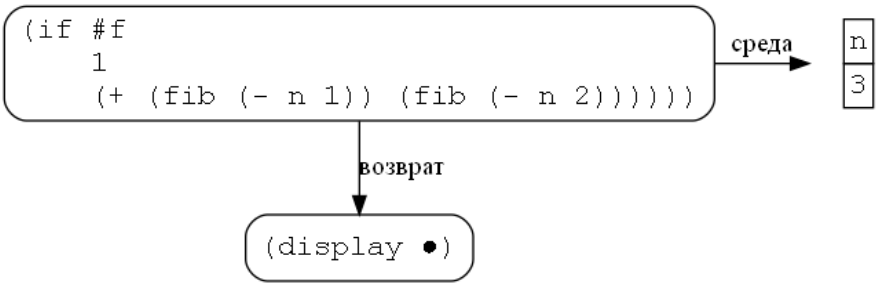
Подстановка значения переменной:



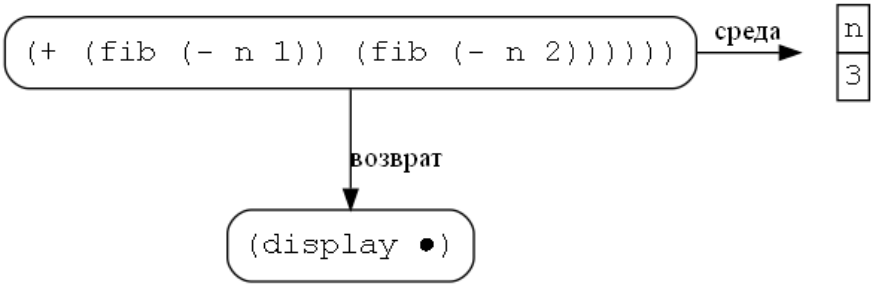
Вызов =:



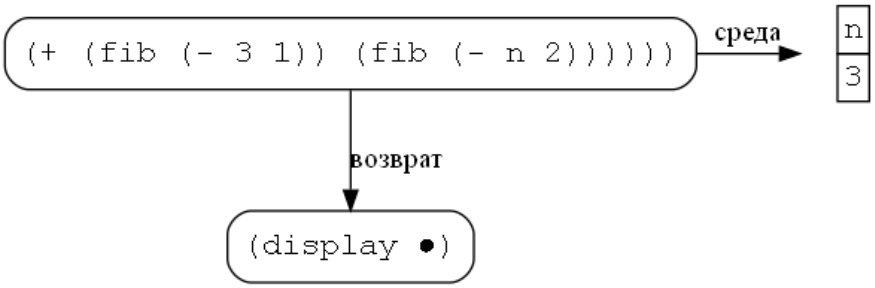
Фрейм стека для =:



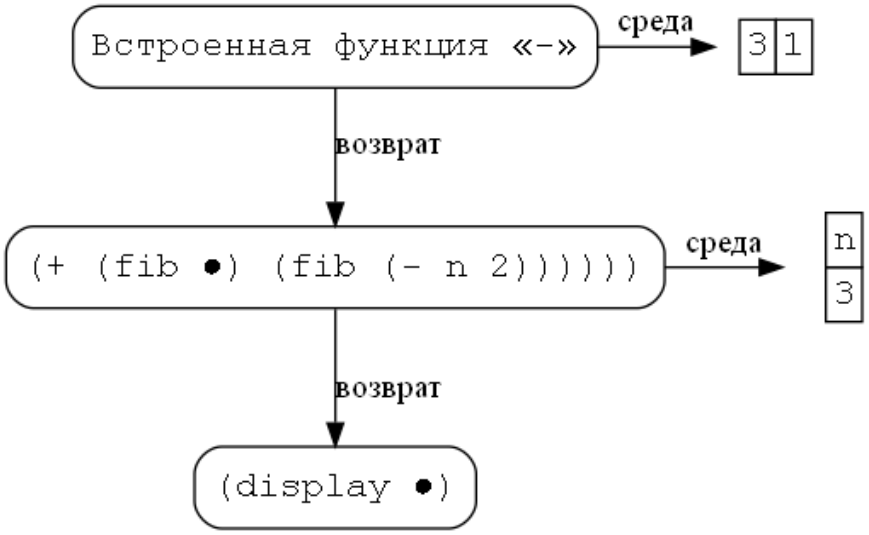
Возврат `#f` из = (опущен), редукция `if`:



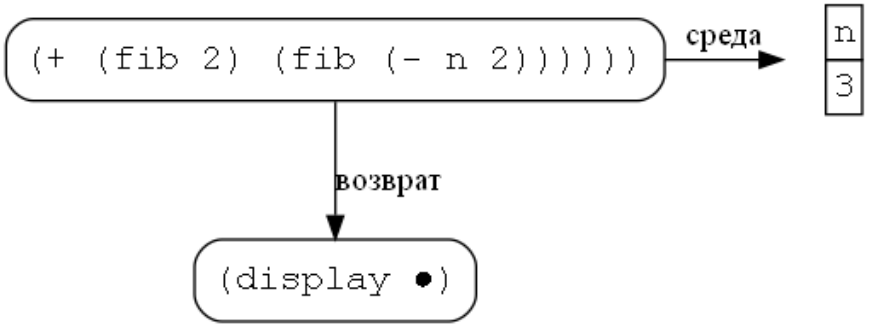
Подстановка значения переменной:



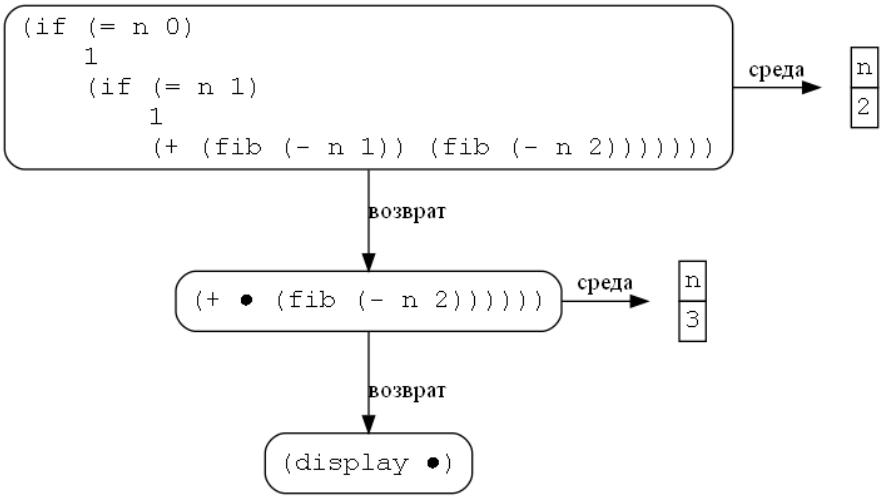
Аргументы у встроенной функции `-` вычислены, её вызов и фрейм стека:



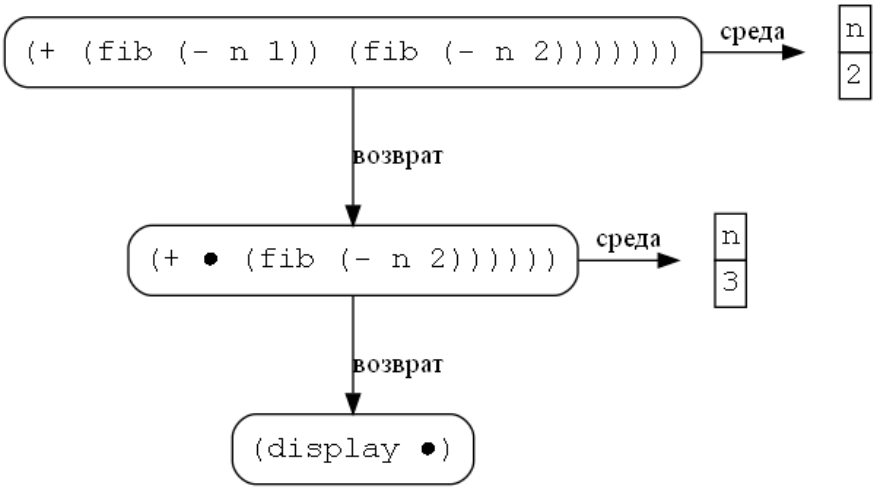
Возврат из функции `-`:



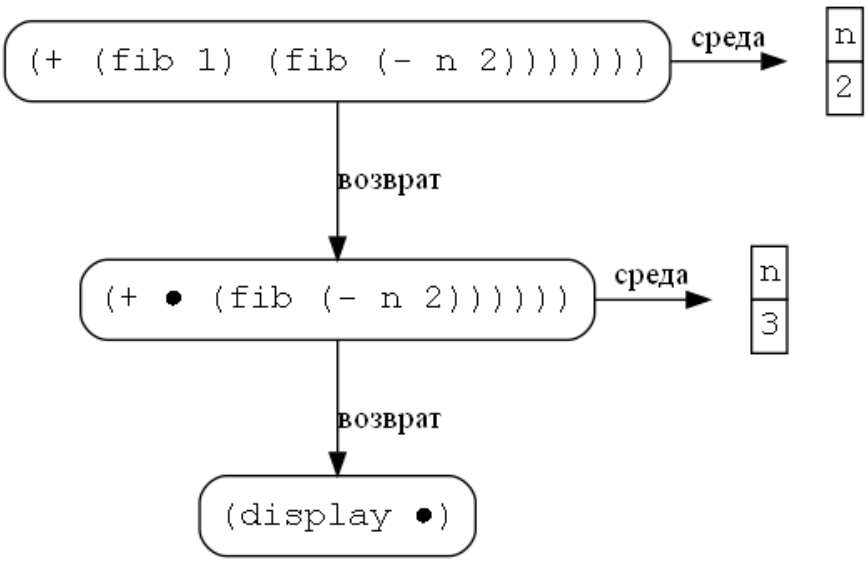
Рекурсивный вызов `fib`:



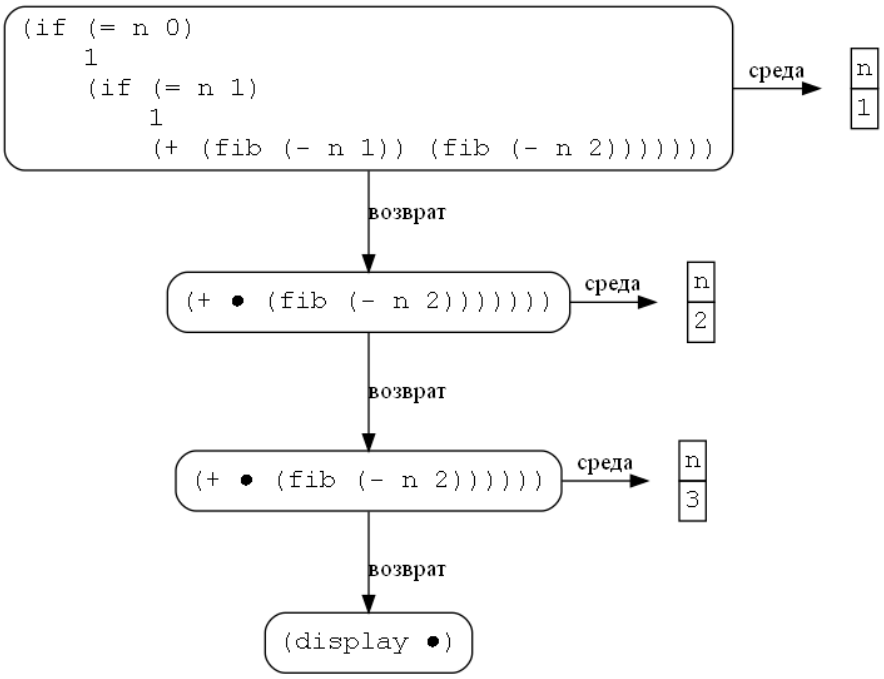
Очевидные вызовы = и шаги редукции `if` пропускаем:



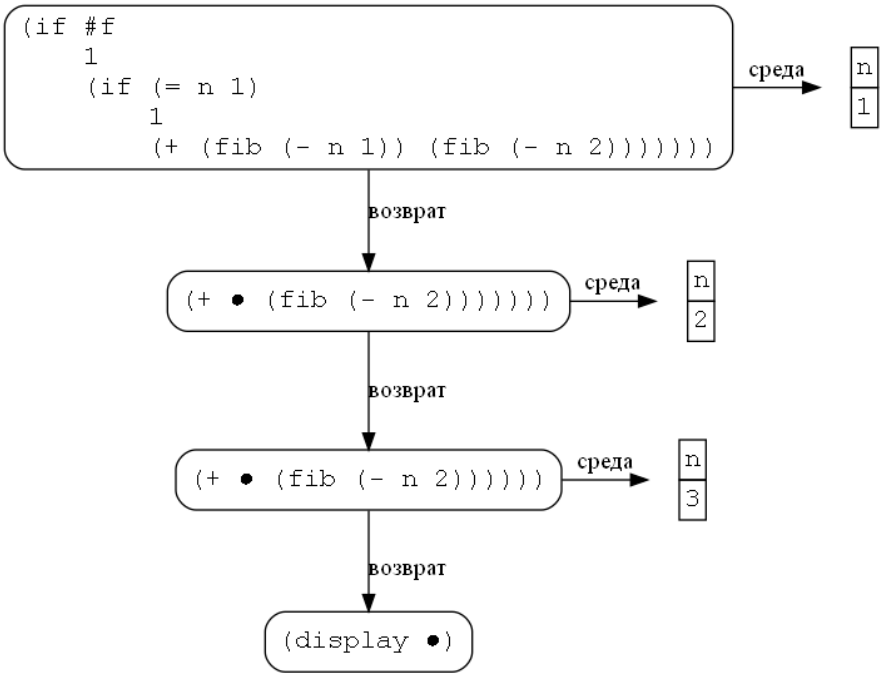
Вычисление `(- n 1)` → `(- 2 1)` → 1:



