# Основы информатики РК1

## 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 некоторых задач.

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 имеет конечное количество шагов.
5. Массовость - один алгоритм применим к некоторому классу задач.
6. Результативность - алгоритм должен выдавать результат.

**Компьютерная программа** - алгоритм, записанный на некотором языке программирования.

**Язык программирования** - формальный язык, предназначенный для записи компьютерных программ.

**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:

1. **Императивное программирование** - способ записи программ, в котором указывается последовательность действий, изменяющих состояние вычислительной среды (изменение памяти, ввод, вывод).
2. **Декларативное программирование** - способ записи программ, в котором описываются взаимосвязь между данными; описывается цель, а не последовательность шагов для её достижения.
3. **Метапрограммирование** - программа рассматривается как данные.

Основные подходы императивной парадигмы:

1. **Структурное программирование** - программа рассматривается как набор "фрагментов" кода (в т.ч. и вложенных друг в друга), имеющих один вход и один

выход. Конструкции структурного ЯП: примитивный оператор (присваивание, вызов процедуры), ветвление, цикл.
2. **Процедурное программирование** - в рамках этого подхода программа рассматривается как набор отдельных подпрограмм, которые могут вызвать друг друга.
3. **Объектно-ориентированное программирование** - программа рассматривается как набор взаимодействующих объектов, объекты сочетают в себе состояние (данные) и поведение (связанные с объектом функции).

Основные подходы декларативной парадигмы:

1. **Функциональное программирование** - алгоритм описывается как набор функций в математическом смысле (функция - отображение входных данных на выходные). Единица декомпозиции программы - функция
   **Чистые функции** - детерминированные функции без побочного эффекта.
2. **Логическое программирование (Prolog)** - алгоритм описывает взаимосвязь между понятиями; выполнение программы сводится к выполнению запросов.

Основные подходы метапрограммирования парадигмы:

1. **Программы пишут программы** - макросы, генераторы кода, метапрограммирование шаблонов в C++
2. **Программы взаимодействуют с вычислительной средой** - рефлексия или интроспекция, программа анализирует свойства самой себя.

**Подпрограмма (subroutine)** - именованный блок кода; вызывающая программа приостанавливается, управление передаётся подпрограмме. При завершении работы подпрограммы вызывающая программа возобновляет свою работу. Синонимы подпрограммы: процедура, функция, метод.

**Сопрограмма (coroutine)**, в отличие от подпрограммы, работает поочерёдно с вызывающей - одна сопрограмма приостанавливает свою работу, передавая управление другой.
*Примеры сопрограмм* в различных ЯП: оператор yield в Python, go-программы в языке Go, механизмы async/await в различных языках тоже можно рассматривать как разновидность сопрограмм.

## Списки

**Список** — последовательность термов (возможно пустая) в круглых скобках.
**Терм** — это либо атом, либо список.
**Атом** — имя переменной, число, символ или строка.

Процедуры, которые возвращают логическое значение (т.е. `#t` или `#f`), называются предикатами.

**Голова** — первый элемент непустого списка.
**Хвост** — список, полученный путем исключения из непустого списка первого элемента.

Объект, который строится процедурой `cons` — т.н. cons-ячейка или пара. Аргументами процедуры `cons` могут быть любые объекты. Правильный список — это или пустой список, или cons-пара, вторым элементом которой является правильный список.

## Вычислительная сложность

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

$T(\langle данные \rangle)$ — функция, возвращающая точное значение времени работы программы на конкретных входных данных.

Асимптотическая оценка $O(f(\langle данные \rangle))$ показывает, что функция $T(\bullet)$ при росте входных данных ведёт себя как функция $f(\bullet)$ с точностью до некоторого постоянного сомножителя. Т.е. существует такое $k$, что

$$\lim_{|data|\to\infty} \frac{T(data)}{f(data)} = k \quad \text{или} \quad T(data) \approx k \times f(data)$$

при росте аргумента $data$. Здесь $|data|$ означает размер входных данных (например, длина списка).

Сложность некоторых процедур в Scheme:

- `(map f xs)` - $O(|xs| \times T(f))$, где $T(f)$ - среднее время работы (f x).
- `(length xs)` — $O(|xs|)$.
- `(append xs ys)` — $O(|xs|)$,
- `(append xs ys zs)` — $O(|xs| + |ys|)$,
- `(append xs ys zs ts)` — $O(|xs| + |ys| + |zs|)$,
- `(reverse xs)` — $O(|xs|)$,
- `(list? xs)` — $O(|xs|)$,
- `(list-ref xs i)` — $O(\min(|xs|, i))$

## Идиома

**Идиома** — устойчивый способ сочетания базовых конструкций языка, выражающий некоторое высокоуровневое понятие.

*Пример*. В языке Си нет цикла со счётчиком (как, например, в Паскале), цикл `for` — цикл с предусловием. Понятие цикла со счётчиком в языке Си принято выражать как `for (i = 0; i < N; i++)`

## Типы данных

**Типы 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.

**Система типов** - совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным конструкциям, составляющим программу — переменные, выражения, функции и модули.

