1-ая лекция

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 делятся на три большие группы:

1. **Императивное** программирование — способ записи программ, в котором указывается последовательность действий, изменяющих состояние вычислительной среды *(изменение памяти, ввод, вывод)*:
    a. **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.
    b. **Процедурное** программирование — в рамках этого подхода программа рассматривается как набор отдельных подпрограмм, которые могут вызывать друг друга.
    c. **Объектно-ориентированное** программирование — программа рассматривается как набор взаимодействующих объектов, объекты сочетают в себе состояние *(данные)* и поведение *(связанные с объектом функции)*.
2. **Декларативное** программирование — способ записи программ, в котором описываются взаимосвязь между данными; *описывается цель, а не последовательность шагов для её достижения*:
    a. **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.
    b. **Логическое** программирование *(Prolog)* — алгоритм описывает взаимосвязь между понятиями; выполнение программы сводится к выполнению запросов. Единица декомпозиции программы — предикат.
3. **Метапрограммирование** — программа рассматривается как данные:
    a. **Программы пишут программы** — макросы, генераторы кода, метапрограммирование шаблонов в *C++*
    b. Программы **взаимодействуют с вычислительной средой** — рефлексия или интроспекция, программа анализирует свойства самой себя.

Подпрограмма *(subroutine)* — именованный блок кода; вызывающая программа приостанавливается, управление передается подпрограмме. При завершении работы подпрограммы вызывающая программа возобновляет свою работу.
Сопрограмма *(coroutine)*, в отличие от подпрограммы, работает поочерёдно с вызывающей — одна сопрограмма приостанавливает свою работу, передавая управление другой.

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.

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**.

2-ая лекция

Замыкание — ситуация, когда из процедуры возвращается другая процедура, *«запоминающая»* локальные переменные окружающей процедуры.

Рекурсия — делим задачу на меньшие подзадачи, подобные исходной.
Итерация — задача делится на некоторое количество одинаковых подзадач, одинаковых шагов, приближающих к цели.
Хвостовой вызов — вызов, который является последним, результат этого вызова становится результатом работы функции.
Хвостовая рекурсия в языке *Scheme* **эквивалента** итерации по вычислительным затратам.

3-ая лекция

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 — 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 $T(\cdot)$ при росте входных данных ведёт себя как функция $f(\cdot)$ 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).

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.

4-ая и 5-ая лекция

В языке *Си* есть понятие статическая переменная — **глобальная переменная**, видимость которой ограничена одной функцией.

Свертка — объединение нескольких значений одной операцией.
Примеры: вычислить сумму нескольких чисел, произведение нескольких чисел и т.д.
Свертка может быть *правой* и *левой*.

6-ая лекция

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

Для типов данных языка *Scheme* обычно определены **четыре вида операций**:

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

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 определенных поведений программы путем классификации конструкции в соответствии с видами вычисляемых значений.

Классификации систем типов:

1. Наличие системы типов: **есть/нет:**
    - **Нет**: язык ассемблера, язык *FORTH*, язык *B (Би)* — предшественник *Си*.
    - **Есть**: все остальные языки.
2. Типизация **статическая/динамическая:**
    - **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*.
3. Типизация **явная/неявная:**
    - **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.
4. 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 ее на число, она неявно преобразуется в число: `'1000' * 5 → 5000`. Примеры языков: *JavaScript*, *Си*, *Perl*, *PHP*.

Алгебра типов:

1. **Декартово произведение** (прямое произведение) двух множеств X и Y есть такое множество $X \times Y$, элементами которого являются упорядоченные пары (x, y) для всевозможных $x \in X$ и $y \in Y$.
    - В функциональных языках программирования **алгебраические типы данных *(АТД)*** — типы, описанные как размеченные объединения декартовых произведений.
2. **Размеченное** объединение *(дизъюнктивное объединение, несвязное объединение, несвязная сумма)* множеств Ai, $1 \le i \le N$ — множество пар (ai,j, i), где $ai,j \in Ai$:

$$\bigsqcup_{i \in I} A_i = \bigcup_{i \in I} \{(x, i) | x \in A_i\}$$

Первый компонент пары — элемент исходного

множества, второй — индекс множества, из которого был взят элемент. Таким образом, даже если в исходных множествах есть одинаковые элементы, у них в размеченном объединении будут разные индексы. Можность $|A \sqcup B|$ равна сумме мощностей $|A|$ + $|B|$.

3. Типы строятся из следующих конструкций:
    - **Единичный** тип (unit) — имеет только одно значение.
    - **Декартово произведение** типов.
    - **Размеченное объединение** типов.
    - **Встроенный примитивный** тип, например, число, строка и т.д.
    - *Частный случай:* размеченное объединение единичных типов — перечисление.