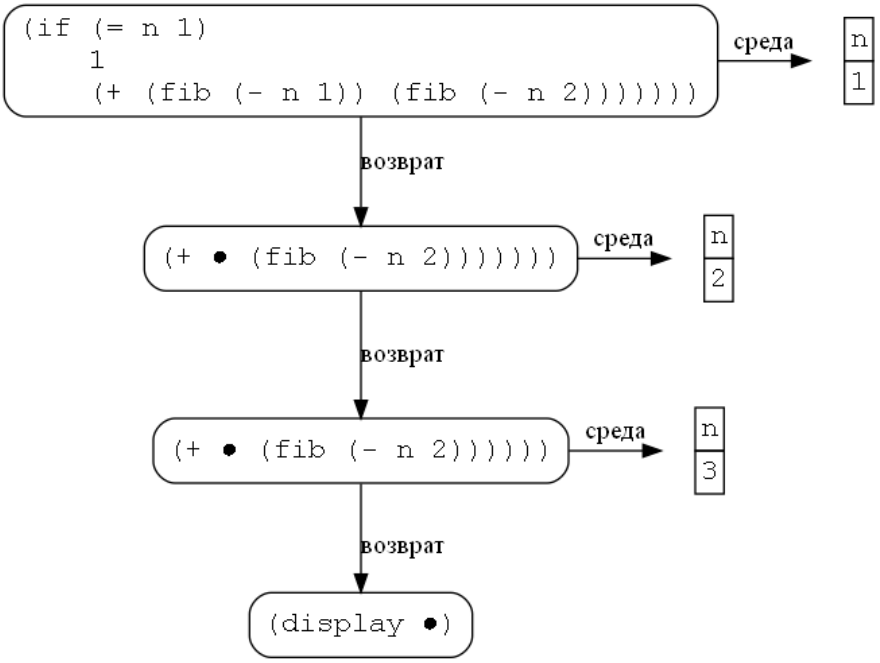
Рекурсивный вызов `fib`:



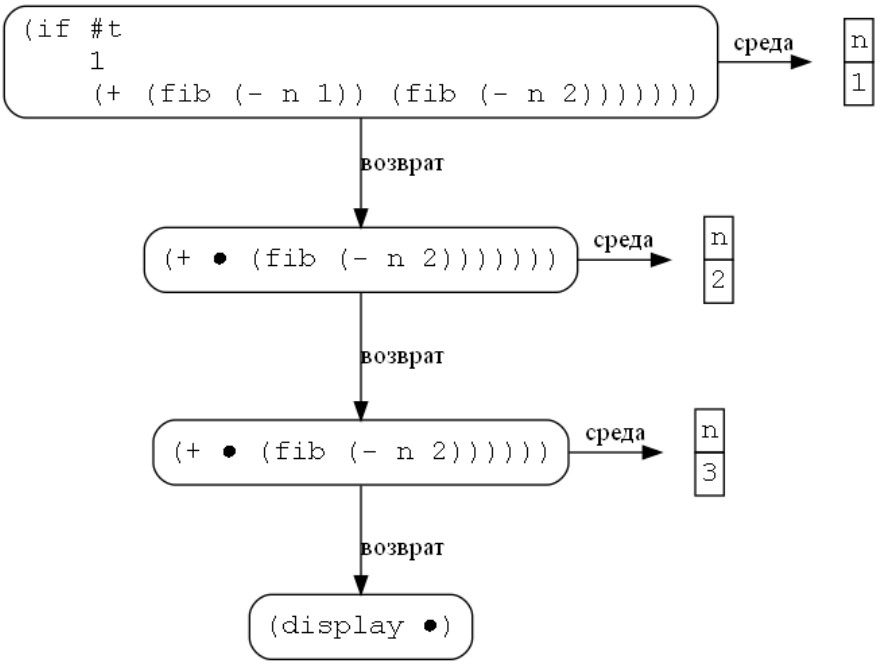
Вычисление `(= n 0)` → `(= 1 0)` → `#f`:



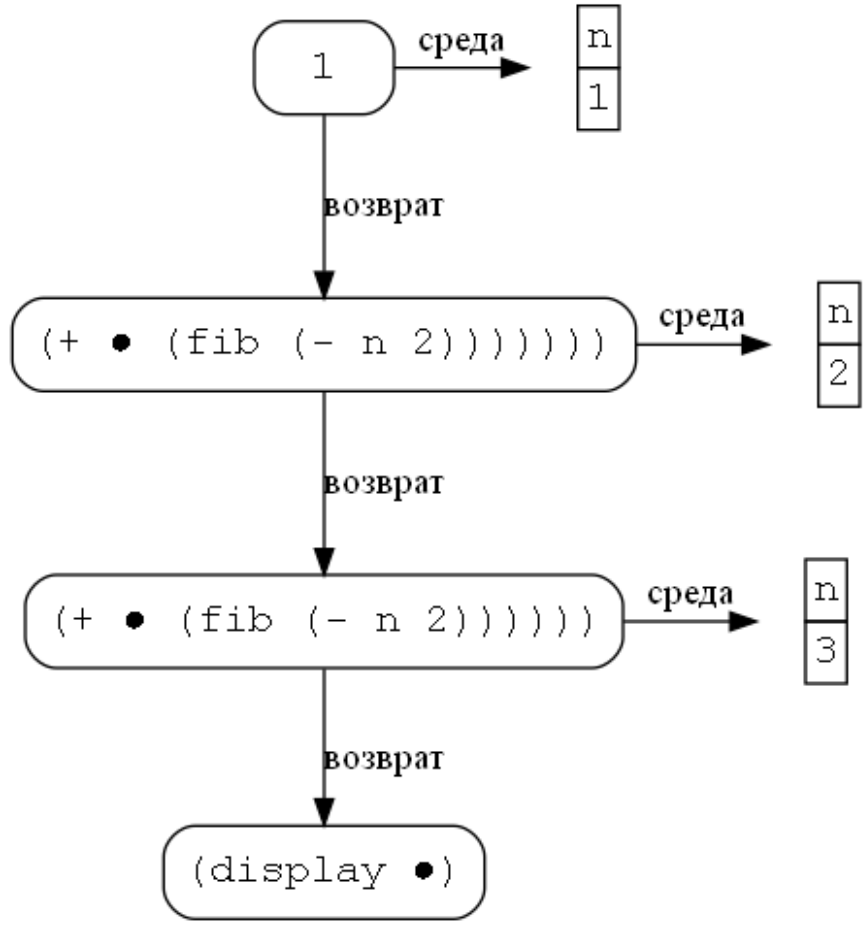
Редукция `if`:



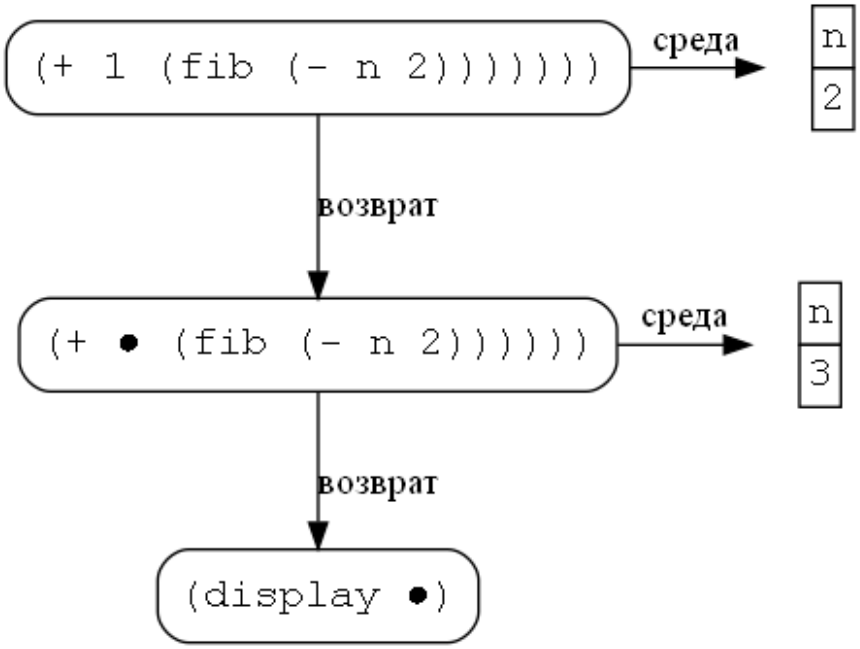
Вычисление `(= n 1)` → `(= 1 1)` → `#t`:


(if #t
1
(+ (fib (- n 1)) (fib (- n 2)))))))
среда
n
1
возврат
(+ • (fib (- n 2)))))))
среда
n
2
возврат
(+ • (fib (- n 2))))))
среда
n
3
возврат
(display •)

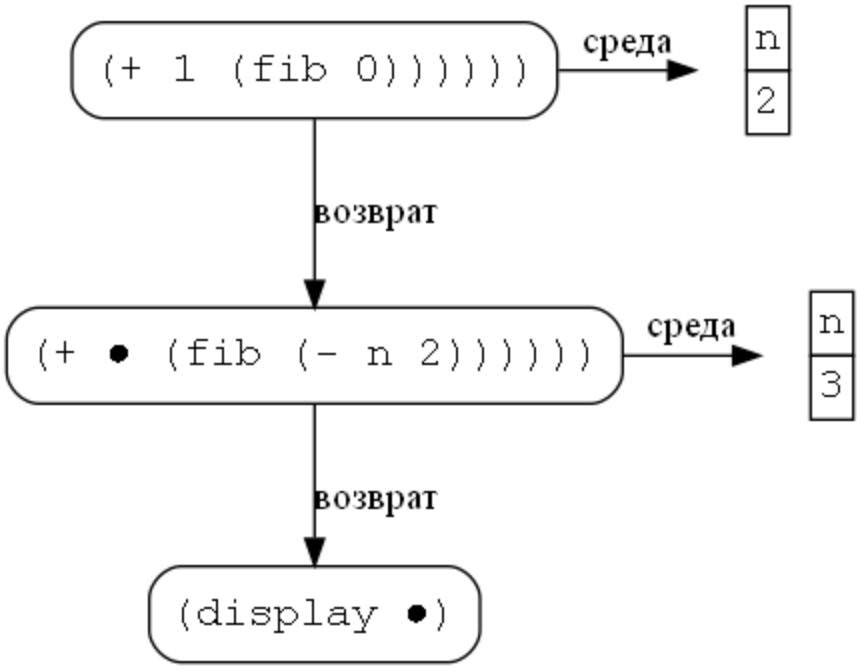
Редукция `if`:



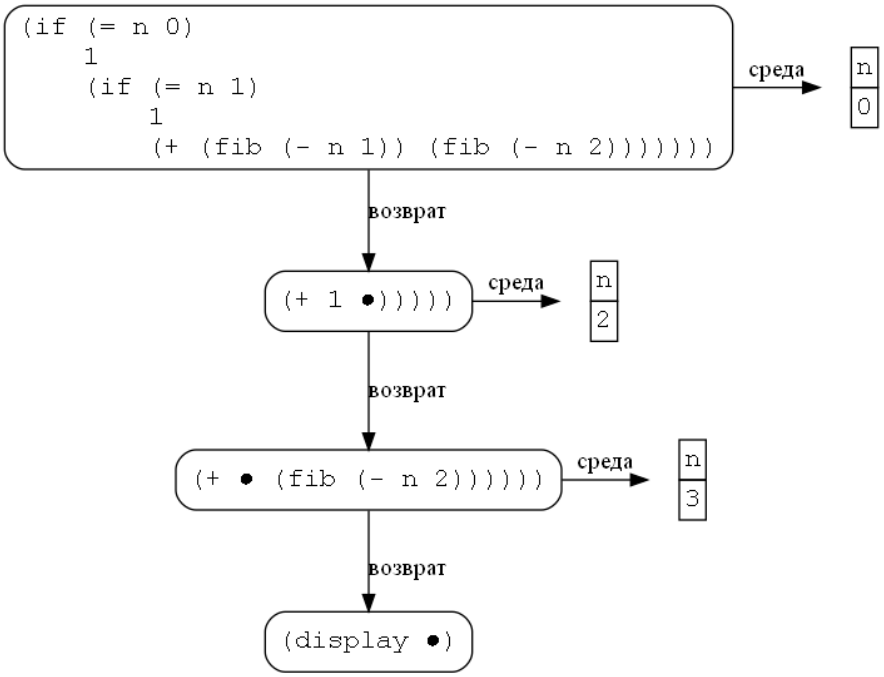
Возврат вычисленного значения:



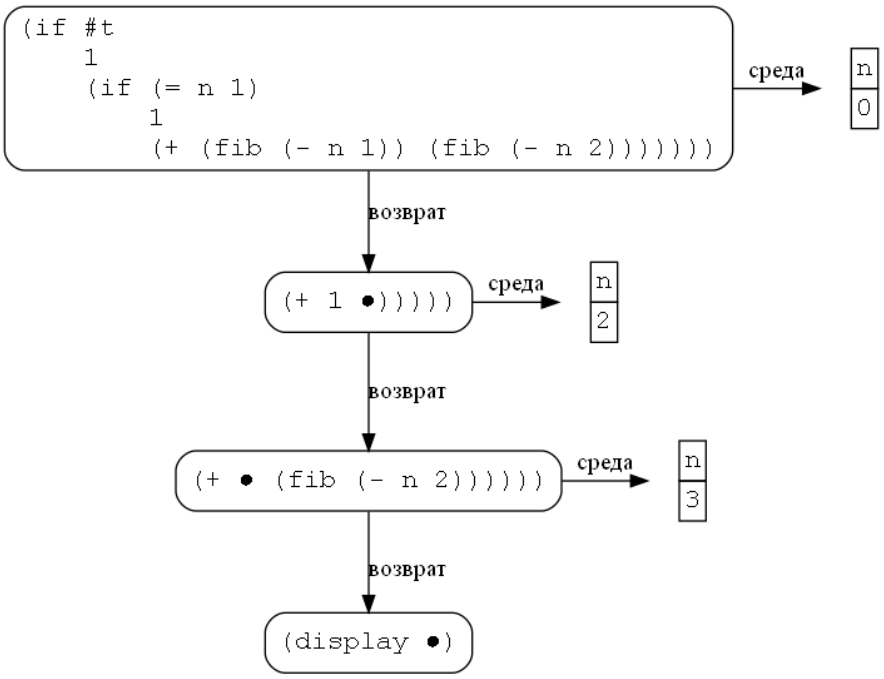
Вычисление `(- n 2)` → `(- 2 2)` → `0`:



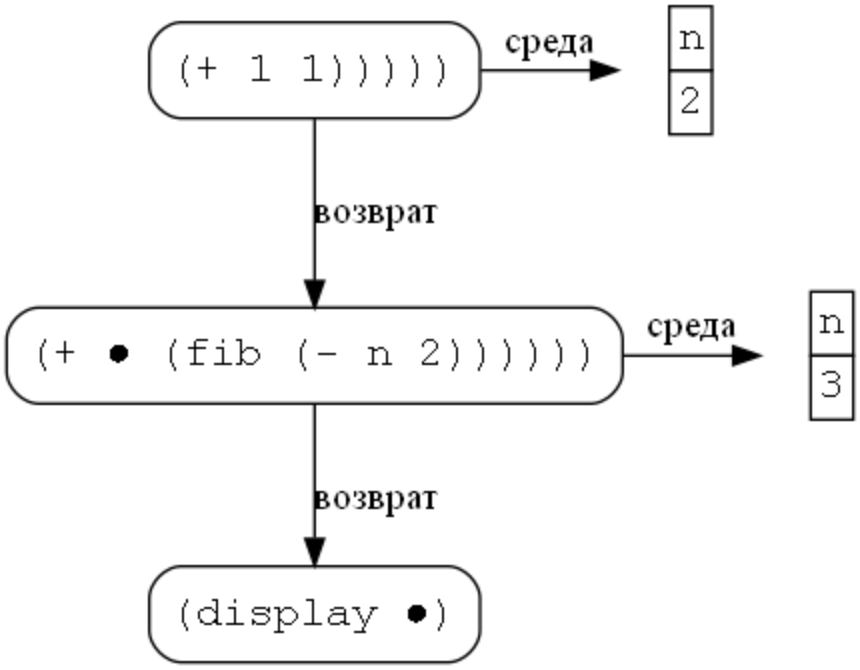
Рекурсивный вызов `fib`:



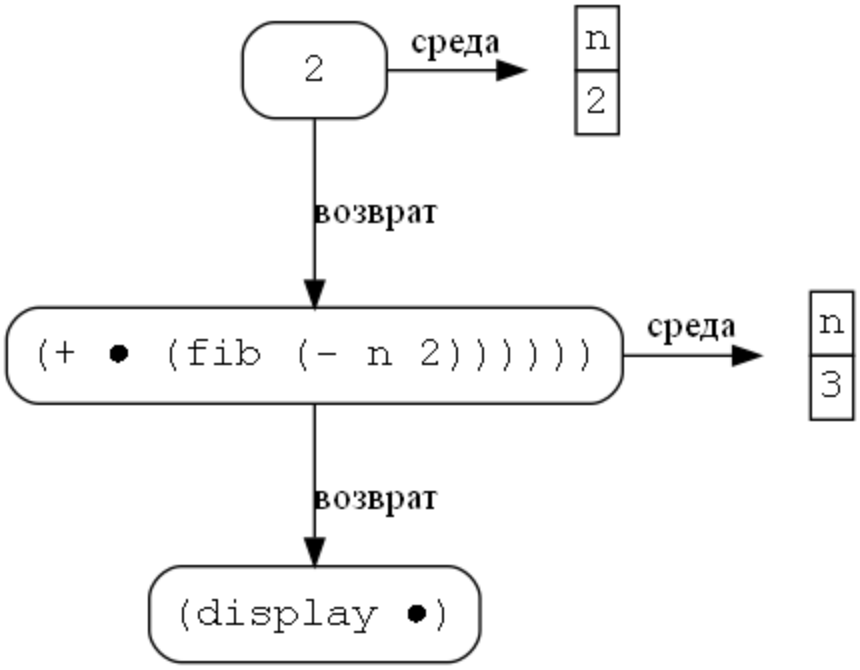
Вычисление `(= n 0)` → `(= 0 0)` → `#t`:



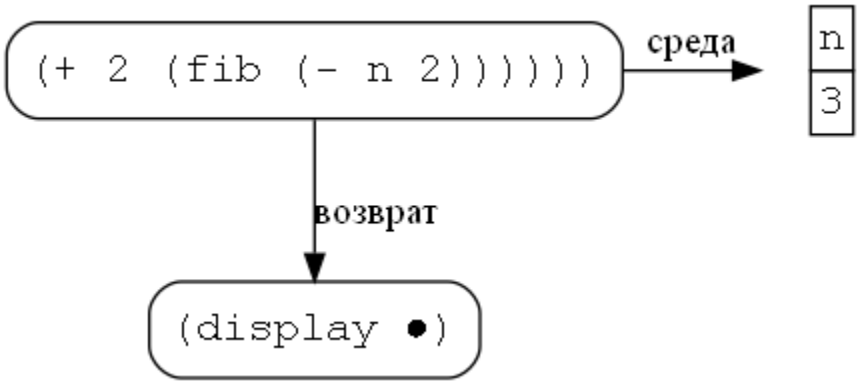
Редукция `#if`, возврат из функции:



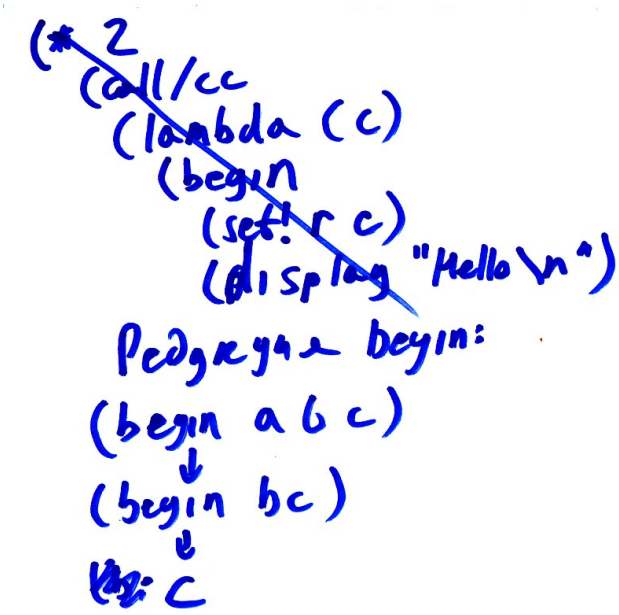
Вычисление (+ 1 1) → 2:



Возврат из функции:

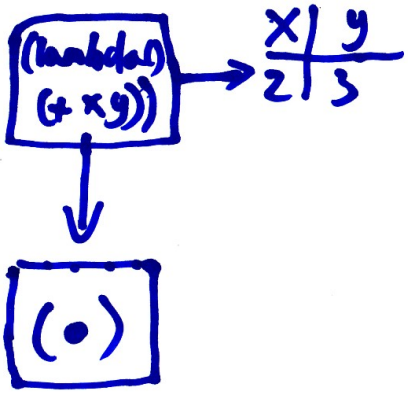


Рисунки «на доске»

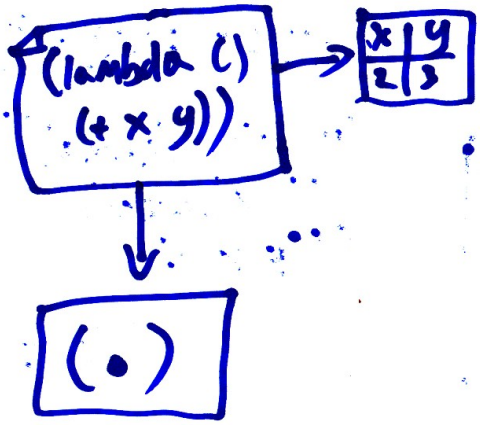


Замыкания

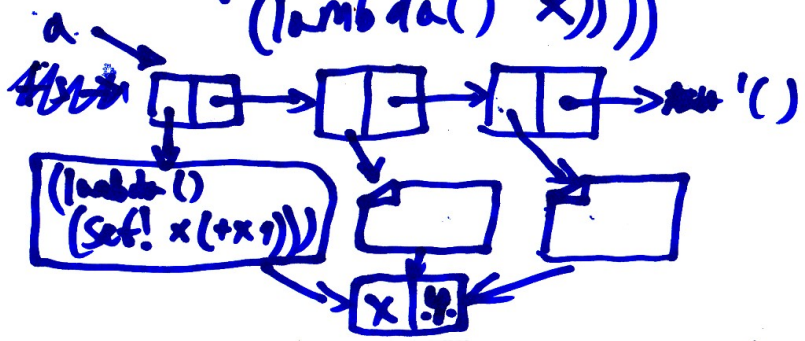
```
(((lambda (x y)
   (lambda ()
     (+ x y)))
 2 3))
```



Замыкания



```
(define fs
  (lambda (x)
    (list (lambda () (set! x (+ x 1)))
          (lambda () (set! x (* x 2)))
          (lambda () x))))
```




```
(define a (fs 4))
((caddr a)) -> 4
((car a))
((caddr a)) -> 5
((cadr a))
((caddr a)) -> 10
```

#<void>

```
(if #f #f)
```

```
int x;
if(x){
  printf("A");
}
if(!x){
  printf("B");
}
```

Продолжения. ~~call-with-current-continuation~~ call/cc

call-with-current-continuation

(define call/cc
  call-with-current-continuation)

Как вызывается:

(call/cc proc)

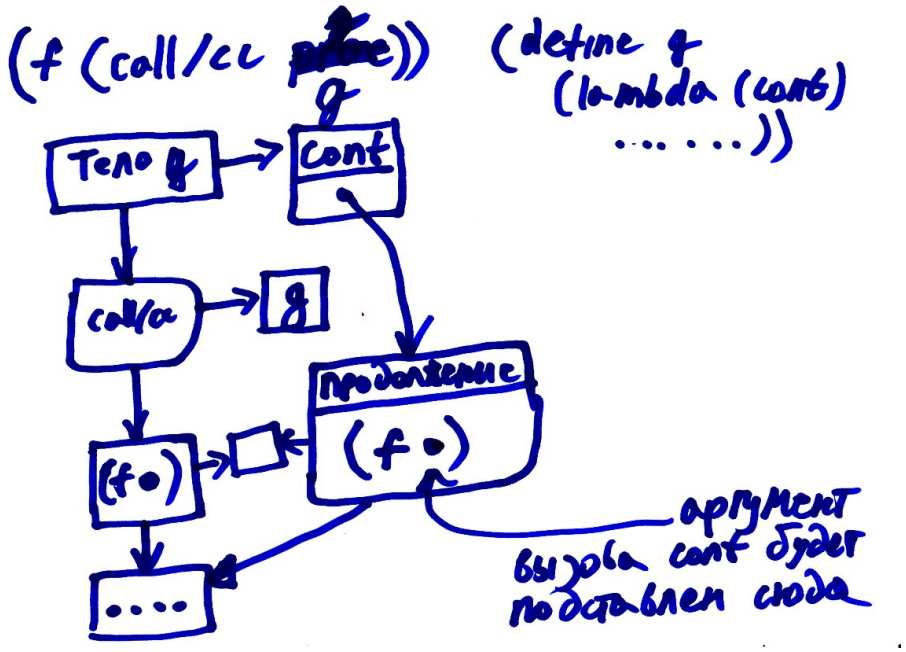
где proc – процедура.

proc вызывается с одним параметром-продолжением, которое также можно вызвать как процедуру:

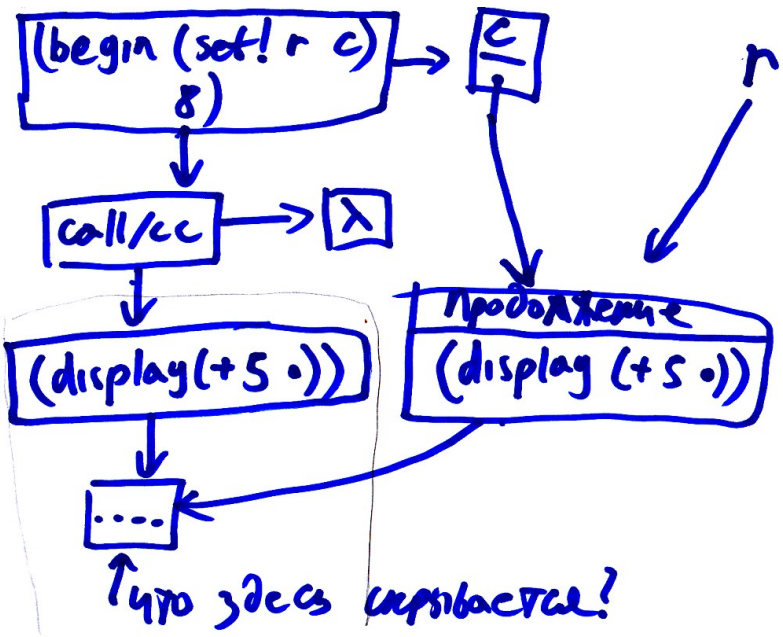
```
(call/cc
  (lambda (cont)
    ........
    .... (cont 5)
    .....
    ....))
```

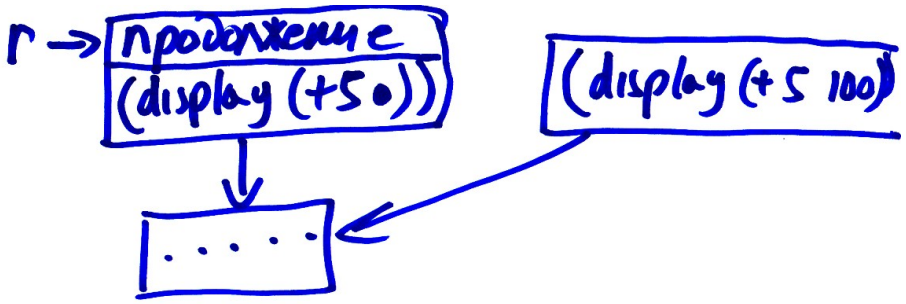
Если proc завершилась обычным образом-результат (call/cc proc) тот же, что и у (proc cont)

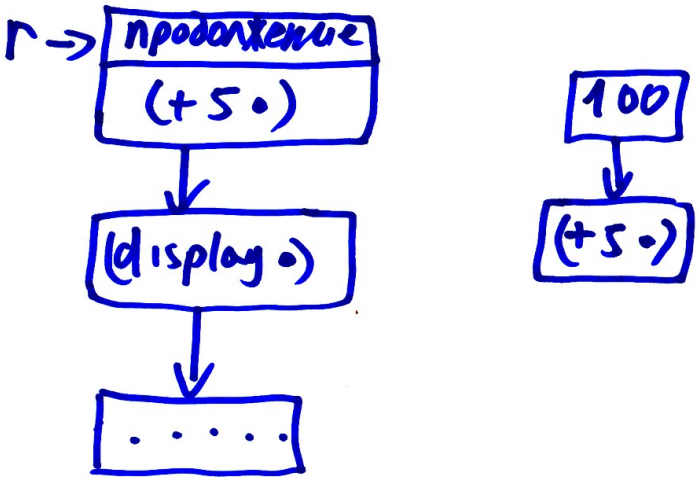
Другой случай – вызов cont (с одним параметром)
В этом случае восстанавливается фрейм стека на момент вызова call/cc.