**Система типов по Пирсу** - разрешимый синтаксический метод доказательства отсутствия определённых поведений программы путём классификации конструкции в соответствии с видами вычисляемых значений.

Классификация систем типов:

1. **Наличие системы типов**: есть/нет
   *Нет:* ASM, FORTH, B.
   *Есть:* все остальные языки
2. Типизация: **статическая/динамическая**
   *Статическая:* C, C++, Java, Haskell, Rust, Go.
   *Динамическая:* Scheme, JavaScript, Python
3. Типизация: **явная/неявная**
   *Явная (явно записывается):* C, C++, Java.
   *Неявная (можно не записывать):* C++(auto), Go(когда тип не указан), Rust, Haskell
4. Типизация: **сильная/слабая**
   *Сильная* (неявные преобразования типов запрещены): Scheme, Python, Haskell.
   *Слабая* (неявные преобразования допустимы): JavaScript, C, Perl, PHP (`'1000' * 5 -> 5000`)

Первая классификация типов:

1. **Простые** - неделимые порции данных: число, символ, литера (Scheme: `52`, `'a` C: `int`, `"a"`).
2. **Составные** - содержащие значения других типов: cons-ячейки, список, вектор, строка (Scheme: `(1, 2, 3)` C: `{1, 2, 3}`).

Вторая классификация типов:

1. **Встроенные типы данных** - уже заранее есть в языке.
2. **Пользовательский** - их определяет пользователь.
   Пользовательские типы данных часто представляют как списки, первым элементом которых является символ с именем типа, а остальные — хранимые значения.

## Свёртка

**Свёртка (fold)** - 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)`

Примеры в Scheme:

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

## Рекурсия, хвостовая рекурсия

**Рекурсия** - задача делится на меньшие подзадачи, подобные исходной.

**Хвостовая рекурсия** - это форма рекурсии, при которой рекурсивный вызов является последним, результат этого вызова становится результатом работы функции.
**Особенность в Scheme:** Хвостовая рекурсия в языке Scheme эквивалента итерации по вычислительным затратам.

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

**Пример:** Вычисление остатка от деления.

```
(define (my-remainder a b)
  (if (< a b)
      a
      (my-remainder (- a b) b)))
```

## Задачи для РК

### Задача 1

На языке Scheme напишите определение процедуры `(scan-integer str)`, принимающую строку `str` и выводящую десятичное целое число, если оно было записано в этой строке (со знаком или без), иначе - `#f`.

```
(scan-integer "123")  -> 123
(scan-integer "-123") -> -123
(scan-integer "1/2")  -> #f
```

```
(define (scan-integer str)
  (let ((num (string->number str)))
    (if (and num (integer? num))
        num
        #f)))
```

## Задача 2

На языке Scheme напишите определение процедуры `and-fold`, принимающей переменное число аргументов и выполняющие их свертку с помощью операции `логическое И`.

```
(apply and-fold '(1 #f 2))  -> #f
(apply and-fold '(#t #t 5)) -> 5
```

```
(define (and-fold . xs)
  (if (= (length xs) 1)
      (car xs)
      (and (car xs) (apply and-fold (cdr xs)))))
```

## Задача 3

Напишите определения, необходимые для того, чтобы в программу на языке Scheme ввести тип данных `точка на плоскости`. Определите операции переноса точки и отражения точки от осей `Ox` и `Oy`.

```
(define (point x y) (list x y))
(define (shiftX point x) (set-car! point (+ (car point) x)) point)
(define (shiftY point y) (set-cdr! point (+ (cdr point) y)) point)
(define (revx point) (set-cdr! point (* (cdr point) -1)) point)
(define (revy point) (set-car! point (* (car point) -1)) point)
(define (point? point)
  (if (and (number? (car point)) (number? (car (cdr point))))
      #t
      #f))
```

## Задача 4

Реализуйте процедуру `(improper->proper lst)`, принимающую неправильный список `(improper list)` и преобразующую его в правильный список `(proper list)`. Неправильные списки могут быть вложенными. Правильный список, переданный процедуре, возвращается "как есть".

```
(improper->proper '(a b . c))           -> (a b c)
(improper->proper '(a (1 2 . 3) . c))   -> (a (1 2 3) c)
(improper->proper '(0 1 2 . (a b . c))) -> (0 1 2 (a b c))
(improper->proper '(a b (c d) e f))     -> (a b (c d) e f)
```

```
(define (improper->proper xs)
  (cond ((not (pair? xs)) xs)
        ((and (not (list? xs)) (not (pair? (cdr xs))))
         (cons (improper->proper (car xs)) (cons (cdr xs) '())))
        (else (cons (improper->proper (car xs)) (improper->proper
(cdr xs))))))
```