Литерный тип (character):

1. 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.)*.
   Литерный тип хранит в себе **печатные знаки**, т.е. знаки, которые вводятся с клавиатуры и выводятся на экран. Из литер состоят строки.
2. **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.
3. **Строка** — последовательность литер. Строка может быть пустой. **Литерал** — текст, записанный в двойных кавычках. Внутри строки допустимы стандартные escape-последовательности языка *Си*.

Числовые типы:

1. **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}*)*.

2. **Вещественные** числа имеют ограниченную точность *(мантисса имеет конечное число значимых цифр)*. Скорее всего, они будут представлены как `double`.
3. **Точные** числа — целые числа, рациональные числа и комплексные числа, обе компоненты которых тоже точные *(т.е. целые или рациональные)*.
4. **Неточные** числа — вещественные числа или комплексные с вещественными компонентами.

Тип данных vector:

1. **Списки** — основные структуры данных в языках семейства *Lisp*. В *Scheme* они не примитивный тип, а надстройка над cons-ячейками
2. **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.

Простые макросы:

Макрос — инструмент переписывания кода.
Процедура применяется к значениям во время выполнения и порождает новое значение.
Макрос применяется к фрагменту кода и порождает код *до его выполнения*.

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

Цикл разработки через тестирование:

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

эффективности). В процессе рефакторинга ни один из модульных тестов сломаться не должен.

Продолжительность одного цикла — около минуты.

7-ая лекция

Символьные вычисления — это преобразования символьных данных: алгебраические выкладки, трансляция языков программирования и т.д.
ЛИСП был одним из первых языков, ориентированных на символьные вычисления.

Символьный тип данных — это «зацитированное», «замороженное» имя `'hello`
Одинаковые символы **всегда равны** — их можно сравнивать на равенство предикатом `eq?`.

Нельзя сказать, что у символов есть свои литералы. Но символы создаются операцией цитирования.
Операция цитирования — это особая форма `(quote ...)`, которая принимает терм и *«цитирует»* его — все идентификаторы в нём становятся символами, остальные атомарные значения остаются как есть, выражения превращаются в списки.

Квазицитирование позволяет внутри цитаты вычислять какие-то значения. Обозначается она обратной кавычкой *(на клавиатуре на букве «ё»)*. Для «расцитирования» внутри квазицитаты перед термом записывается знак *«запятая»*.

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.
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, embedded DSL, EDSL). В Scheme для определения DSL'ей часто используются макросы.*

Макрос — это инструмент переписывания кода. Т.е. способ создать новую языковую конструкцию на основании имеющихся.
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.
Макросы в *Scheme* **гигиенические**, т.е. о конфликте имен при их раскрытии беспокоиться не нужно.

Функции и примеры применений

Лекция 1: Основные понятия. Язык программирования Scheme

- **define** — определение переменной или функции. Пример: `(define pi (* 4 (atan 1))) ; определение переменной pi`
- **if** — условное выражение, выбирает результат на основе проверки условия. Пример: `(if (> x 0) + -)`
- **lambda** — создание анонимных функций. Пример: `(lambda (x y) (+ x y)) ; функция сложения двух чисел`
- **quote** — операция цитирования, не вычисляет выражение. Пример: `(quote (1 2 3)) ; возвращает список (1 2 3)`
- **car** — возвращает первый элемент списка. Пример: `(car (list 1 2 3)) ; результат: 1`
- **cdr** — возвращает хвост списка (все элементы после первого). Пример: `(cdr (list 1 2 3)) ; результат: (2 3)`

Лекция 2: Процедуры высшего порядка

- **lambda** — создание анонимных функций, применимо для процедур высшего порядка. Пример: `(lambda (x y) (+ x y))`

- ➔ **define** — используется для определения функций, также применимо к высшим порядкам. Пример: `(define (f x y) (+ x y))`
- ➔ *let, let, letrec** — для создания локальных переменных. Пример: `(let ((x 1) (y 2)) (+ x y)) ; let создает локальные переменные`
- ➔ **map** — функция для применения другой функции ко всем элементам списка. Пример : `(map square '(1 2 3 4)) ; результат: (1 4 9 16)`
- ➔ **apply** — применяется к функции и списку аргументов. Пример: `(apply + '(1 2 3)) ; результат: 6`