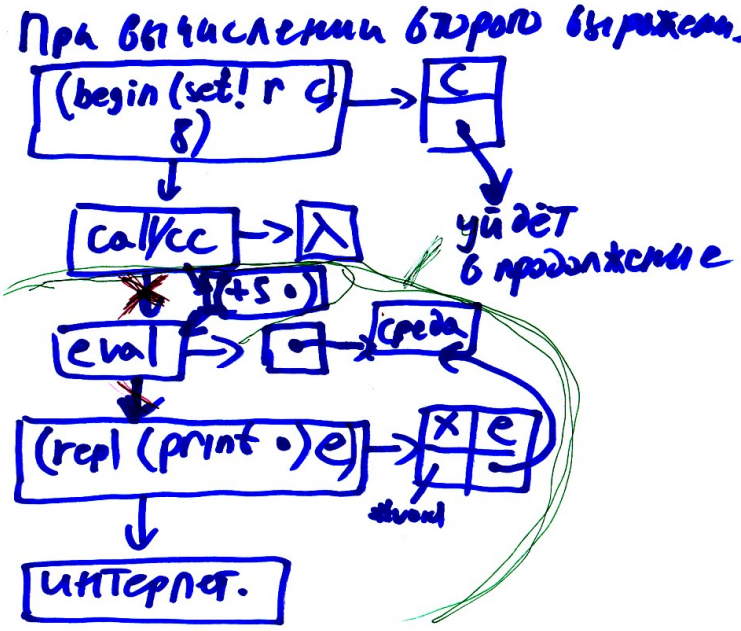
```
(define r #f)
(display (+ 5
            (call/cc
              (lambda (c)
                (set! r c)
                8))))   → 13

(r 100) → 105
```




r
продолжение
(display (+5 •))
(display (+5 100))
. . . . .


r
продолжение
(+5 •)
(display •)
. . . . .
100
(+5 •)

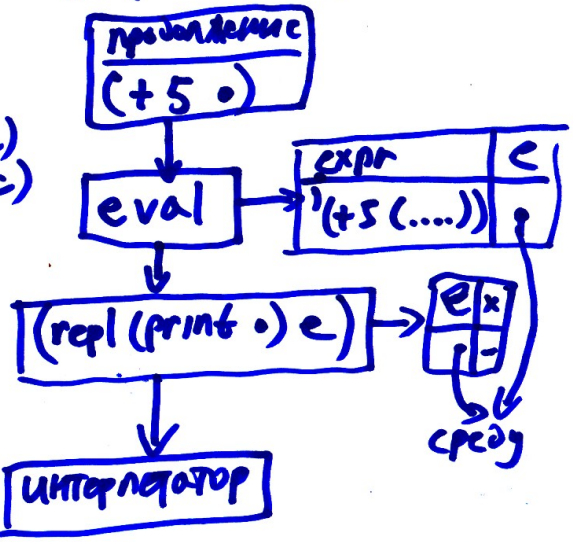
При вычислении второго выражения:



Входной файл

```
(define r #f)
(+5 (call/cc
      (lambda (c)
        (set! r c)
        8)))
(display 'Hello)
(newline)
(r 100)
```

В переменной r:



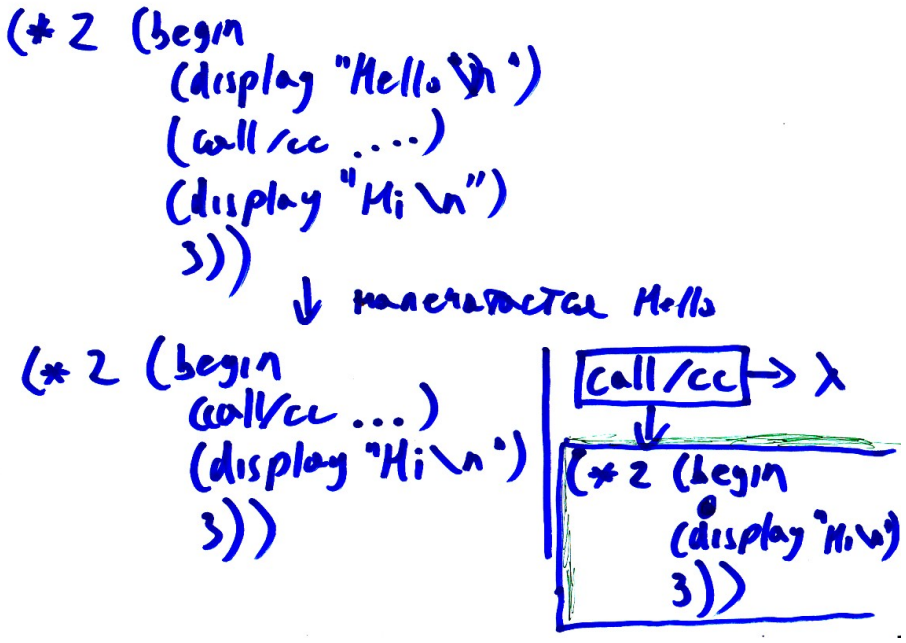
```
(define repl
  (lambda (x e)
    (repl (print (eval (read)
                       e))
          e)))
```

```
(*2 (begin
      (display "Hello\n")
      (call/cc
        (lambda (c)
          (set! r c)))
      (display "Hi\n")
      3))
```

Hello
Hi
→ 6

(r #f) → Hi
6

call/cc запомнил позицию вычисления после того, как Hello напечаталось



Лекция 11. Введение в трансляцию программ

Компьютер исполняет машинный код — последовательность примитивных операций: сложение двух ячеек памяти, пересылка значения из одной ячейки в другую, обращение к устройству, передача управления на другую инструкцию (безусловная или условная — в зависимости от результата предыдущей операции).

Человеку писать машинный код трудоёмко и муторно. Решение: программист пишет программу на человекочитаемом языке, компьютер её выполняет.

Способы реализации языка программирования: интерпретация и компиляция.

При **интерпретации** компьютер читает программу на языке программирования и выполняет инструкции, записанные в этой программе.

Преимущества: удобство разработки (нет промежуточной фазы компиляции), возможно, удобство отладки, переносимость.

Недостатки: низкое быстродействие, зависимость от интерпретатора.

При **компиляции** компьютер переводит программу с человекочитаемого языка на машинный язык. Транслятор — синоним компилятора.

Преимущества: высокое быстродействие, автономность готовых программ. Недостаток: фаза компиляции, зависимость готовых программ от платформы.

Гибридный подход: компилятор формирует промежуточный более низкоуровневый код, который затем выполняется интерпретатором.

Стадии компиляции:

1. **Лексический анализ программы** — программа делится на «слова» — **лексемы,** некоторые небольшие структурные элементы: знаки операций, идентификаторы, литеральные константы (числа, строки, символы…). На стадии лексического анализа отбрасываются комментарии и символы пустого пространства (пробелы, табуляции, переводы строк).

   Лексема — подстрока исходной программы: `)`, `counter`, `007`. **Токен** — «обработанная» лексема, токен состоит из метки типа, позиции в исходном файле и атрибута (значения лексемы): `('CLOSE-BRACKET (1 1))`, `('IDENT (2 1) "counter")`, `('NUMBER (2 10) 7)`.

2. **Синтаксический анализ** — принимает последовательность токенов и строит из них синтаксическое дерево. Последовательность токенов плоская, выход синтаксического анализатора иерархичен — отражает структуру программы.

3. **Семантический анализ** — проверяет допустимость операций, правильность имён переменных, функций…

4. **Генерация промежуточного представления.**

5. **Оптимизации.**

6. **Генерация машинного кода.**

7. **Компоновка.** Программа может состоять из отдельно транслируемых файлов, в том числе библиотек — нужно из них собрать единую готовую программу.

Стадии 1–3 — стадии анализа (front end), стадии 4–7 — стадии синтеза (back end).

Стадии интерпретации.

1. **Лексический анализ.**
2. **Синтаксический анализ.**
3. **Семантический анализ.**
4. **Исполнение** (интерпретация) построенной программы.

Лекция 12. Вычисления на стеке, конкатенативное программирование

Конкатенативное программирование — парадигма программирования, в которой композиция функций выражается через конкатенацию строк. Примеры языков: FORTH, Joy, Factor.

Т.е. пусть у нас есть две программы `P1` и `P2`, конкатенация этих двух программ `P1 P2` будет выражать применение `P2` к результату выполнения `P1`.

В конкатенативных языках явных переменных нет, данные передаются неявно, через стек. Программа представляет собой последовательность операторов, каждый из которых выполняет какую либо операцию со стеком. Частный случай: константы — это тоже операторы, которые на стек кладут соответствующее значение.

В конкатенативных языках программирования принято записывать стек, растущий слева направо, причём верхушка стека расположена справа.

Действия операторов принято записывать как

```
оператор : ... стек до  =>  ... стек после
```

Например:

```
+ : ... x y  =>  ... (x+y)
```

Программа пишется в обратной польской записи или постфиксной записи: сначала записываются операнды, а потом сама операция.

Местность (арность) операции, функции — количество операндов (аргументов) у неё.

Коместность (коарность) — количество возвращаемых значений.

Поскольку местность каждой операции фиксирована, скобки не нужны.

Пример.

Выражение в обычной (инфиксной записи):

```
(2 - 1) * (3 + 4)
```

Выражение в постфиксной записи (обратной польской):

```
2 1 - 3 4 + *
```

- Аргументы операции `-`: 2 и 1,
- аргументы операции `+`: 3 и 4,
- аргументы `*`: `2 1 -`, `3 4 +`.

Можно добавить скобки для наглядности, но их никто не ставит:

```
((2 1 -) (3 4 +) *)
```

Местность констант 0 (не принимают аргументов), коместность — 1.

```
константа : ...  =>  ... константа
```

Обратная польская запись допускает простую и эффективную реализацию:

- в цикле читаем очередную операцию,
- снимаем со стека соответствующее количество аргументов операции,
- выполняем операцию,
- кладём на стек её результаты.

Пример:

```
Стек                Программа

...                 2 1 - 3 4 + *
                    ↑
... 2               2 1 - 3 4 + *
                      ↑
... 2 1             2 1 - 3 4 + *
                        ↑
... 1               2 1 - 3 4 + *
                          ↑
... 1 3             2 1 - 3 4 + *
                            ↑
... 1 3 4           2 1 - 3 4 + *
                              ↑
... 1 7             2 1 - 3 4 + *
                                ↑
... 7               2 1 - 3 4 + *
                                  ↑
программа завершилась
```

Как выполняются вызовы функций в стековом языке программирования

Функции в FORTH принято называть **статьями,** хранилище функций — **словарём.**

Программа на языке FORTH состоит из последовательности **слов,** словом может быть или целочисленная константа, или некоторое имя. Часть слов предопределены (встроены в язык), часть определяются пользователем в виде статей:

Определение статьи выглядит так

```
 : ИМЯ  слова…  ;
```

Знак `:` начинает определение, знак `;` — заканчивает.

Для вызовов функций вводится второй стек — **стек возвратов.** В основном стеке, **стеке данных** находятся значения, которыми обмениваются операции, в классическом FORTH'е это целые числа. В стеке возвратов хранятся адреса команд в словарных статьях.

Интерпретатор работает в следующем цикле:

- Если в статье слова не кончились, читается очередное слово.
  - Если слово есть в словаре, адрес следующего слова кладётся на стек возвратов, управление передаётся на первое слово словарной статьи.
  - Если нет в словаре и слово является записью целого числа, то число кладётся на стек данных.
  - Если слова нет в словаре и оно не является записью числа — ОШИБКА.
- Если слова в статье кончились — со стека возвратов снимается адрес следующего слова и передаётся на него управление.