Лекция 3: Списки, вычислительная сложность, равенство

- ➔ **list** — создание списка. Пример: `(list 1 2 3) ; результат: (1 2 3)`
- ➔ **cons** — добавляет элемент в начало списка. Пример: `(cons 1 '(2 3)) ; результат: (1 2 3)`
- ➔ **car, cdr** — работа с головной частью и хвостом списка (упомянуто ранее). Пример: `(car (list 1 2 3)) ; результат: 1`
- ➔ **length** — возвращает длину списка. Пример: `(length '(1 2 3)) ; результат: 3`
- ➔ **append** — конкатенация списков. Пример: `(append '(1 2) '(3 4)) ; результат: (1 2 3 4)`
- ➔ **reverse** — переворачивает список. Пример: `(reverse '(1 2 3)) ; результат: (3 2 1)`
- ➔ **list-ref** — обращение к элементу списка по индексу. Пример: `(list-ref '(1 2 3) 1) ; результат: 2`
- ➔ **pair?** — проверяет, является ли аргумент парой (списком). Пример: `(pair? '(1 . 2)) ; результат: #t`
- ➔ **list?** — проверяет, является ли аргумент правильным списком. Пример: `(list? '(1 2 3)) ; результат: #t`

Лекция 4: Императивное программирование на языке Scheme

- ➔ **set!** — оператор присваивания, изменяет значение переменной. Пример: `(set! counter (+ counter 1))`

- ➔ **begin** — выполняет несколько действий в определенном порядке. Пример:
  `(begin (display "Hello, ") (display "World!"))`
- ➔ **do** — цикл с возможностью инициализации переменных, проверки условия и выполнения действий. Пример:
  `(do ((i 0 (+ i 1))) ((> i 5) i) (display i))`

Лекция 5: Понятие свертки

- ➔ **fold (свертка)** — объединение нескольких элементов с помощью одной операции (например, сложение, умножение).
  - **Левая** свёртка: Пример:
    `(foldl + 0 '(1 2 3 4)) ; результат: 10`
  - **Правая** свертка: Пример:
    `(foldr + 0 '(1 2 3 4)) ; результат: 10`
- ➔ **apply** — применяется для свёрток (ранее упомянуто). Пример:
  `(apply + '(1 2 3 4)) ; результат: 10`
- ➔ **min, max** — процедуры свертки для нахождения минимального и максимального значения. Пример:
  `(min 3 5 2 8) ; результат: 2`
  `(max 3 5 2 8) ; результат: 8`
- ➔ **string-append** — конкатенация строк. Пример:
  `(string-append "Hello" " " "World!") ; результат: "Hello World!"`
- ➔ **append** — конкатенация списков (ранее упомянуто). Пример:
  `(append '(1 2) '(3 4)) ; результат: (1 2 3 4)`
- ➔ **Процедуры сравнения** — например, операции на числа (не свертки): Пример:
  `(= 1 1 1) ; результат: #t`
  `(> 5 3 2) ; результат: #t`

Лекция 6: Типы данных и типизация. Встроенные типы данных языка Scheme

- ➔ **make-‹имя типа›** — создание нового значения определённого типа. Пример: `(define (make-circle x y r) (list 'circle x y r))`

- ➔ **‹имя типа›?** — предикат, проверяющий принадлежность значению определенному типу. Пример:
  `(define (circle? c) (and (list? c) (equal? (car c) 'circle)))`
- ➔ **vector-set!, string-append** — операции модификации данных (модификаторы). Пример:
  `(vector-set! v 2 'a) ; устанавливает 'a' в индекс 2 вектора`
- ➔ **char?** — предикат для проверки, является ли объект литерой. Пример: `(char? #\a) ; результат: #t`
- ➔ **char->integer** и **integer->char** — преобразования между символами и их числовыми кодами. Пример:
  `(char->integer #\A) ; результат: 65`
  `(integer->char 65) ; результат: #\A`
- ➔ **char-ci=?** — сравнение символов без учёта регистра. Пример:
  `(char-ci=? #\a #\A) ; результат: #t`

Лекция 7: Символьные вычисления и макросы

- ➔ **symbol?** — проверяет, является ли объект символом. Пример:
  `(symbol? 'hello) ; результат: #t`
- ➔ **symbol->string** — преобразует символ в строку. Пример:
  `(symbol->string 'hello) ; результат: "hello"`
- ➔ **string->symbol** — преобразует строку в символ. Пример:
- ➔ **quote** — цитирует данные, превращая выражение в список (ранее упомянуто). Пример:
  `(quote (1 2 3)) ; результат: (1 2 3)`
- ➔ **quasiquote (`) и unquote (,)** — квазицитирование с возможностью вычисления отдельных элементов. Пример:
  `` `(1 2 ,(+ 3 4)) ; результат: (1 2 7) ``
- ➔ **eval** — выполняет выражение, записанное в виде данных. Пример: `(eval '(+ 1 2)) ; результат: 3`
- ➔ **assoc и member** — функции для работы с ассоциативными списками. Пример:
  `(assoc 'b '((a 1) (b 2) (c 3))) ; результат: (b 2)`
- ➔ **define-syntax и syntax-rules** — создание макросов. Пример:
  `(define-syntax when (syntax-rules () ((_ test expr ...) (if test (begin expr ...)))))`