Некоторые встроенные слова FORTH:

- Арифметика: `+`, `-`, `*`, `/`.
- Слова работы со стеком:

```
DUP :  ... x      =>  ... x x    -- дублирует верхушку стека
DROP : ... x      =>  ...        -- удаляет слово с верхушки стека
SWAP : ... x y    =>  ... y x    -- обменивает местами два слова на верхушке
ROT :  ... x y z  =>  ... y z x  -- поднимает на верхушку третий по счёту элем
OVER : ... x y    =>  ... x y x  -- копирует подвершину на верхушку
```

- Управляющие конструкции
  - `IF ... THEN` — если на вершине не ноль, выполняются слова между `IF` и `THEN`, иначе ничего не делается. В обоих случаях число со стека снимается.
  - `IF ... ELSE ... THEN` — если на верхушке не ноль, он снимается с верхушки и выполняются слова между `IF` и `ELSE`, иначе ноль снимается с верхушки и выполняются слова между `ELSE` и `THEN`.
  - `WHILE ... WEND` — цикл с предусловием повторяется до тех пор, пока на верхушке не ноль.
- Ввод-вывод

```
. : ... x  =>  ...              -- печатает число, снимая его со стека
```

Пример программы на FORTH. Функция (слово) `hypot` вычисляет гипотенузу прямоугольного треугольника:

```
\ square : ... x  =>  ... x²
: square DUP * ;

\ hypot : ... x y  =>  ... √(x²+y²)
: hypot square SWAP square + SQRT ;
```

Выполнение слова `square`:

```
Стек             Программа
--------------------------
```

```
... x            DUP * ;
                 ↑
... x x          DUP * ;
                     ↑
... x²           DUP * ;
                       ↑
происходит возврат из square
```

Выполнение слова hypot:

```
Стек             Программа
-------------------------
... x y          square SWAP square + SQRT ;
                 ↑
```

Когда слово square вызывается, на стек возвратов кладётся указатель на слово SWAP в определении слова hypot.

```
Стек             Программа
-------------------------
... x y²         square SWAP square + SQRT ;
                        ↑
... y² x         square SWAP square + SQRT ;
                             ↑
... y² x²        square SWAP square + SQRT ;
                                   ↑
... y²+x²        square SWAP square + SQRT ;
                                     ↑
... √(y²+x²)     square SWAP square + SQRT ;
                                           ↑
происходит возврат из hypot
```

Пример, характерный для FORTH:

```
: 2 3 ;
2 2 * .
```

Она выведет 9, а не 4.

```
: + - ;
10 5 + .
```

Выведет 5, а не 15.

Слово с переменным числом параметров:

```
: SUM DUP WHILE + SWAP DUP WEND DROP ;
```

Сложит все числа на стеке до ближайшего нуля.

```
1 2 3 0 4 5 6 SUM   =>  1 2 3 15
===== ~~~~~~~           ===== ~~

1 2 3 0 4 5 6          DUP WHILE + SWAP DUP WEND DROP ;
                       ↑
1 2 3 0 4 5 6 6        DUP WHILE + SWAP DUP WEND DROP ;
```

```
                          ↑
1 2 3 0 4 5 6          DUP WHILE + SWAP DUP WEND DROP ;
                              ↑
1 2 3 0 4 11           DUP WHILE + SWAP DUP WEND DROP ;
                                ↑
1 2 3 0 11 4           DUP WHILE + SWAP DUP WEND DROP ;
                                    ↑
1 2 3 0 11 4 4         DUP WHILE + SWAP DUP WEND DROP ;
                                        ↑
1 2 3 0 11 4 4         DUP WHILE + SWAP DUP WEND DROP ;
                          ↑
1 2 3 0 11 4           DUP WHILE + SWAP DUP WEND DROP ;
                              ↑
1 2 3 0 15             DUP WHILE + SWAP DUP WEND DROP ;
                                ↑
1 2 3 15 0             DUP WHILE + SWAP DUP WEND DROP ;
                                    ↑
1 2 3 15 0 0           DUP WHILE + SWAP DUP WEND DROP ;
                                        ↑
1 2 3 15 0 0           DUP WHILE + SWAP DUP WEND DROP ;
                          ↑
1 2 3 15 0             DUP WHILE + SWAP DUP WEND DROP ;
                                             ↑
1 2 3 15               DUP WHILE + SWAP DUP WEND DROP ;
                                                 ↑
```

Лекция 13. Основы лексического и синтаксического анализа

Формальная грамматика — это способ описания синтаксиса языков программирования.

Мы будем рассматривать не все языки программирования, а только контекстно-свободные и регулярные (автоматные).

Формальная грамматика — набор правил, позволяющих породить строку, принадлежащую данному языку программирования. Формальная грамматика состоит из

- аксиомы,
- множества терминальных символов,
- множества нетерминальных символов и
- множества правил грамматики.

Терминальные символы — символы алфавита, из которых строятся строки данного языка программирования. Для стадии лексического анализа (грамматики токенов) терминальные символы — литеры (characters) текста. Для стадии синтаксического анализа терминальные символы — токены.

Нетерминальные символы — символы, которые раскрываются согласно правилам грамматики.

В синтаксическом дереве (дереве разбора) терминальные символы соответствуют листьям, нетерминальные — внутренним узлам.

Аксиома грамматики — нетерминальный символ, выбранный в качестве стартового.

Правила грамматики описывают, как в строке символов (терминальных и нетерминальных) раскрываются нетерминальные символы.

Порождение строки, принадлжежащей языку, начинается с аксиомы, заканчивается цепочкой терминальных символов.

Контекстно-свободные языки — языки, правила грамматик которых описываются выражением вида

`X → a b c …`

где `X` — нетерминальный символ, `a`, `b`, `c` — некоторые символы грамматики.

При описании грамматик для наглядности пустую строку обозначают знаком ε.

Грамматика арифметических выражений:

- аксиома `E`,
- терминальные символы: `+`, `*`, `n` (некоторое число), `(`, `)`.
- нетерминальные символы: `E`, `T`, `F`.
- правила
  ```
  E → T
  E → E + T
  T → F
  T → T * F
  F → n
  F → (E)
  ```

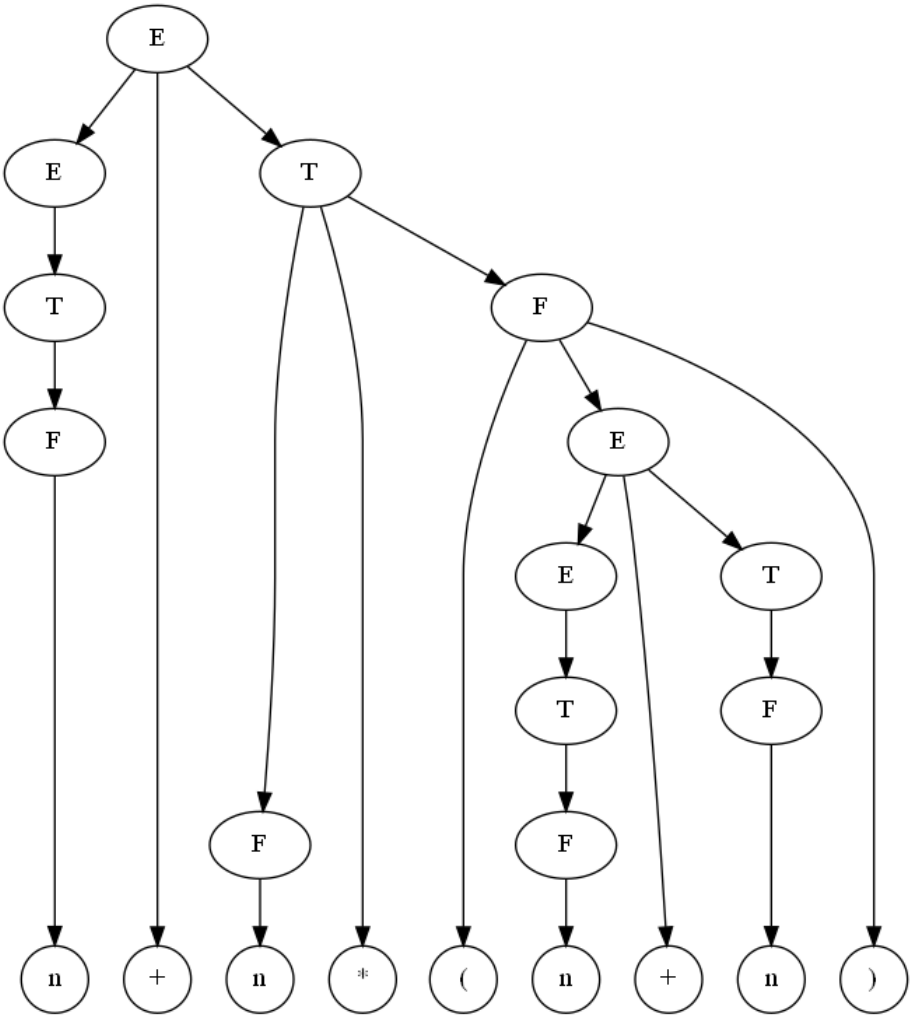
Часто правила с общими левыми частями (нетерминалами) объединяют, разделяя варианты знаком `|`.

```
E → T | E + T
T → F | T * F
F → n | ( E )
```

Пример вывода в этой грамматике:

```
E  →  E + T  →  E + T * F  →  T + T * F  →  F + T * F  →
↑        ↑     ↑              ↑             ↑
   →  n + T * F  →  n + T * (E)  →  n + F * (E)  →  n + n * (E)  →
              ↑         ↑               ↑                   ↑
   →  n + n * (E + T)  →  n + n * (T + T)  →  n + n * (F + T)  →
               ↑                  ↑                  ↑
   →  n + n * (n + T)  →  n + n * (n + F)  →  n + n * (n + n)
                  ↑                  ↑
```

Дерево вывода для этого примера:



Задача синтаксического анализа — имеем цепочку нетерминальных символов, нужно построить дерево разбора, соответствующее этой цепочке или показать, что такового не существует — цепочка не принадлежит данному языку. Т.е. для некоторой грамматики и некоторой строки определить, принадлежит ли данная строка языку, описываемому данной грамматикой.

Способы описания грамматики

Форма Бекуса-Наура (БНФ) — способ описания грамматики, где правила имеют вид

```
<Нетерминал> ::= альтернатива | … | альтернатива
```

нетерминалы записываются в угловых скобках (<…>), терминальные символы записываются или сами собой (для знаков операций,

например), или словами БОЛЬШИМИ БУКВАМИ. Альтернативные варианты разделяются знаками |.

Пример:

```
<Выражение> ::= <Слагаемое> | <Выражение> + <Слагаемое>
<Слагаемое> ::= <Множитель> | <Слагаемое> * <Множитель>
<Множитель> ::= ЧИСЛО | ( <Выражение> )
```

Впервые она была использована при описании Алгола-60.

При описании многих языков программирования (в учебниках, стандартах) используется тот или иной вариант БНФ. Нотация может быть расширена такими обзначениями как * или + после нетерминала, означающие повторение ноль или более раз (*) или один или более раз (+) данного нетерминала.

Как правило, если записана грамматика языка программирования, то под аксиомой подразумевается самый первый нетерминал (в примере <Выражение>).

Синтаксические диаграммы — это графический способ описания грамматики языка. Впервые был использован Никлаусом Виртом при описании синтаксиса языка Паскаль в 1973 году.

Используются редко, т.к. грамматика в виде БНФ более компактная и её проще записывать (диаграммы нужно рисовать).

LL(1)-грамматики

LL(k)-грамматики — грамматики, в которых мы можем определить правило для раскрытия нетерминала по первым k символам входной цепочки.

Дано: цепочка терминальных символов и нетерминальный символ. Требуется определить, по какому правилу нужно раскрыть нетерминальный символ, чтобы получить префикс этой цепочки. Для LL(k)-грамматик это можно сделать, зная первые k символов.

Чаще всего рассматриваются LL(1)-грамматики, где раскрытие определяется по первому символу.

Пример: не-LL(k)-грамматика:

```
E → T | E + T
T → F | T * F
F → n | ( E )
```

Если имеем строку `n * n * n + n + n` и нетерминал `E`, то мы не знаем, по какому правилу нужно раскрывать `E`. Поскольку в начале строки может быть сколько угодно сомножителей, в общем случае, чтобы выбрать правило раскрытия для `E` (т.е. `E → T` или `E → E + T`), нужно прочитать неизвестное количество входных знаков. А для LL(k)-грамматики k должно быть конечно и фиксировано.

Пример: LL(1)-грамматика для тех же арифметических выражений:

```
E → T E′
E′ → ε | + T E′
T → F T′
T′ → ε | * F T′
F → n | ( E )
```

здесь ε — пустая строка. В данной грамматике мы всегда можем определить применимое правило. Например, для E правило только одно, его используем. Для E′: если строка начинается на +, то выбираем вторую ветку E′ → + T E′, иначе выбираем первую E′ → ε. Для F: знак n выбирает первую ветку, знак (— вторую.

Пример не-LL(1)-грамматики:

```
A → B x z
B → ε | x y
```

Для строки x … и нетерминала B мы не можем определить раскрытие по первому символу, т.к. допустимо и то, и другое правило. Язык включает в себя две строки: x z и x y x z. По первому символу невозможно определить правило для B.

Однако, это грамматика LL(2). По первым двум символам определить раскрытие можно.

Также грамматика не LL(1) если разные правила начинаются с одинаковых символов:

```
A → x a | x b
```

Грамматика не LL(1), если в правилах имеем т.н. левую рекурсию:

```
A → x | A y
```

Преимущество LL(1)-грамматик — для них сравнительно легко написать синтаксический анализатор методом рекурсивного спуска.

Метод рекурсивного спуска

Метод рекурсивного спуска — способ написания синтаксических анализаторов для LL(1)-грамматик на алгоритмических языках программирования. Для каждого нетерминала грамматики записывается процедура, тело которой выводится из правил для данного нетерминала.

Построенный синтаксический анализатор выдаёт сообщение о принадлежности входной строки к заданному языку.

Написание синтаксического анализатора состоит из этапов:

1. Составление LL(1)-грамматики для данного языка программирования.

2. Формальное выведение парсера из правил грамматики. Парсер либо молча принимает строку, либо выводит сообщение об ошибке.
3. Наполнение парсера семантическими действиями — построение дерева разбора, выполнение проверок на корректность типов операций, возможно даже, вычисление результата в процессе разбора.

Вспомогательная структура данных — поток (stream).

У потока есть возможность получить текущий символ (`peek stream`) без продвижения вперёд, получить символ и удалить из потока (`next stream`).

Пример реализации потока: stream.scm.

Все функции, соответствующие нетерминалам, принимают поток, первым символом которого должен быть первый символ раскрытия нетерминала, и функцию ошибки, которая вызывается, чтобы прервать разбор. При успешном разборе функция поглощает из входного потока все токены, соответствующие раскрытию данного нетерминала.

```
(define (nterm stream error)
  …)
```

Пусть `w` — некоторая строка символов грамматики (терминальных и нетерминальных). Обозначим `FIRST(w)` — множество терминальных символов, с которого может начинаться строка токенов, полученная из `w` раскрытием всех нетерминалов. Если строка может быть пустой, то `ε` также входит в `FIRST(w)`.

Построим множество `FIRST(w)` для правил грамматики арифметических выражений:

```
E → T E′
E′ → ε | + T E′
T → F T′
T′ → ε | * F T′
F → n | ( E )

FIRST(E) = FIRST(T) = FIRST(F) = { n, ( }
FIRST(E′) = { +, ε }
FIRST(+ T E′) = { + }
FIRST(* F T′) = { * }
FIRST(T′) = { *, ε }
…
```

Пусть правило имеет вид

```
nterm → alt1 | … | altN
```

Если среди альтернатив есть такая `altK`, что `ε ∈ FIRST[altK]`, то она должна быть последней.

Функция для нетерминала имеет вид, если FIRST[altN] не содержит ε:

```
(define (nterm stream error)
  (cond (( (peek stream) ∈ FIRST[alt1] )  PARSE[alt1] )
        …
        (( (peek stream) ∈ FIRST[altK] )  PARSE[altK] )
        …
        (( (peek stream) ∈ FIRST[altN] )  PARSE[altN] )
        (else (error #f))))
```

Если FIRST[altN] содержит ε, то функция имеет вид

```
(define (nterm stream error)
  (cond (( (peek stream) ∈ FIRST[alt1] )  PARSE[alt1] )
        …
        (( (peek stream) ∈ FIRST[altK] )  PARSE[altK] )
        …
        (( (peek stream) ∈ FIRST[altN–1] )  PARSE[altN–1] )
        (else PARSE[altN])))
```

Функция PARSE[w] описывает последовательность команд для разбора правой части правила w:

```
PARSE[] → #t
PARSE[term w] → (expect stream term error) PARSE[w]
PARSE[nterm w] → (nterm stream error) PARSE[w]
```

где функция (expect stream term error) имеет вид

```
(define (expect stream term error)
  (if (equal? (peek stream) term)
      (next stream)
      (error #f)))
```

Разбор начинается с создания потока и сохранения точки возврата. После успешного разбора аксиомы в потоке должен располагаться символ конца потока.

```
(define (parse tokens)
  ;; Создаём поток
  (define stream (make-stream tokens))

  ;; Создаём точку возврата
  (call-with-current-continuation
    (lambda (error)
      ;; разбираем аксиому
      (axiom stream error)
      ;; в конце должен остаться признак конца потока
      (equal? (peek stream) #f))))
```

Практические советы:

- В конец списка символов, которые потребляет лексический анализатор, в конец списка токенов, которые потребляет

синтаксический анализатор, нужно обязательно добавлять признак конца ввода (EOF, end-of-file).
- На практике язык разделяют на два «слоя»: лексику и синтаксис. Лексика языка определяет набор «слов», **лексем,** на которые бьётся программа, по лексемам создаются **токены,** которые группируются в синтаксическое дерево. Смысл в том, что раздельное описание лексики и синтаксиса гораздо проще, чем писать грамматику для всего сразу.
- Удобно определить тип «поток», как описано выше.
- Для прерывания разбора при ошибке рекомендуется использовать продолжение.

Лексический анализ

Грамматика для стадии лексического анализа описывается, как правило, без рекурсии (имеется ввиду, без нехвостовой рекурсии), т.к. лексическая структура языка не требует вложенных конструкций.

Назначение лексического анализа: разбивает исходный текст на последовательность токенов, которые синтаксический анализ будет группировать в дерево. Либо, если исходный текст не соответствует грамматике — выдача сообщения (сообщений) об ошибке.

Входные данные: строка символов (или список символов), выходные: последовательность токенов. Можно сказать, что дерево разбора для грамматики лексем вырожденное — рекурсия есть только по правой ветке (`cdr`).

Мы будем его реализовывать методом рекурсивного спуска.

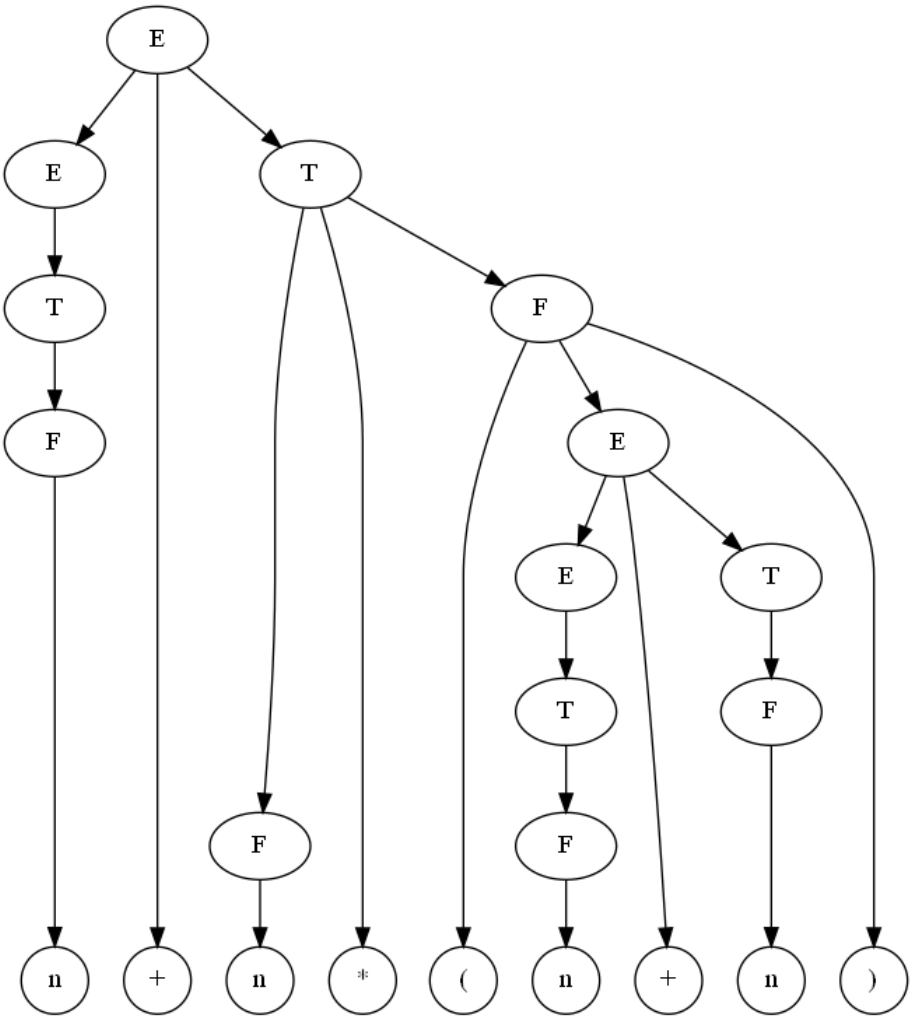
Синтаксический анализ

Его грамматика как правило описывается уже с использованием рекурсии, дерево разбора рекурсивное.

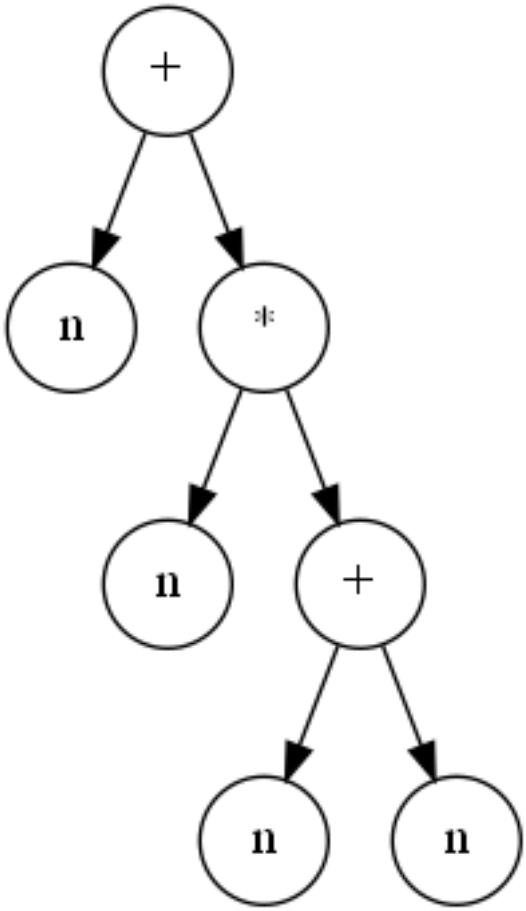
Назначение: построение синтаксического дерева из списка токенов. Либо выдача сообщения об ошибке.

Входные данные: список токенов, выходные: дерево разбора (или синтаксическое дерево).

Дерево разбора — дерево, построенное для данной грамматики и данной входной строки, такое что, корнем является аксиома грамматики, листьями — символы входной строки, внутренними узлами являются нетерминальные символы грамматики, потомки внутренних узлов упорядочены и соответствуют правилам грамматики, при перечислении листьев слева-направо получаем исходную строку.



Абстрактное синтаксическое дерево отражает уже логическую, смысловую структуру программы. В отличие от дерева разбора, оно не содержит скобок для указания приоритета, символов-разделителей и т.д.



Для построения абстрактного синтаксического дерева функцию `PARSE[w]` можно модифицировать следующим образом:

```
PARSE[s1 s2 ... sN] →
  (let* ((sym1 PARSE-SYM[s1])
         (sym2 PARSE-SYM[s2])
         ...
         (symN PARSE-SYM[sN]))
    ‹построение узла дерева из sym1…symN›)

PARSE-SYM[term] → (expect stream term error)
PARSE-SYM[nterm] → (nterm stream error)
```

Разумеется, в реальной программе переменным `sym1` … `symN` нужно давать осмысленные имена.

Пример лексического и синтаксического анализа

В качестве примера рассмотрим подмножество Scheme с переменными, лямбдами, определениями и вызовами функций.

Первая фаза написание парсера — построение LL(1)-грамматики.

```
(load "stream.scm")

; Подмножество языка Scheme
; <sequence> ::= <term> <sequence> | <empty>
; <empty> ::=
; <term> ::= <define> | <expr>
; <define> ::= (DEFINE VAR <expr>)
; <expr> ::= VAR | <lambda> | <call>
; <lambda> ::= (LAMBDA <varlist> <sequence>)
; <call> ::= (<expr> <exprs>)
; <exprs> ::= <empty> | <expr> <exprs>
;
; Лексика:
; <tokens> ::= <token> <tokens>
;            | <spaces> <tokens>
;            | <empty>
; <spaces> ::= SPACE <spaces> | <empty>
; <token> ::= "DEFINE" | "LAMBDA" | "(" | ")" | <variable>
; <variable> ::= LETTER | <variable> LETTER | <variable> DIGIT
```

Проблема этой грамматики, что она не LL(1).

Грамматика, приведённая к LL(1):

```
; <sequence> ::= <term> <sequence> | <empty>
; <empty> ::=
; <term> ::= <define> | <expr>
; <define> ::= (DEFINE VAR <expr>)
; <expr> ::= VAR | (<complex-const>)
; <complex-constr> ::= <lambda> | <call>
; <lambda> ::= LAMBDA <varlist> <sequence>
; <call> ::= <expr> <exprs>
; <exprs> ::= <empty> | <expr> <exprs>
;
; Лексика:
; <tokens> ::= <token> <tokens>
;            | <spaces> <tokens>
;            | <empty>
; <spaces> ::= SPACE <spaces> | <empty>
; <token> ::= "(" | ")" | <variable-or-keyword>
; <variable-or-keyword> ::= LETTER <variable-tail>
; <variable-tail> ::= <empty>
;                   | LETTER <variable-tail>
;                   | DIGIT <variable-tail>
```

Вторая фаза — механистическое построение парсера по грамматике. Построим лексический анализатор:

```
(define (scan str)
  (define stream
```

```
    (make-stream (string->list str) (integer->char 0)))

  (call-with-current-continuation
   (lambda (error)
     (tokens stream error)))
  (equal? (peek stream) (integer->char 0)))


; <tokens> ::= <spaces> <tokens>
;            | <token> <tokens>
;            | <empty>
(define (tokens stream error)
  (define (start-token? char)
    (or (char-letter? char)
        (char-digit? char)
        (equal? char #\()
        (equal? char #\))))
  (cond ((char-whitespace? (peek stream))
         (spaces stream error)
         (tokens stream error))
        ((start-token? (peek stream))
         (token stream error)
         (tokens stream error))
        (else #t)))

; <spaces> ::= SPACE <spaces> | <empty>
(define (spaces stream error)
  (cond ((char-whitespace? (peek stream))
         ;(if (char-whitespace? (peek stream))
         ;    (next stream)
         ;    (error #f))
         (next stream))
        (else #t)))

(define char-letter? char-alphabetic?)
(define char-digit? char-numeric?)


; <token> ::= "(" | ")" | <variable-or-keyword>
(define (token stream error)
  (cond ((equal? (peek stream) #\() (next stream))
        ((equal? (peek stream) #\)) (next stream))
        ((char-letter? (peek stream))
         (variable-or-keyword stream error))
        (else (error #f))))

; <variable-or-keyword> ::= LETTER <variable-tail>
(define (variable-or-keyword stream error)
  (cond ((char-letter? (peek stream))
         ;(if (char-letter? (peek stream))
```

```
         ;    (next stream)
         ;    (error #f))
         (next stream)
         (variable-tail stream error))
        (else (error #f))))

; <variable-tail> ::= LETTER <variable-tail>
;                   | DIGIT <variable-tail>
;                   | <empty>
(define (variable-tail stream error)
  (cond ((char-letter? (peek stream))
         (next stream)
         (variable-tail stream error))
        ((char-digit? (peek stream))
         (next stream)
         (variable-tail stream error))
        (else #t)))
```

Третья фаза — реализация семантических действий. В случае лексического анализа — это построение цепочки токенов.

```
(define (scan str)
  (let* ((EOF (integer->char 0))
         (stream (make-stream (string->list str) EOF)))

    (call-with-current-continuation
     (lambda (error)
       (define result (tokens stream error))
       (and (equal? (peek stream) EOF)
            result)))))


; <tokens> ::= <spaces> <tokens>
;            | <token> <tokens>
;            | <empty>
;
; (tokens stream error) -> list of tokens
(define (tokens stream error)
  (define (start-token? char)
    (or (char-letter? char)
        (char-digit? char)
        (equal? char #\()
        (equal? char #\))))

  (cond ((char-whitespace? (peek stream))
         (spaces stream error)
         (tokens stream error))
        ((start-token? (peek stream))
         (cons (token stream error)
               (tokens stream error)))
        (else '())))
```

```
; <spaces> ::= SPACE <spaces> | <empty>
;
; (spaces stream error) -> <void>
(define (spaces stream error)
  (cond ((char-whitespace? (peek stream))
         ;(if (char-whitespace? (peek stream))
         ;    (next stream)
         ;    (error #f))
         (next stream))
        (else #t)))

(define char-letter? char-alphabetic?)
(define char-digit? char-numeric?)


; <token> ::= "(" | ")" | <variable-or-keyword>
;
; (token stream error) -> token
(define (token stream error)
  (cond ((equal? (peek stream) #\() (next stream))
        ((equal? (peek stream) #\)) (next stream))
        ((char-letter? (peek stream))
         (variable-or-keyword stream error))
        (else (error #f))))

; <variable-or-keyword> ::= LETTER <variable-tail>
;
; (variable-or-keyword stream error) -> SYMBOL
(define (variable-or-keyword stream error)
  (cond ((char-letter? (peek stream))
         ;(if (char-letter? (peek stream))
         ;    (next stream)
         ;    (error #f))
         (string->symbol
          (list->string (cons (next stream)
                              (variable-tail stream error)))))
        (else (error #f))))

; <variable-tail> ::= LETTER <variable-tail>
;                   | DIGIT <variable-tail>
;                   | <empty>
;
; (variable-tail stream error) -> List of CHARs
(define (variable-tail stream error)
  (cond ((char-letter? (peek stream))
         (cons (next stream)
               (variable-tail stream error)))
        ((char-digit? (peek stream))
         (cons (next stream)
```

```
            (variable-tail stream error)))
        (else '())))
```

Полная реализация..

Лекция 14. Файловая система. Командные интерпретаторы

Файловая система — способ хранения информации в долговременной памяти компьютера (жёсткие диски, флешки, …) и соотвеющее API операционной системы.

API — application programming interface — интерфейс прикладного программирования. Так называют набор функций, посредством которых программист может взаимодействовать с операционной системой.

Файл — поименованная область диска. **Каталог, папка** — поименованная группа файлов. **Родительский каталог** для файла или папки — это папка, в которой находится данный файл или папка. **Корневой каталог** — каталог, у которого нет родительского каталога.

Путь к файлу — способ указания конкретного файла в файловой системе.

Далее мы будем рассматривать файловую систему UNIX-подобных операционных систем.

В отличие от Windows, в UNIX-подобных системах дерево файлов единое, т.е. содержимое отдельных устройств подключается в общее дерево папок как подпапки. (На Windows для каждого устройства выделяется отдельная буква диска.)

Для каждого процесса существует своего рода глобальная переменная — **текущая папка.** Как правило, текущая папка — это папка, которая была текущей в родительском процессе на момент запуска дочернего. Но процесс при желании может эту папку сменить.

Путь к файлу может быть **аблолютным** или **относительным.** Абсолютный путь к файлу указывается относительно корня операционной системы, относительный — относительно текущего каталога.

Имя файла в UNIX-подобных операционных системах может содержать любые знаки кроме знака `/` и знака с кодом `\0`. Символ с кодом `\0` запрещён, т.к. API для UNIX-подобных систем пишется на Си, а в Си этот символ является ограничителем строк. А знак `/` служит для разделения имён каталогов в пути к файлу.

По соглашению, имя файла может содержать точку, часть имени файла после точки определяет тип файла. Например, `example.c` —

исходный текст на Си, `index.html` — веб-страница. Эта часть имени файла называется «расширение» или «суффикс».

Абсолютный путь к файлу записывается, начиная со знака `/`:

`/папка/папка/…/имя-файла-или-папки`

Относительный путь к файлу не начинается со `/`:

`папка/папка/…/имя-файла-или-папки`

UNIX-подобная ОС — операционная система, реализующая стандарт POSIX. Примеры: Linux, macOS. На Windows имитируют окружение POSIX такие проекты как Cygwin и MinGW/MSys. В Windows 10 появилась подсистема WSL (Windows Subsystem for Linux), которая также имитирует окружение POSIX.

В путях можно использовать такие синонимы, как `.` и `...` Знак `.` является синонимом текущей папки, знак `..` — родительской папки. Можно считать, что в каждой папке находится папка `.`, которая является синонимом для неё же самой и папка `..`, которая является синонимом для родительской папки (ссылка на родительскую папку).

В корневой папке `..` ссылается на неё же саму.

Т.е., например, следующие пути будут эквиваленты:

```
/usr/bin/gcc
/usr/./././bin/./gcc
/var/log/../../usr/bin/gcc
/../../../../usr/bin/gcc
```

Путь `/var/log/..` ссылается на папку `/var`, т.к. `..` в `/var/log` ссылаются на родительскую для неё папку. `/var/log/../..` ссылается на корень.

Ссылки `.` и `..`, как правило, используются в относительных путях.

Оболочка операционной системы — это программа, которая позволяет пользователю взаимодействовать с операционной системой: запускать программы, работать с файлами. Оболочки бывают текстовыми и графическими, текстовые появились исторически раньше.

Командный процессор — текстовая оболочка операционной системы. Пользователь вводит команду, операционная система команду выполняет, выводит что-то на экран и ожидает следующей команды. Примерно как в REPL.

Исторически в UNIX первой оболочкой была оболочка, которая так и называлась, shell и располагалась по пути `/bin/sh`. Позже Борн создал оболочку Born Shell, затем был создан open source клон этой оболочки Born Again Shell — bash. Bash располагается по пути `/bin/bash`. Unix shell стандартизирован в POSIX.

Bash является расширением Unix shell, на большинстве дистрибутивов Linux `/bin/sh` является ссылкой на `/bin/bash`.

Bash отображает приглашение командной строки, как правило, включающее имя пользователя, имя компьютера, путь к текущей папке и знак привилегий: `$` для ограниченного пользователя, `#` для администратора (суперпользователя).

Знак `~` является сокращением для домашнего каталога пользователя, каталога вида `/home/username`.

В командной строке можно вводить как встроенные команды Bash, так и имена программ. Если имя программы не включает знак `/`, то исполнимый файл программы ищется в стандартных путях поиска, как правило, включающих `/bin` и `/usr/bin`. Для суперпользователя — также `/sbin` и `/usr/sbin`.

Если указан путь к программе, включающий `/` (относительный или абсолютный), то стандартные пути поиска не учитываются, запускается программа по заданному пути. Т.е. если в текущей папке лежит программа, то её приходится запускать как

```
./progname
```

Программы могут принимать аргументы командной строки. Т.е. после имени программы можно указать одно или несколько слов, эти слова запущенная программа может проанализировать и выполнить какие-либо действия:

```
./progname arg1 arg2 ...
```

Нулевым аргументом командной строки является само имя запущенной программы, последующие аргументы — те, что указаны пользователем.

Программа `example.c`:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("program arguments:\n");

  for (int i = 0; i < argc; ++i) {
    printf("[%d] = %s\n", i, argv[i]);
  }

  return 0;
}
```

Пример работы:

```
mazdaywik@Mazdaywik-NB10:~$ vim example.c
mazdaywik@Mazdaywik-NB10:~$ gcc example.c
mazdaywik@Mazdaywik-NB10:~$ ./a.out
program arguments:
[0] = ./a.out
```

```
mazdaywik@Mazdaywik-NB10:~$ ./a.out hello world
program arguments:
[0] = ./a.out
[1] = hello
[2] = world
mazdaywik@Mazdaywik-NB10:~$
```

Программы `vim` (текстовый редактор) и `gcc` (компилятор Си) получали в качестве аргумента имя файла, программа `a.out` (результат трансляции) — произвольные строки.

Список стандартных команд оболочки (встроенные команды и стандартные утилиты из `/bin`):

- `pwd` — вывести текущую папку.
- `cd имя-папки` — сменить текущую папку.
- `mkdir имя-папки` — создать папку.
- `rm файл...`, `rmdir папка...` — удаляет файлы и папки.
- `cp старый-файл новый-файл` — копирует файл.
- `cp файл... папка` — копирует несколько файлов в папку.
- `mv старый-файл новый-файл`, `mv файл... папка` — аналогично перемещает или переименовывает файлы и папки.
- `man команда` — показывает интерактивную справку по данной команде.
- `cat [файл...]` — распечатывает содержимое файлов на экран. Если имена файлов отсутствуют, то дублируется на экран ввод пользователя.
- `ls [папка]` — распечатывает содержимое папки на экран. По умолчанию — текущей папки.
- `clear` — очищает экран.
- `more [файл]` — вывод содержимого файла постранично, утилита POSIX.
- `less [файл]` — улучшенный вариант `more`, в POSIX не входит, но обычно есть.
- `tree [папка]` — вывод дерева папок указанной папки.
- `wc [файл...]` — подсчёт символов, слов и строк в указанных файлах.
- `echo строка` — вывод строки на экран.

Для аргументов командной строки существует соглашение, что параметры делятся на **ключи** и имена файлов. Ключи (опции) всегда начинаются на один или два знака `-`. Если аргумент не начинается с дефиса — он считается именем файла.

Ключи управляют режимом работы программы. Ключи, начинающиеся на `-`, как правило, однобуквенные, ключи на `--` записываются целым словом.

Например, команда

```
mkdir -p foo/bar/baz
```

создаст папки `foo`, `foo/bar` и `foo/bar/baz`, если их до этого не существовало. Без ключа `-p` программа выдаст ошибку, т.к. для

папки baz родительской папки foo/bar не существует.

У большинства команд (программ) есть ключ -h или --help, который отображает краткую справку. Не для все команд есть справка, выдаваемая через man.

Bash умеет раскрывать шаблоны имён файлов. Если среди аргументов присутствует аргумент со знаками * или ?, то он считается шаблоном и вместо него помещаются файлы, чьи имена соответствуют шаблону.

В шаблоне знак * означает произвольную последовательность знаков, ? — один знак.

Примеры: *.c — все файлы текущей папки с расширением .c, backups/2020-12-*.zip — архивы, датированные декабрём этого года из папки backups. Если в папке присутствуют файлы с расширениями .cpp и .cxx, то шаблон *.c?? выберет их все.

Пример раскрытия шаблона

```
mazdaywik@Mazdaywik-NB10:~$ ./a.out *.c*
program arguments:
[0] = ./a.out
[1] = example.c
[2] = hello.cpp
```

В текущей папке было только 2 файла, подпадающие под шаблон.

Для того, чтобы записать аргумент, например, с пробелами или какими-то другими знаками, которые интерпретируются в Bash, используются кавычки.

Двойные кавычки "..." допускают некоторую интерпретацию внутри них, например, раскрытие переменных или шаблонов. Одинарные '...' — трактуют содержимое буквально.

```
mazdaywik@Mazdaywik-NB10:~$ X=Foo
mazdaywik@Mazdaywik-NB10:~$ echo $X
Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out "Hello, $X"
program arguments:
[0] = ./a.out
[1] = Hello, Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out 'Hello, $X'
program arguments:
[0] = ./a.out
[1] = Hello, $X
mazdaywik@Mazdaywik-NB10:~$ ./a.out Hello, $X
program arguments:
[0] = ./a.out
[1] = Hello,
[2] = Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out '*.c*'
program arguments:
```

```
[0] = ./a.out
[1] = *.c*
```

«Философия Unix гласит:

- Пишите программы, которые делают что-то одно и делают это хорошо.
- Пишите программы, которые бы работали вместе.
- Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс».

Дуг Макилрой, изобретатель каналов Unix и один из основателей традиции Unix

Процесс — это экземпляр работающей программы.

Когда мы в Bash пишем команду, запускающую программу, запускается новый процесс, а сама оболочка ждёт его завершения. Но процесс можно запустить и в фоне:

```
$ ./program args &
```

Знак & означает, что процесс запущен в фоне. Список фоновых программ, запущенных в текущем сеансе, можно получить при помощи команды `jobs`, она выведет пронумерованные процессы. Команда `fg` переводит фоновый процесс на передний план.

Процесс может быть приостановлен (заморожен, поставлен на паузу). Постановка текущей выполняемой программы на паузу выполняется комбинацией клавиш CTRL-Z. Процесс в этом случае приостанавливается и уходит в фон.

Команда `fg` к приостановленному процессу его возобновляет и переводит на передний план. Команда `bg` — возобновляет и отправляет в фон.

Пример. Запустили архиватор, увидели, что он будет работать долго, решили послать его в фон:

```
mazdaywik@Mazdaywik-NB10:~$ tar czf archive.tar.gz *
^Z
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ bg
[1]+ tar czf archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]+  Запущен          tar czf archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ fg
tar czf archive.tar.gz *
^Z
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ tar czf second-archive.tar.gz * &
[2] 25
mazdaywik@Mazdaywik-NB10:~$ jobs
```

```
[1]+  Остановлен       tar czf archive.tar.gz *
[2]-  Запущен          tar czf second-archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ bg 1
[1]+ tar czf archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]-  Запущен          tar czf archive.tar.gz * &
[2]+  Запущен          tar czf second-archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$
```

Для прерывания процесса используется комбинация клавиш CTRL-C:

```
mazdaywik@Mazdaywik-NB10:~$ fg 1
tar czf archive.tar.gz *
^C
mazdaywik@Mazdaywik-NB10:~$ fg 2
tar czf second-archive.tar.gz *
^C
mazdaywik@Mazdaywik-NB10:~$ jobs
mazdaywik@Mazdaywik-NB10:~$
```

Процессы в unix-подобных системах идентифицируются по PID — целое число.

Для получения списка запущенных процессов используется команда `ps`, по умолчанию выводит список процессов текущего пользователя. Команда `ps aux` выводит все процессы в системе с выдачей подробных сведений.

Процессам можно посылать сигналы. Для посылки сигналов используется команда `kill`. Синтаксис

```
kill [-N] pid
```

где `-N` — номер сигнала. По умолчанию посылается сигнал `SIGTERM`. Сигнал `SIGTERM` — просьба процессу завершиться. Аналогичную роль играет `SIGINT`, он как раз посылается с клавиатуры комбинацией клавиш CTRL-C. Сигнал `SIGSTOP` посылается как CTRL-Z.

Список доступных сигналов с номерами:

```
kill -l
```

Сигнал `SIGKILL` — сигнал на безусловное прерывание программы, имеет код 9. Поэтому, чтобы жёстко убить процесс, нужно набрать

```
kill -9 pid
```

Если в программе произошла ошибка доступа к памяти (например, из-за неверного указателя), операционная система посылает процессу сигнал `SIGSEGV` (segmentation violation, segmentation fault, ошибка сегментации).

Процесс может иметь несколько открытых дескрипторов (небольшие целые числа), из которых он может читать данные, либо в них

писать. Обычно это дескрипторы открытых файлов или сетевых соединений.

Но есть 3 по умолчанию открытых дескриптора, которые соответствуют двум устройствам:

- Дескриптор `0` — чтение с клавиатуры.
- Дескриптор `1` — вывод на экран.
- Дескриптор `2` — вывод на экран.

Для того, чтобы ввести «конец файла» с клавиатуры, используется комбинация клавиш CTRL-D. На Windows «конец файла» вводится как CTRL-Z.

В языке Си тип `FILE*` — обёртка над дескрипторами ОС, обёртки над этими тремя дескрипторами доступны как константы `stdin`, `stdout` и `stderr`.

```
fprintf(stdout, "Hello!\n");
```

эквивалентно

```
printf("Hello!\n");
```

Оболочка `bash` может перенаправлять дескрипторы. Для запущенной программы можно связать `stdin`, `stdout` и `stderr` с файлом или каналом.

Канал (`pipe`) — особый тип файла. Если один процесс откроет канал для чтения, а второй — для записи, то всё, что запишет второй, будет читать первый. Когда пишущий процесс канал закроет, читающий увидит «конец файла».

Перенаправление стандартного ввода

```
$ ./program args... < input.txt
```

Если исходно программа запрашивала у пользователя ввод с клавиатуры, то теперь она читает файл `input.txt`.

Перенаправление стандартного вывода:

```
$ ./program args... > output.txt
```

На экран ничего не выводится, а то, что программа печатает на экран, на самом деле пишется в файл `output.txt`. Если до запуска программы файл `output.txt` существовал, то он перезапишется. Если использовать знак `>>`, то запись будет осуществляться в конец файла.

Пример:

```
$ echo hello > hello.txt
$ echo world >> hello.txt
```

Перенаправление стандартного потока ошибок:

```
$ ./program args... 2> errors.txt
```

Программа может выводить на stdout полезные данные, а на stderr ошибки. Тогда, если stdout перенаправлен и возникнет что-то, о чём нужно уведомить пользователя, (а) сообщение об ошибке пользователь увидит (stderr по-прежнему связан с экраном), (б) сообщения об ошибках не перепутаются с полезными данными.

Пример. Программа cat, предназначенная для конкатенации файлов, получает в командной строке имена файлов и их содержимое последовательно пишет на stdout. Перенаправив stdout, мы получим файл с конкатенацией содержимого исходных файлов:

```
$ cat header.txt body.txt footer.txt > document.txt
```

Несколько программ можно объединять в конвейер:

```
$ prog1 args... | prog2 args... | prog3 args
```

В этом случае stdout каждой из программ будет связан со стандартным вводом (stdin) следующей программы.

Задача: найти в файлах с расширением .c все строки, содержащие #include и вывести их в алфавитном порядке и без повторяющихся строк:

```
$ cat *.c | grep "#include" | sort | uniq
```

Многие утилиты в unix-подобных ОС или принимают список файлов в качестве аргументов, или, если файлов не указано, читают стандартный ввод.

Написание скриптов

Исполнимые файлы в UNIX-подобных ОС отличаются от обычных флагом исполнимости. У каждого файла есть три набора флагов rwxrwxrwx, r — доступ на чтение, w — доступ на запись, x — доступ на исполнение. Первая группа — права владельца файла, вторая — права группы пользователей, владеющих файлом, третья — права для всех остальных.

Права доступа типичного файла: rw-r--r--, т.е. владелец может в файл писать, все остальные — только читать.

Права доступа: --x--x--x — файл нельзя прочитать, но можно запустить.

Установка и сброс атрибутов выполняется командой chmod:

```
chmod +x prog         # добавить флаг исполнимости
chmod +w file.dat     # разрешить запись
chmod -w file.dat     # запретить запись
chmod go-r file.dat   # запретить чтение (r) группе (g) и всем остальным (o)
```

Исполнимые файлы могут быть либо двоичными в формате исполнимых файлов ОС (ELF для Linux, формат Mach-O для macOS),

либо **скриптами (сценариями).** Скрипты должны начинаться со строки с указанием интерпретатора (так называемый shebang).

```
#!/путь/до/интерпретатора
```

Для Bash это

```
#!/bin/bash
```

или

```
#!/bin/sh
```

Если shebang не указан, то на Linux по умолчанию вызывается `/bin/sh`.

В сценарии последовательно записываются команды Bash. Среди них могут быть как вызовы программ, так и встроенные команды включая операторы.

Процессы при завершении устанавливают код возврата. В языке Си кодом возврата является возвращаемым значением функции `main()`:

```
int main() {
  return 100;
}
```

По соглашению, успешное завершение работы соответствует коду `0`, неуспешное — ненулевому числу, при этом разные значения соответствуют разным ошибкам.

Несколько команд можно объединять знаками `(`, `)`, `&&`, `||`.

```
prog1 && prog2
```

Код возврата будет нулевым, если обе программы завершились успешно. Если `prog1` завершилась неуспешно, `prog2` даже не запустится.

```
./gen-source > source.c && gcc source.c
```

```
prog1 || prog2
```

Соответственно, логическое ИЛИ. `prog2` вызовется, если `prog1` завершилась неуспешно.

```
./get-info > info.txt || rm info.txt
```

Команды в Bash разделяются или переводом строки, или знаком `;`.

Оператор ветвления имеет вид:

```
if команда аргументы... ; then
  команда
  ...
elif команда аргументы... ; then
  команда
  ...
```

```
else
  команда
  ...
fi
```

Код после `then` выполняется, если команда в условии завершилась успешно.

`grep` возвращает успех, если что-то нашлось, иначе — неуспех.

```
if grep ERROR file.txt > /dev/null; then
  echo Были ошибки
else
  echo Ошибок не было
fi
```

Встроенные команды `true` и `false` они всегда завершаются, соответственно, успешно и неуспешно.

Цикл `while`

```
while команда аргументы...; do
  команда
  ...
done
```

Цикл `for`:

```
for перем in строка...; do
  команда $перем
  ...
done
```

Переменные окружения. В UNIX есть понятие переменных окружения — набора некоторых глобальных переменных, которые хранят некоторые строки.

Например, переменная `PATH` хранит список стандартных путей, в которых ищутся исполнимые файлы. Типичное содержимое: `/bin:/usr/bin:/usr/local/bin` (пути разделяются двоеточием). `HOME` — путь к домашнему каталогу пользователя, `USER` — имя пользователя.

В Bash можно устанавливать переменные среды при помощи синтаксиса

```
VAR=VALUE
```

Получить значение переменной можно при помощи `$VARNAME` или `${VARNAME}`.

```
MY_NAME="Vasiliy Pupkin"
echo $MY_NAME
```

Можно писать так:

```
RESULT=false
```

```
if …; then
  …
  RESULT=true
  …
fi

if $RESULT; then
  …
fi
```

Особые переменные среды:

- `$!` — PID процесса, запущенного в фоне предыдущей командой.
- `$?` — код возврата предыдущей команды.
- `$1`, `$2`, … — параметры командной строки скрипта.
- `$*` и `$@` — список всех параметров. Посмотрите в руководстве, чем они отличаются. Желательно их указывать в кавычках `"$1"`, тогда при раскрытии параметры с пробелами не рассыпятся на кусочки.
- `$0` — имя скрипта.

Команда `shift` (встроенная в Bash) сдвигает аргументы командной строки: `$2` становится `$1`, `$3` → `$2` и т.д., значение `$1` теряется.

Программы-фильтры — это программы, которые принимают какой-то текст на `stdin`, фильтруют его как-то и выводят на `stdout`. Либо, если указаны файлы в командной строке, они читают каждый файл последовательно.

Программы-фильтры как правило используются в конвейерах.

- `sort` — сортирует строки в алфавитном порядке (в соответствии с кодами символов). У команды есть множество дополнительных ключей, выполняющих числовую сортировку, сортировку в обратном порядке, сортировку по номеру поля и т.д. Ключи можно посмотреть в `man sort` или `sort --help`.
- `uniq` — удаляет последовательные повторяющиеся строки. Комбинация `sort | uniq` позволяет получить поток, в котором нет вообще повторяющихся строк.
- `grep`, `egrep` — выбирает из потока строки, содержащие некоторый шаблон.
- `head [-N]`, `tail [-N]` — выбирают первые `N` строк или последние `N` строк файла или потока. По умолчанию `N` равно 10.
- `sed команда` — позволяет выполнять некоторые операции по редактированию потока или файла. Наиболее распространённое использование — делать замену одной подстроки на другую `sed 's/from/to/'` (заменяется первое вхождене), `sed 's/from/to/g'` — все вхождения. Пример: `man cat | sed 's/cat/dog/g'`
- `more` — выводит текст постранично, можно перематывать только вперёд (POSIX).

- `less` — выводит текст постранично, его можно перематывать вверх и вниз стрелками (утилита проекта GNU). Утилиты `more` и `less` используются в конце конвейера.
- `cat` — ничего не делает с потоком, но может в поток положить содержимое нескольких файлов.
- `tac` — выводит поток задом наперёд.
- `awk` — скриптовый язык программирования, ориентированный на фильтрацию потока. Описание языка: `man awk`.

Команда test (она же [)

Команда `test` позволяет проверить некоторое условие, относящееся к файлам или значениям. Если условие верное, код возврата нулевой, иначе — ненулевой.

Может быть вызвана как `test условие` или как `[ условие ]`. Примеры:

```
[ -e filename ]    # проверяет, существует ли файл
[ 100 -lt 200 ]    # проверяет, что 100 меньше 200
[ "ab" != "cd" ]   # проверяет, что строка "ab" не равно "cd"
[ 100 -ne 200 ]    # числа не равны

[ -e filename.txt ] && [ 100 -ge 100 ]
[ -e filename.txt -a 100 -ge 100 ]
test -e filename.txt -a 100 -ge 100
```

Ключи команды `test` см. в `man test` (для самостоятельного изучения).

Но есть и особый синтаксис. В Bash есть встроенная команда `[[`, которая по поведению эквивалентна `test`, но обрабатывается самим Bash.

В Bash можно объявлять функции. Синтаксис:

```
funcname() {
  тело функции
}
```

Функция вызывается как обычная команда, параметры функции доступны в её теле как `$1`, `$2`, ….

Если команду записать внутри обратных кавычек или внутри скобок `$(…)`, то весь вывод программы на `stdout` превратится в последовательность аргументов.

```
echo `cat file.txt`
echo $(cat file.txt)
```

Этот синтаксис часто используют при написании функций. Функция возвращает результат на `stdout`, её вызывают обратными кавычками или `$(…)` и получают её вывод как строку.

Bash может вычислять арифметические выражения: `$((2 + 2 * 2))` → 6.

Встроенная команда `read VARNAME` считывает из стандартного ввода одну строчку и кладёт её в переменную `VARNAME`. Если достигнут конец файла, программа завершается неуспешно. Использование:

```
... | while read X; do
  ...
  ...
done
```

Пример. Рекурсивный обход папок:

```
#!/bin/bash

rec() {
  if [ -d "$1" ]; then
    ls "$1" | while read name; do
      rec "$1/$name"
    done
  else
    echo File "$1"
  fi
}

rec "$1"
```

Для того, чтобы запустить интерпретатор скриптового языка, доступный в PATH, но при этом располагающемся по неизвестному пути, в начало файла добавляют `/usr/bin/env`:

```
#!/usr/bin/env python

# дальше код какой-то на Python
...
```

```
#!/usr/bin/env node

// Дальше код на JavaScript
...
```

Подходы к написанию скриптов на интерпретируемых языках программирования.